# Table of Contents

## Welcome
- Are you a first-time Amazon Redshift user? ................................................................. 1
- Are you a database developer? ......................................................................................... 2
- Prerequisites .................................................................................................................... 3

## Amazon Redshift System Overview ............................................................................. 4
- Data warehouse system architecture ................................................................................ 4
- Performance .................................................................................................................... 6
- Columnar storage ............................................................................................................ 8

## Internal architecture and system operation .................................................................. 9
- Query processing ............................................................................................................. 10
- Client requests and execution ......................................................................................... 10
- Explain plan .................................................................................................................... 12
- Query execution steps .................................................................................................... 12

## Workload management .................................................................................................. 13

## Using Amazon Redshift with other services ................................................................. 14
- Moving data between Amazon Redshift and Amazon S3 ............................................... 14
- Using Amazon Redshift with Amazon DynamoDB ......................................................... 14
- Importing data from remote hosts over SSH ................................................................. 14
- Automating data loads using AWS Data Pipeline ......................................................... 14

## Getting Started Using Databases ................................................................................ 15
- Step 1: Create a database .............................................................................................. 15
- Step 2: Create a database user ...................................................................................... 16
- Delete a database user ................................................................................................... 16
- Step 3: Create a database table ...................................................................................... 16
- Insert data rows into a table ......................................................................................... 17
- Select data from a table ............................................................................................... 17
- Step 4: Load sample data .............................................................................................. 18
- Step 5: Query the system tables .................................................................................... 20
- View a list of table names .............................................................................................. 21
- View database users ...................................................................................................... 21
- View recent queries ....................................................................................................... 22
- Determine the process ID of a running query ............................................................... 22
- Step 6: Cancel a query .................................................................................................. 23
- Cancel a query from another session ........................................................................... 24
- Cancel a query using the Superuser queue .................................................................. 24
- Step 7: Clean up your resources .................................................................................. 25

## Amazon Redshift Best Practices .................................................................................. 26
- Best practices for designing tables ................................................................................ 26
- Take the Tuning Table Design tutorial ......................................................................... 27
- Choosing the best sort key ............................................................................................ 27
- Choosing the best distribution style .............................................................................. 27
- Use automatic compression .......................................................................................... 28
- Define constraints .......................................................................................................... 28
- Use the smallest possible column size .......................................................................... 28
- Using date/time data types for date columns ............................................................... 28

## Best practices for loading data ..................................................................................... 29
- Take the Loading Data tutorial ...................................................................................... 29
- Take the Tuning Table Design tutorial ......................................................................... 29
- Use a COPY command to load data ............................................................................ 29
- Use a single COPY command ....................................................................................... 30
- Split your load data into multiple files .......................................................................... 30
- Compress your data files with gzip or lzop ................................................................. 30
- Use a manifest file ......................................................................................................... 30
- Verify data files before and after a load ........................................................................ 30
- Using a multi-row insert .............................................................................................. 31
# Amazon Redshift Database Developer Guide

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step 5: Run the COPY Commands</td>
<td>69</td>
</tr>
<tr>
<td>COPY Command Syntax</td>
<td>69</td>
</tr>
<tr>
<td>Loading the SSB Tables</td>
<td>70</td>
</tr>
<tr>
<td>Step 6: Vacuum and Analyze the Database</td>
<td>81</td>
</tr>
<tr>
<td>Next Step</td>
<td>81</td>
</tr>
<tr>
<td>Step 7: Clean Up Your Resources</td>
<td>81</td>
</tr>
<tr>
<td>Next</td>
<td>82</td>
</tr>
<tr>
<td>Summary</td>
<td>82</td>
</tr>
<tr>
<td>Managing Database Security</td>
<td>83</td>
</tr>
<tr>
<td>Amazon Redshift security overview</td>
<td>83</td>
</tr>
<tr>
<td>Default database user privileges</td>
<td>84</td>
</tr>
<tr>
<td>Superusers</td>
<td>84</td>
</tr>
<tr>
<td>Users</td>
<td>85</td>
</tr>
<tr>
<td>Creating, altering, and deleting users</td>
<td>85</td>
</tr>
<tr>
<td>Groups</td>
<td>86</td>
</tr>
<tr>
<td>Creating, altering, and deleting groups</td>
<td>86</td>
</tr>
<tr>
<td>Schemas</td>
<td>86</td>
</tr>
<tr>
<td>Creating, altering, and deleting schemas</td>
<td>87</td>
</tr>
<tr>
<td>Search path</td>
<td>87</td>
</tr>
<tr>
<td>Schema-based privileges</td>
<td>87</td>
</tr>
<tr>
<td>Example for controlling user and group access</td>
<td>87</td>
</tr>
<tr>
<td>Designing Tables</td>
<td>90</td>
</tr>
<tr>
<td>Choosing a column compression type</td>
<td>90</td>
</tr>
<tr>
<td>Compression encodings</td>
<td>91</td>
</tr>
<tr>
<td>Testing compression encodings</td>
<td>97</td>
</tr>
<tr>
<td>Example: Choosing compression encodings for the CUSTOMER table</td>
<td>99</td>
</tr>
<tr>
<td>Choosing a data distribution style</td>
<td>101</td>
</tr>
<tr>
<td>Data distribution concepts</td>
<td>101</td>
</tr>
<tr>
<td>Distribution styles</td>
<td>102</td>
</tr>
<tr>
<td>Viewing distribution styles</td>
<td>102</td>
</tr>
<tr>
<td>Evaluating query patterns</td>
<td>103</td>
</tr>
<tr>
<td>Designating distribution styles</td>
<td>103</td>
</tr>
<tr>
<td>Evaluating the query plan</td>
<td>104</td>
</tr>
<tr>
<td>Query plan example</td>
<td>106</td>
</tr>
<tr>
<td>Distribution examples</td>
<td>111</td>
</tr>
<tr>
<td>Choosing sort keys</td>
<td>113</td>
</tr>
<tr>
<td>Defining constraints</td>
<td>113</td>
</tr>
<tr>
<td>Analyzing table design</td>
<td>113</td>
</tr>
<tr>
<td>Loading Data</td>
<td>117</td>
</tr>
<tr>
<td>Using COPY to load data</td>
<td>117</td>
</tr>
<tr>
<td>Preparing your input data</td>
<td>118</td>
</tr>
<tr>
<td>Loading data from Amazon S3</td>
<td>119</td>
</tr>
<tr>
<td>Loading data from Amazon EMR</td>
<td>127</td>
</tr>
<tr>
<td>Loading data from remote hosts</td>
<td>134</td>
</tr>
<tr>
<td>Loading from Amazon DynamoDB</td>
<td>140</td>
</tr>
<tr>
<td>Verifying that the data was loaded correctly</td>
<td>142</td>
</tr>
<tr>
<td>Validating input data</td>
<td>143</td>
</tr>
<tr>
<td>Automatic compression</td>
<td>143</td>
</tr>
<tr>
<td>Optimizing for narrow tables</td>
<td>145</td>
</tr>
<tr>
<td>Default values</td>
<td>145</td>
</tr>
<tr>
<td>Troubleshooting</td>
<td>146</td>
</tr>
<tr>
<td>Updating with DML</td>
<td>150</td>
</tr>
<tr>
<td>Updating and inserting</td>
<td>150</td>
</tr>
<tr>
<td>Merge Method 1: Replacing existing rows</td>
<td>151</td>
</tr>
<tr>
<td>Merge Method 2: Specifying a column list</td>
<td>151</td>
</tr>
<tr>
<td>Creating a temporary staging table</td>
<td>151</td>
</tr>
<tr>
<td>Performing a merge operation by replacing existing rows</td>
<td>152</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>Performing a merge operation by specifying a column list</td>
<td>152</td>
</tr>
<tr>
<td>Merge examples</td>
<td>154</td>
</tr>
<tr>
<td>Performing a deep copy</td>
<td>156</td>
</tr>
<tr>
<td>Analyzing tables</td>
<td>158</td>
</tr>
<tr>
<td>ANALYZE command history</td>
<td>159</td>
</tr>
<tr>
<td>Automatic analysis</td>
<td>160</td>
</tr>
<tr>
<td>Vacuuming tables</td>
<td>161</td>
</tr>
<tr>
<td>VACUUM frequency</td>
<td>161</td>
</tr>
<tr>
<td>Sort stage and merge stage</td>
<td>161</td>
</tr>
<tr>
<td>Vacuum types</td>
<td>162</td>
</tr>
<tr>
<td>Managing vacuum times</td>
<td>162</td>
</tr>
<tr>
<td>Vacuum column limit exceeded error</td>
<td>168</td>
</tr>
<tr>
<td>Managing concurrent writes</td>
<td>169</td>
</tr>
<tr>
<td>Serializable isolation</td>
<td>169</td>
</tr>
<tr>
<td>Write and read-write operations</td>
<td>170</td>
</tr>
<tr>
<td>Concurrent write examples</td>
<td>171</td>
</tr>
<tr>
<td>Unloading Data</td>
<td>173</td>
</tr>
<tr>
<td>Unloading data to Amazon S3</td>
<td>173</td>
</tr>
<tr>
<td>Unloading encrypted data files</td>
<td>176</td>
</tr>
<tr>
<td>Unloading data in delimited or fixed-width format</td>
<td>177</td>
</tr>
<tr>
<td>Reloading unloaded data</td>
<td>178</td>
</tr>
<tr>
<td>Tuning Query Performance</td>
<td>180</td>
</tr>
<tr>
<td>Troubleshooting Queries</td>
<td>181</td>
</tr>
<tr>
<td>Connection Fails</td>
<td>181</td>
</tr>
<tr>
<td>Query Hangs</td>
<td>181</td>
</tr>
<tr>
<td>Query Takes Too Long</td>
<td>182</td>
</tr>
<tr>
<td>Load Fails</td>
<td>183</td>
</tr>
<tr>
<td>Load Takes Too Long</td>
<td>183</td>
</tr>
<tr>
<td>Load Data Is Incorrect</td>
<td>184</td>
</tr>
<tr>
<td>Analyzing the query plan</td>
<td>184</td>
</tr>
<tr>
<td>Simple EXPLAIN example</td>
<td>186</td>
</tr>
<tr>
<td>EXPLAIN operators</td>
<td>187</td>
</tr>
<tr>
<td>Join examples</td>
<td>188</td>
</tr>
<tr>
<td>Mapping the query plan to system views</td>
<td>190</td>
</tr>
<tr>
<td>Managing how queries use memory</td>
<td>190</td>
</tr>
<tr>
<td>Determining whether a query is writing to disk</td>
<td>191</td>
</tr>
<tr>
<td>Determining which steps are writing to disk</td>
<td>192</td>
</tr>
<tr>
<td>Monitoring disk space</td>
<td>194</td>
</tr>
<tr>
<td>Compiled code</td>
<td>195</td>
</tr>
<tr>
<td>Setting the JDBC fetch size parameter</td>
<td>196</td>
</tr>
<tr>
<td>Implementing workload management</td>
<td>196</td>
</tr>
<tr>
<td>Defining query queues</td>
<td>197</td>
</tr>
<tr>
<td>WLM queue assignment rules</td>
<td>199</td>
</tr>
<tr>
<td>Modifying the WLM configuration</td>
<td>201</td>
</tr>
<tr>
<td>Assigning queries to queues</td>
<td>202</td>
</tr>
<tr>
<td>Monitoring workload management</td>
<td>203</td>
</tr>
<tr>
<td>SQL Reference</td>
<td>205</td>
</tr>
<tr>
<td>Amazon Redshift SQL</td>
<td>205</td>
</tr>
<tr>
<td>SQL functions supported on the leader node</td>
<td>205</td>
</tr>
<tr>
<td>Amazon Redshift and PostgreSQL</td>
<td>206</td>
</tr>
<tr>
<td>Using SQL</td>
<td>212</td>
</tr>
<tr>
<td>SQL reference conventions</td>
<td>212</td>
</tr>
<tr>
<td>Basic elements</td>
<td>212</td>
</tr>
<tr>
<td>Expressions</td>
<td>235</td>
</tr>
<tr>
<td>Conditions</td>
<td>239</td>
</tr>
<tr>
<td>SQL Commands</td>
<td>256</td>
</tr>
<tr>
<td>ABORT</td>
<td>257</td>
</tr>
<tr>
<td>ALTER DATABASE</td>
<td>258</td>
</tr>
<tr>
<td>SQL Functions Reference</td>
<td>Page</td>
</tr>
<tr>
<td>--------------------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>Leader-node only functions</td>
<td>414</td>
</tr>
<tr>
<td>Aggregate functions</td>
<td>415</td>
</tr>
<tr>
<td>Bit-wise aggregate functions</td>
<td>424</td>
</tr>
<tr>
<td>Window functions</td>
<td>428</td>
</tr>
<tr>
<td>Conditional expressions</td>
<td>465</td>
</tr>
<tr>
<td>Date functions</td>
<td>471</td>
</tr>
<tr>
<td>Math functions</td>
<td>494</td>
</tr>
<tr>
<td>System Tables Reference</td>
<td>Page</td>
</tr>
<tr>
<td>-------------------------</td>
<td>------</td>
</tr>
<tr>
<td>System tables and views</td>
<td>582</td>
</tr>
<tr>
<td>Types of system tables and views</td>
<td>583</td>
</tr>
<tr>
<td>Visibility of data in system tables and views</td>
<td>583</td>
</tr>
<tr>
<td>Filtering system-generated queries</td>
<td>583</td>
</tr>
<tr>
<td>STL tables for logging</td>
<td>584</td>
</tr>
<tr>
<td>STL_ALERT_EVENT_LOG</td>
<td>587</td>
</tr>
<tr>
<td>STL_BCAST</td>
<td>588</td>
</tr>
<tr>
<td>STL_COMMIT_STATS</td>
<td>590</td>
</tr>
<tr>
<td>STL_CONNECTION_LOG</td>
<td>591</td>
</tr>
<tr>
<td>STLDDLTEXT</td>
<td>591</td>
</tr>
<tr>
<td>STL_DIST</td>
<td>593</td>
</tr>
<tr>
<td>STL_DELETE</td>
<td>594</td>
</tr>
<tr>
<td>STL_EXPLAIN</td>
<td>597</td>
</tr>
<tr>
<td>STL_FILE_SCAN</td>
<td>599</td>
</tr>
<tr>
<td>STL_HASH</td>
<td>600</td>
</tr>
<tr>
<td>STL_HASHJOIN</td>
<td>602</td>
</tr>
<tr>
<td>STL_INSERT</td>
<td>603</td>
</tr>
<tr>
<td>STL_LIMIT</td>
<td>604</td>
</tr>
<tr>
<td>STL_LOAD_COMMITS</td>
<td>606</td>
</tr>
<tr>
<td>STL_LOAD_ERRORS</td>
<td>608</td>
</tr>
<tr>
<td>STL_LOADERROR_DETAIL</td>
<td>610</td>
</tr>
<tr>
<td>STL_MERGE</td>
<td>612</td>
</tr>
<tr>
<td>STL_MERGEJOIN</td>
<td>613</td>
</tr>
<tr>
<td>STL_NESTLOOP</td>
<td>614</td>
</tr>
<tr>
<td>STL_PARSE</td>
<td>615</td>
</tr>
<tr>
<td>STL_PLAN_INFO</td>
<td>616</td>
</tr>
<tr>
<td>STL_PROJECT</td>
<td>618</td>
</tr>
<tr>
<td>STL_QUERY</td>
<td>620</td>
</tr>
<tr>
<td>STL_QUERYTEXT</td>
<td>622</td>
</tr>
<tr>
<td>STL_REPLACEMENTS</td>
<td>623</td>
</tr>
<tr>
<td>STL_RETURN</td>
<td>624</td>
</tr>
<tr>
<td>STL_SAVE</td>
<td>625</td>
</tr>
<tr>
<td>STL_S3CLIENT</td>
<td>626</td>
</tr>
<tr>
<td>STL_S3CLIENT_ERROR</td>
<td>628</td>
</tr>
<tr>
<td>STL_SCAN</td>
<td>629</td>
</tr>
<tr>
<td>STL_SESSIONS</td>
<td>631</td>
</tr>
<tr>
<td>STL_SORT</td>
<td>632</td>
</tr>
<tr>
<td>STL_SSHCLIENT_ERROR</td>
<td>633</td>
</tr>
<tr>
<td>STL_STREAM_SEGs</td>
<td>634</td>
</tr>
<tr>
<td>STL_TR_CONFLICT</td>
<td>635</td>
</tr>
<tr>
<td>STL_UNDONE</td>
<td>635</td>
</tr>
<tr>
<td>STL_UNIQUE</td>
<td>636</td>
</tr>
<tr>
<td>STL_UNLOAD_LOG</td>
<td>638</td>
</tr>
<tr>
<td>STL_USERLOG</td>
<td>639</td>
</tr>
<tr>
<td>STLUTILITYTEXT</td>
<td>640</td>
</tr>
<tr>
<td>STL_VACUUM</td>
<td>641</td>
</tr>
<tr>
<td>STL_WARNING</td>
<td>643</td>
</tr>
<tr>
<td>STL_WINDOW</td>
<td>644</td>
</tr>
<tr>
<td>STL_WLM_ERROR</td>
<td>645</td>
</tr>
</tbody>
</table>
Welcome

Topics
• Are you a first-time Amazon Redshift user? (p. 1)
• Are you a database developer? (p. 2)
• Prerequisites (p. 3)

This is the Amazon Redshift Database Developer Guide.

Amazon Redshift is an enterprise-level, petabyte scale, fully managed data warehousing service.

This guide focuses specifically on using Amazon Redshift to create and manage a data warehouse. If you work with databases as a designer, software developer, or administrator, it gives you the information you need to design, build, query, and maintain the relational databases that make up your data warehouse.

Are you a first-time Amazon Redshift user?

If you are a first-time user of Amazon Redshift, we recommend that you begin by reading the following sections.

• Service Highlights and Pricing – The product detail page provides the Amazon Redshift value proposition, service highlights, and pricing.
• Getting Started – The Getting Started Guide includes an example that walks you through the process of creating an Amazon Redshift data warehouse cluster, creating database tables, uploading data, and testing queries.

After you complete the Getting Started guide, we recommend that you explore one of the following guides:

• Amazon Redshift Cluster Management Guide – The Cluster Management guide shows you how to create and manage Amazon Redshift clusters.

If you are an application developer, you can use the Amazon Redshift Query API to manage clusters programmatically. Additionally, the AWS SDK libraries that wrap the underlying Amazon Redshift API can help simplify your programming tasks. If you prefer a more interactive way of managing clusters, you can use the Amazon Redshift console and the AWS command line interface (AWS CLI). For information about the API and CLI, go to the following manuals:

• API Reference
Are you a database developer?

If you are a database user, database designer, database developer, or database administrator, the following table will help you find what you're looking for.

<table>
<thead>
<tr>
<th>If you want to ...</th>
<th>We recommend</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quickly start using Amazon Redshift</td>
<td>Begin by following the steps in the Getting Started guide to quickly deploy a cluster, connect to a database, and try out some queries. When you are ready to build your database, load data into tables, and write queries to manipulate data in the data warehouse, return here to the Database Developer Guide.</td>
</tr>
<tr>
<td>Learn about the internal architecture of the Amazon Redshift data warehouse.</td>
<td>The Amazon Redshift System Overview (p. 4) gives a high-level overview of Amazon Redshift's internal architecture. If you want a broader overview of the Amazon Redshift web service, go to the Amazon Redshift product detail page.</td>
</tr>
<tr>
<td>Create databases, tables, users, and other database objects.</td>
<td>Getting Started Using Databases (p. 15) is a quick introduction to the basics of SQL development. The Amazon Redshift SQL (p. 205) has the syntax and examples for Amazon Redshift SQL commands and functions and other SQL elements. Best practices for designing tables (p. 26) provides a summary of our recommendations for choosing sort keys, distribution keys, and compression encodings.</td>
</tr>
<tr>
<td>Learn how to design tables for optimum performance.</td>
<td>Designing Tables (p. 90) details considerations for applying compression to the data in table columns and choosing distribution and sort keys.</td>
</tr>
<tr>
<td>Load data.</td>
<td>Loading Data (p. 117) explains the procedures for loading large datasets from Amazon DynamoDB tables or from flat files stored in Amazon S3 buckets. Best practices for loading data (p. 29) provides for tips for loading your data quickly and effectively.</td>
</tr>
<tr>
<td>Manage users, groups, and database security.</td>
<td>Managing Database Security (p. 83) covers database security topics.</td>
</tr>
</tbody>
</table>
If you want to ... | We recommend
---|---
Monitor and optimize system performance. | The System Tables Reference (p. 582) details system tables and views that you can query for the status of the database and monitor queries and processes. 
You should also consult the Amazon Redshift Management Guide to learn how to use the AWS Management Console to check the system health, monitor metrics, and back up and restore clusters.

Analyze and report information from very large datasets. | Many popular software vendors are certifying Amazon Redshift with their offerings to enable you to continue to use the tools you use today. For more information, see the Amazon Redshift partner page.

The SQL Reference (p. 205) has all the details for the SQL expressions, commands, and functions Amazon Redshift supports.

---

**Prerequisites**

Before you use this guide, you should complete these tasks.

- Install a SQL client.
- Launch an Amazon Redshift cluster.
- Connect your SQL client to the cluster database.

For step-by-step instructions, see the Amazon Redshift Getting Started Guide.

You should also know how to use your SQL client and should have a fundamental understanding of the SQL language.
An Amazon Redshift data warehouse is an enterprise-class relational database query and management system.

Amazon Redshift supports client connections with many types of applications, including business intelligence (BI), reporting, data, and analytics tools.

When you execute analytic queries, you are retrieving, comparing, and evaluating large amounts of data in multiple-stage operations to produce a final result.

Amazon Redshift achieves efficient storage and optimum query performance through a combination of massively parallel processing, columnar data storage, and very efficient, targeted data compression encoding schemes. This section presents an introduction to the Amazon Redshift system architecture and discusses how you can design your database and write queries to leverage that architecture for maximum performance.

Data warehouse system architecture

This section introduces the elements of the Amazon Redshift data warehouse architecture as shown in the following figure.
Amazon Redshift integrates with various data loading and ETL (extract, transform, and load) tools and business intelligence (BI) reporting, data mining, and analytics tools. Amazon Redshift is based on industry-standard PostgreSQL, so most existing SQL client applications will work with only minimal changes. For information about important differences between Amazon Redshift SQL and PostgreSQL, see Amazon Redshift and PostgreSQL (p. 206).

Connections

Amazon Redshift communicates with client applications by using industry-standard PostgreSQL JDBC and ODBC drivers. For more information, see Amazon Redshift and PostgreSQL JDBC and ODBC (p. 207).

Clusters

The core infrastructure component of an Amazon Redshift data warehouse is a cluster.

A cluster is composed of one or more compute nodes. If a cluster is provisioned with two or more compute nodes, an additional leader node coordinates the compute nodes and handles external communication. Your client application interacts directly only with the leader node. The compute nodes are transparent to external applications.

Leader Node

The leader node manages communications with client programs and all communication with compute nodes. It parses and develops execution plans to carry out database operations, in particular, the series of steps necessary to obtain results for complex queries. Based on the execution plan, the leader node compiles code, distributes the compiled code to the compute nodes, and assigns a portion of the data to each compute node.

The leader node distributes SQL statements to the compute nodes only when a query references tables that are stored on the compute nodes. All other queries run exclusively on the leader node. Amazon Redshift is designed to implement certain SQL functions only on the leader node. A query that uses any of these functions will return an error if it references tables that reside on the compute nodes. For more information, see SQL functions supported on the leader node (p. 205).
Compute Nodes

The leader node compiles code for individual elements of the execution plan and assigns the code to individual compute nodes. The compute nodes execute the compiled code send intermediate results back to the leader node for final aggregation.

Each compute node has its own dedicated CPU, memory, and attached disk storage, which are determined by the node type. As your workload grows, you can increase the compute capacity and storage capacity of a cluster by increasing the number of nodes, upgrading the node type, or both.

Amazon Redshift provides two node types; dense storage nodes and dense compute nodes. Each node provides two storage choices. You can start with a single 160 GB node and scale up to multiple 16 TB nodes to support a petabyte of data or more.

For a more detailed explanation of data warehouse clusters and nodes, see Internal architecture and system operation (p. 9).

Node slices

A compute node is partitioned into slices; one slice for each core of the node’s multi-core processor. Each slice is allocated a portion of the node's memory and disk space, where it processes a portion of the workload assigned to the node. The leader node manages distributing data to the slices and apportions the workload for any queries or other database operations to the slices. The slices then work in parallel to complete the operation.

When you create a table, you can optionally specify one column as the distribution key. When the table is loaded with data, the rows are distributed to the node slices according to the distribution key that is defined for a table. Choosing a good distribution key enables Amazon Redshift to use parallel processing to load data and execute queries efficiently. For information about choosing a distribution key, see Choose the best distribution style (p. 27).

Internal Network

Amazon Redshift takes advantage of high-bandwidth connections, close proximity, and custom communication protocols to provide private, very high-speed network communication between the leader node and compute nodes. The compute nodes run on a separate, isolated network that client applications never access directly.

Databases

A cluster contains one or more databases. User data is stored on the compute nodes. Your SQL client communicates with the leader node, which in turn coordinates query execution with the compute nodes.

Amazon Redshift is a relational database management system (RDBMS), so it is compatible with other RDBMS applications. Although it provides the same functionality as a typical RDBMS, including online transaction processing (OLTP) functions such as inserting and deleting data, Amazon Redshift is optimized for high-performance analysis and reporting of very large datasets.

Amazon Redshift is based on PostgreSQL 8.0.2. Amazon Redshift and PostgreSQL have a number of very important differences that you need to take into account as you design and develop your data warehouse applications. For information about how Amazon Redshift SQL differs from PostgreSQL, see Amazon Redshift and PostgreSQL (p. 206).

Performance

Amazon Redshift achieves extremely fast query execution by employing these performance features:

- Massively parallel processing
Massively-parallel processing

Massively-parallel processing (MPP) enables fast execution of the most complex queries operating on large amounts of data. Multiple compute nodes handle all query processing leading up to final result aggregation, with each core of each node executing the same compiled query segments on portions of the entire data.

Amazon Redshift distributes the rows of a table to the compute nodes so that the data can be processed in parallel. By selecting an appropriate distribution key for each table, you can optimize the distribution of data to balance the workload and minimize movement of data from node to node. For more information, see Choose the best distribution style (p. 27).

Loading data from flat files takes advantage of parallel processing by spreading the workload across multiple nodes while simultaneously reading from multiple files. For more information about how to load data into tables, see Best practices for loading data (p. 29).

Columnar data storage

Columnar storage for database tables drastically reduces the overall disk I/O requirements and is an important factor in optimizing analytic query performance. Storing database table information in a columnar fashion reduces the number of disk I/O requests and reduces the amount of data you need to load from disk. Loading less data into memory enables Amazon Redshift to perform more in-memory processing when executing queries. See Columnar storage (p. 8) for a more detailed explanation.

When columns are sorted appropriately, the query processor is able to rapidly filter out a large subset of data blocks. For more information, see Choose the best sort key (p. 27).

Data compression

Data compression reduces storage requirements, thereby reducing disk I/O, which improves query performance. When you execute a query, the compressed data is read into memory, then uncompressed during query execution. Loading less data into memory enables Amazon Redshift to allocate more memory to analyzing the data. Because columnar storage stores similar data sequentially, Amazon Redshift is able to apply adaptive compression encodings specifically tied to columnar data types. The best way to enable data compression on table columns is by allowing Amazon Redshift to apply optimal compression encodings when you load the table with data. To learn more about using automatic data compression, see Loading tables with automatic compression (p. 143).

Query optimizer

The Amazon Redshift query execution engine incorporates a query optimizer that is MPP-aware and also takes advantage of the columnar-oriented data storage. The Amazon Redshift query optimizer implements significant enhancements and extensions for processing complex analytic queries that often include multi-table joins, subqueries, and aggregation. To learn more about optimizing queries, see Tuning Query Performance (p. 180).

Compiled code

The leader node distributes fully optimized compiled code across all of the nodes of a cluster. Compiling the query eliminates the overhead associated with an interpreter and therefore increases the execution speed, especially for complex queries. The compiled code is cached and shared across sessions on the same cluster, so subsequent executions of the same query will be faster, often even with different parameters.
The execution engine compiles different code for the JDBC connection protocol and for ODBC and psql (libq) connection protocols, so two clients using different protocols will each incur the first-time cost of compiling the code. Other clients that use the same protocol, however, will benefit from sharing the cached code.

**Columnar storage**

Columnar storage for database tables is an important factor in optimizing analytic query performance because it drastically reduces the overall disk I/O requirements and reduces the amount of data you need to load from disk.

The following series of illustrations describe how columnar data storage implements efficiencies and how that translates into efficiencies when retrieving data into memory.

This first illustration shows how records from database tables are typically stored into disk blocks by row.

<table>
<thead>
<tr>
<th>SSN</th>
<th>Name</th>
<th>Age</th>
<th>Addr</th>
<th>City</th>
<th>St</th>
</tr>
</thead>
<tbody>
<tr>
<td>101259797</td>
<td>SMITH</td>
<td>88</td>
<td>899 FIRST ST</td>
<td>JUNO</td>
<td>AL</td>
</tr>
<tr>
<td>892375862</td>
<td>CHIN</td>
<td>37</td>
<td>16137 MAIN ST</td>
<td>POMONA</td>
<td>CA</td>
</tr>
<tr>
<td>318370701</td>
<td>HANOU</td>
<td>12</td>
<td>42 JUNE ST</td>
<td>CHICAGO</td>
<td>IL</td>
</tr>
</tbody>
</table>

In a typical relational database table, each row contains field values for a single record. In row-wise database storage, data blocks store values sequentially for each consecutive column making up the entire row. If block size is smaller than the size of a record, storage for an entire record may take more than one block. If block size is larger than the size of a record, storage for an entire record may take less than one block, resulting in an inefficient use of disk space. In online transaction processing (OLTP) applications, most transactions involve frequently reading and writing all of the values for entire records, typically one record or a small number of records at a time. As a result, row-wise storage is optimal for OLTP databases.

The next illustration shows how with columnar storage, the values for each column are stored sequentially into disk blocks.

<table>
<thead>
<tr>
<th>SSN</th>
<th>Name</th>
<th>Age</th>
<th>Addr</th>
<th>City</th>
<th>St</th>
</tr>
</thead>
<tbody>
<tr>
<td>101259797</td>
<td>SMITH</td>
<td>88</td>
<td>899 FIRST ST</td>
<td>JUNO</td>
<td>AL</td>
</tr>
<tr>
<td>892375862</td>
<td>CHIN</td>
<td>37</td>
<td>16137 MAIN ST</td>
<td>POMONA</td>
<td>CA</td>
</tr>
<tr>
<td>318370701</td>
<td>HANOU</td>
<td>12</td>
<td>42 JUNE ST</td>
<td>CHICAGO</td>
<td>IL</td>
</tr>
</tbody>
</table>

Using columnar storage, each data block stores values of a single column for multiple rows. As records enter the system, Amazon Redshift transparently converts the data to columnar storage for each of the columns.

In this simplified example, using columnar storage, each data block holds column field values for as many as three times as many records as row-based storage. This means that reading the same number of column field values for the same number of records requires a third of the I/O operations compared to...
row-wise storage. In practice, using tables with very large numbers of columns and very large row counts, storage efficiency is even greater.

An added advantage is that, since each block holds the same type of data, block data can use a compression scheme selected specifically for the column data type, further reducing disk space and I/O. For more information about compression encodings based on data types, see Compression encodings (p. 91).

The savings in space for storing data on disk also carries over to retrieving and then storing that data in memory. Since many database operations only need to access or operate on one or a small number of columns at a time, you can save memory space by only retrieving blocks for columns you actually need for a query. Where OLTP transactions typically involve most or all of the columns in a row for a small number of records, data warehouse queries commonly read only a few columns for a very large number of rows. This means that reading the same number of column field values for the same number of rows requires a fraction of the I/O operations and uses a fraction of the memory that would be required for processing row-wise blocks. In practice, using tables with very large numbers of columns and very large row counts, the efficiency gains are proportionally greater. For example, suppose a table contains 100 columns. A query that uses five columns will only need to read about five percent of the data contained in the table. This savings is repeated for possibly billions or even trillions of records for large databases. In contrast, a row-wise database would read the blocks that contain the 95 unneeded columns as well.

Typical database block sizes range from 2 KB to 32 KB. Amazon Redshift uses a block size of 1 MB, which is more efficient and further reduces the number of I/O requests needed to perform any database loading or other operations that are part of query execution.

Internal architecture and system operation

Topics
- Query processing (p. 10)
- Client requests and execution (p. 10)
- Explain plan (p. 12)
- Query execution steps (p. 12)

The following diagram shows a high level view of internal components and functionality of the Amazon Redshift data warehouse.
Query processing

The execution engine controls how SQL queries or other database operations are planned and executed by a cluster, with results or status being sent back to the client. This section provides an overview of Amazon Redshift query processing. For more information, see Tuning Query Performance (p. 180).

Client requests and execution

Amazon Redshift routes a requested operation (a SQL query or some other database operation) through the parser and optimizer to develop a query execution plan to perform the operation.

This illustration provides a high-level view of the query planning and execution workflow.
Note
These steps assume table data has been loaded in the database, and statistics have already been collected by executing an ANALYZE command.

1. **Parser** When a new request arrives that includes a SELECT, UPDATE, INSERT, or DELETE statement, the leader node passes the request to the parser. The parser also processes statements that contain a SELECT clause, such as CREATE TABLE AS.

2. **Query tree** The parser produces an initial query tree that is a logical representation of the original query or statement. This is input to the Amazon Redshift optimizer, which does a logical transformation of the query performs physical planning that it will use to develop the query execution plan.

3. **Logical transformation** The optimizer performs a rewrite of the query that incorporates optimizations such as predicate pushing, correlated subquery decorrelation, join elimination, common subexpression optimization, and several other processes.

4. **Query plan** The final query tree is converted to a query plan. Creating a query plan involves determining which methods and processing steps to use, such as, hash join or merge join, aggregation planning, and join ordering.

   You can use the **EXPLAIN** (p. 340) command to view the query plan, or explain plan. The query plan is a fundamental tool for analyzing and tuning complex queries. For more information about how to use an explain plan to optimize your queries, see **Analyzing the query plan**.

5. **Execution engine** The execution engine assembles a sequence of steps, segments, and streams to execute the query plan supplied by the optimizer. It then generates and compiles C++ code to be executed by the compute nodes. Compiled code executes much faster than interpreted code and uses less compute capacity. When benchmarking your queries, you should always compare the times for the second execution of a query, because the first execution time includes the overhead of compiling the code. For more information, see **Benchmarking with compiled code** (p. 195).

6. **Compute nodes** The execution engine sends executable code, corresponding to a stream, to each of the compute nodes.
One key to Amazon Redshift's query execution performance is the fact that separate query processes in each of the node slices execute the compiled query code in parallel. In addition, Amazon Redshift takes advantage of optimized network communication, memory, and disk management to pass intermediate results from one query plan step to the next, which also helps to speed query execution.

**Explain plan**

The next example shows a SQL query and the final query plan (explain plan) that Amazon Redshift creates to show logical steps needed to perform the query. Reading the explain plan from the bottom up, you can see a breakdown of logical operations needed to perform the query as well as an indication of their relative cost and the amount of data that needs to be processed. By analyzing the query plan, you can often identify opportunities to improve query performance.

```sql
select eventname, sum(pricepaid) from sales, event
where sales.eventid = event.eventid
group by eventname
order by 2 desc;
```

<table>
<thead>
<tr>
<th>QUERY PLAN</th>
</tr>
</thead>
<tbody>
<tr>
<td>XN Merge (cost=1000451920505.33..1000451920506.77 rows=576 width=27)</td>
</tr>
<tr>
<td>Merge Key: sum(sales.pricepaid)</td>
</tr>
<tr>
<td>-&gt; XN Network (cost=1000451920505.33..1000451920506.77 rows=576 width=27)</td>
</tr>
<tr>
<td>Send to leader</td>
</tr>
<tr>
<td>-&gt; XN Sort (cost=1000451920505.33..1000451920506.77 rows=576 width=27)</td>
</tr>
<tr>
<td>Sort Key: sum(sales.pricepaid)</td>
</tr>
<tr>
<td>-&gt; XN HashAggregate (cost=451920477.48..451920478.92 rows=576 width=27)</td>
</tr>
<tr>
<td>-&gt; XN Hash Join DS_DIST_INNER (cost=47.08..451920458.65 rows=3766 width=27)</td>
</tr>
<tr>
<td>Inner Dist Key: sales.eventid</td>
</tr>
<tr>
<td>Hash Cond: (&quot;outer&quot;.eventid = &quot;inner&quot;.eventid)</td>
</tr>
<tr>
<td>-&gt; XN Seq Scan on event (cost=0.00..87.98 rows=8798 width=21)</td>
</tr>
<tr>
<td>-&gt; XN Hash (cost=37.66..37.66 rows=3766 width=14)</td>
</tr>
<tr>
<td>-&gt; XN Seq Scan on sales (cost=0.00..37.66 rows=3766 width=14)</td>
</tr>
</tbody>
</table>

For more information about how to read an explain plan, see Analyzing the Explain Plan (p. 184).

**Query execution steps**

As part of generating an execution plan, the query optimizer breaks down the plan into streams, segments, and steps in preparation for distributing the data and query workload to the compute nodes. You can view the explain plan, together with system views, to consider whether you can improve query performance by specifying column constraints, sort keys, and distribution keys as part of your table design. By examining the explain plan, you might also find that you can improve query performance by rewriting your query.

The following illustration uses the explain plan in the previous example to show how the query optimizer converts logical operations in the explain plan into streams, segments, and steps that Amazon Redshift uses to generate compiled code for the compute node slices.
**Step** Each step is an individual operation in the explain plan. Steps can be combined to allow compute nodes to perform a query, join, or other database operation.

**Segment** Some number of steps that can be done by a single process. A segment is also a single compilation unit executable by compute nodes. Each segment begins with a SCAN of table data and ends either with a materialization step or some other network activity.

**Stream** A collection of segments that always begins with a SCAN of some data set and ends with a materialization or blocking step. Materialization or blocking steps include HASH, AGG, SORT, and SAVE.

For more information about the operators used in the explain plan, see EXPLAIN operators (p. 187).

---

**Workload management**

Amazon Redshift workload management (WLM) enables users to flexibly manage priorities within workloads so that short, fast-running queries won’t get stuck in queues behind long-running queries.

Amazon Redshift WLM creates query queues at runtime according to service classes, which define the configuration parameters for various types of queues, including internal system queues and user-accessible queues. From a user perspective, a user-accessible service class and a queue are functionally equivalent. For consistency, this documentation uses the term queue to mean a user-accessible service class as well as a runtime queue.

When you run a query, WLM assigns the query to a queue according to the user's user group or by matching a query group that is listed in the queue configuration with a query group label that the user sets at runtime.
By default, Amazon Redshift configures one queue with a concurrency level of five, which enables up to five queries to run concurrently, plus one predefined Superuser queue, with a concurrency level of one. You can define up to eight queues. Each queue can be configured with a maximum concurrency level of 50. The maximum total concurrency level for all user-defined queues (not including the Superuser queue) is 50.

The easiest way to modify the WLM configuration is by using the Amazon Redshift Management Console. You can also use the Amazon Redshift command line interface (CLI) or the Amazon Redshift API.

For more information about implementing and using workload management, see Implementing workload management (p. 196).

Using Amazon Redshift with other services

Amazon Redshift integrates with other AWS services to enable you to move, transform, and load your data quickly and reliably, using data security features.

Moving data between Amazon Redshift and Amazon S3

Amazon Simple Storage Service (Amazon S3) is a web service that stores data in the cloud. Amazon Redshift leverages parallel processing to read and load data from multiple data files stored in Amazon S3 buckets. For more information, see Loading data from Amazon S3 (p. 119).

You can also use parallel processing to export data from your Amazon Redshift data warehouse to multiple data files on Amazon S3. For more information, see Unloading Data (p. 173).

Using Amazon Redshift with Amazon DynamoDB

Amazon DynamoDB is a fully managed NoSQL database service. You can use the COPY command to load an Amazon Redshift table with data from a single Amazon DynamoDB table. For more information, see Loading data from an Amazon DynamoDB table (p. 140).

Importing data from remote hosts over SSH

You can use the COPY command in Amazon Redshift to load data from one or more remote hosts, such as Amazon EMR clusters, Amazon EC2 instances, or other computers. COPY connects to the remote hosts using SSH and executes commands on the remote hosts to generate data. Amazon Redshift supports multiple simultaneous connections. The COPY command reads and loads the output from multiple host sources in parallel. For more information, see Loading data from remote hosts (p. 134).

Automating data loads using AWS Data Pipeline

You can use AWS Data Pipeline to automate data movement and transformation into and out of Amazon Redshift. By using the built-in scheduling capabilities of AWS Data Pipeline, you can schedule and execute recurring jobs without having to write your own complex data transfer or transformation logic. For example, you can set up a recurring job to automatically copy data from Amazon DynamoDB into Amazon Redshift. For a tutorial that walks you through the process of creating a pipeline that periodically moves data from Amazon S3 to Amazon Redshift, see Copy Data to Amazon Redshift Using AWS Data Pipeline in the AWS Data Pipeline Developer Guide.
Getting Started Using Databases

Topics

• Step 1: Create a database (p. 15)
• Step 2: Create a database user (p. 16)
• Step 3: Create a database table (p. 16)
• Step 4: Load sample data (p. 18)
• Step 5: Query the system tables (p. 20)
• Step 6: Cancel a query (p. 23)
• Step 7: Clean up your resources (p. 25)

This section describes the basic steps to begin using the Amazon Redshift database.

The examples in this section assume you have signed up for the Amazon Redshift data warehouse service, created a cluster, and established a connection to the cluster from your SQL query tool. For information about these tasks, see the Amazon Redshift Getting Started Guide.

Important
The cluster that you deployed for this exercise will be running in a live environment. As long as it is running, it will accrue charges to your AWS account. The on-demand rate is $0.25 per hour for using the cluster; for more pricing information, go to the Amazon Redshift pricing page. To avoid unnecessary charges, you should delete your cluster when you are done with it. The final step of the exercise explains how to do so.

Step 1: Create a database

After you have verified that your cluster is up and running, you can create your first database. This database is where you will actually create tables, load data, and run queries. A single cluster can host multiple databases. For example, you can have a TICKIT database and an ORDERS database on the same cluster.

After you connect to the initial cluster database, the database you created when you launched the cluster, you use the initial database as the base for creating a new database.

For example, to create a database named tickit, issue the following command:
create database tickit;

For this exercise, we'll accept the defaults. For information about more command options, see CREATE DATABASE (p. 308) in the SQL Command Reference.

After you have created the TICKIT database, connect to the new database from your SQL client. The process for connecting to the database depends on which SQL client you are using. For information, see Connecting to a Cluster and the documentation for your client.

If you prefer not to connect to the TICKIT database, you can try the rest of the examples in this section using the default database.

**Step 2: Create a database user**

By default, only the master user that you created when you launched the cluster has access to the initial database in the cluster. To grant other users access, you must create one or more user accounts. Database user accounts are global across all the databases in a cluster; they do not belong to individual databases.

Use the CREATE USER command to create a new database user. When you create a new user, you specify the name of the new user and a password. A password is required. It must have between 8 and 64 characters, and it must include at least one uppercase letter, one lowercase letter, and one numeral.

For example, to create a user named GUEST with password ABCd4321, issue the following command:

cREATE USER guest WITH PASSWORD 'ABCd4321';

For information about other command options, see CREATE USER (p. 325) in the SQL Command Reference.

**Delete a database user**

You won't need the GUEST user account for this tutorial, so you can delete it. If you delete a database user account, the user will no longer be able to access any of the cluster databases.

Issue the following command to drop the GUEST user:

DROP USER guest;

The master user you created when you launched your cluster continues to have access to the database.

**Important**

Amazon Redshift strongly recommends that you do not delete the master user.

For information about command options, see DROP USER (p. 336) in the SQL Reference.

**Step 3: Create a database table**

After you create your new database, you create tables to hold your database data. You specify any column information for the table when you create the table.

For example, to create a table named testtable with a single column named testcol for an integer data type, issue the following command:

```sql
drop table testtable;
```

For information about more command options, see DROP TABLE (p. 313) in the SQL Command Reference.
create table testtable (testcol int);

The PG_TABLE_DEF system table contains information about all the tables in the cluster. To verify the result, issue the following SELECT command to query the PG_TABLE_DEF system table.

```sql
select * from pg_table_def where tablename = 'testtable';
```

The query result should look something like this:

<table>
<thead>
<tr>
<th>schemaname</th>
<th>tablename</th>
<th>column</th>
<th>type</th>
<th>encoding</th>
<th>distkey</th>
<th>sortkey</th>
<th>notnull</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>testtable</td>
<td>testcol</td>
<td>integer</td>
<td>none</td>
<td>f</td>
<td>0</td>
<td>f</td>
</tr>
</tbody>
</table>

(1 row)

By default, new database objects, such as tables, are created in a schema named "public". For more information about schemas, see Schemas (p. 86) in the Managing Database Security section.

The encoding, distkey, and sortkey columns are used by Amazon Redshift for parallel processing. For more information about designing tables that incorporate these elements, see Best practices for designing tables (p. 26).

**Insert data rows into a table**

After you create a table, you can insert rows of data into that table.

**Note**
The `INSERT` (p. 349) command inserts individual rows into a database table. For standard bulk loads, use the `COPY` (p. 276) command. For more information, see Use a COPY command to load data (p. 29).

For example, to insert a value of 100 into the `testtable` table (which contains a single column), issue the following command:

```sql
insert into testtable values (100);
```

**Select data from a table**

After you create a table and populate it with data, use a SELECT statement to display the data contained in the table. The SELECT * statement returns all the column names and row values for all of the data in a table and is a good way to verify that recently added data was correctly inserted into the table.

To view the data that you entered in the `testtable` table, issue the following command:

```sql
select * from testtable;
```

The result will look like this:

```
testcol
--------
100
(1 row)
```
For more information about using the SELECT statement to query tables, see SELECT (p. 359) in the SQL Command Reference.

### Step 4: Load sample data

Most of the examples in this guide use the TICKIT sample database. If you want to follow the examples using your SQL query tool, you will need to load the sample data for the TICKIT database.

The sample data for these examples is provided in the Amazon S3 bucket `awssampled`. Any valid AWS account has READ access to the data files.

**Note**

If you followed the steps in the Amazon Redshift Getting Started Guide, these tables already exist.

To load the sample data for the TICKIT database, you will first create the tables, then use the COPY command to load the tables with sample data that is stored in an Amazon S3 bucket. For more information, see Loading data from Amazon S3 (p. 119).

You create tables using the CREATE TABLE command with a list of columns paired with datatypes. Many of the create table statements in this example specify options for the column in addition to the data type, such as not null, distkey, and sortkey. These are column attributes related to optimizing your tables for query performance. You can visit Designing Tables (p. 90) to learn about how to choose these options when you design your table structures.

1. Create the tables for the database.

   The following SQL creates these tables: USERS, VENUE, CATEGORY, DATE, EVENT, LISTING, and SALES.

```sql
create table users(
    userid integer not null distkey sortkey,
    username char(8),
    firstname varchar(30),
    lastname varchar(30),
    city varchar(30),
    state char(2),
    email varchar(100),
    phone char(14),
    likesports boolean,
    liketheatre boolean,
    likeconcerts boolean,
    likejazz boolean,
    likeclassical boolean,
    likeopera boolean,
    likerock boolean,
    likevegas boolean,
    likebroadway boolean,
    likemusicals boolean);

create table venue(
    venueid smallint not null distkey sortkey,
    venuename varchar(100),
    venuecity varchar(30),
    venuestate char(2),
    venueaddress varchar(100));
```

API Version 2012-12-01

18
create table category(
catid smallint not null distkey sortkey,
catgroup varchar(10),
catname varchar(10),
catdesc varchar(50));

create table date(
dateid smallint not null distkey sortkey,
caldate date not null,
day character(3) not null,
week smallint not null,
month character(5) not null,
qtr character(5) not null,
year smallint not null,
holiday boolean default('N'));

create table event(
eventid integer not null distkey,
venueid smallint not null,
catid smallint not null,
dateid smallint not null sortkey,
eventname varchar(200),
starttime timestamp);

create table listing(
listid integer not null distkey,
sellerid integer not null,
eventid integer not null,
dateid smallint not null sortkey,
umtickets smallint not null,
priceperticket decimal(8,2),
totalprice decimal(8,2),
listtime timestamp);

create table sales(
salesid integer not null,
listid integer not null distkey,
sellerid integer not null,
buyerid integer not null,
eventid integer not null,
dateid smallint not null sortkey,
qtysold smallint not null,
pricepaid decimal(8,2),
commission decimal(8,2),
saletime timestamp);

2. Load the tables with data.

In this step, you will use the COPY command to load the tables using data from an Amazon S3 bucket. The Amazon S3 bucket awssampled0 contains the sample data for your use in these examples. The bucket has public READ permission, so any valid AWS account has access to the data. To use these COPY commands, replace <your-access-key-id> and <your-secret-access-key> with valid AWS account credentials.

Important
This example uses an Amazon S3 bucket that is located in the US East (Northern Virginia) region. When you load data using a COPY command, the bucket containing your data must
be in the same region as your cluster. If your cluster is in a region other than US East, you will need to use a different bucket. For information about how to load sample data from another region, see Create Tables, Upload Data, and Try Example Queries in the Amazon Redshift Getting Started Guide.

copy users from 's3://awssampledb/ticket/allusers_pipe.txt'
credentials 'aws_access_key_id=<access-key-id>;aws_secret_access_key=<secret-access-key>'
delimiter '|';
copy venue from 's3://awssampledb/ticket/venue_pipe.txt'
credentials 'aws_access_key_id=<access-key-id>;aws_secret_access_key=<secret-access-key>'
delimiter '|';
copy category from 's3://awssampledb/ticket/category_pipe.txt'
credentials 'aws_access_key_id=<access-key-id>;aws_secret_access_key=<secret-access-key>'
delimiter '|';
copy date from 's3://awssampledb/ticket/date2008_pipe.txt'
credentials 'aws_access_key_id=<access-key-id>;aws_secret_access_key=<secret-access-key>'
delimiter '|';
copy event from 's3://awssampledb/ticket/allevents_pipe.txt'
credentials 'aws_access_key_id=<access-key-id>;aws_secret_access_key=<secret-access-key>'
delimiter '|' timeformat 'YYYY-MM-DD HH:MI:SS';
copy listing from 's3://awssampledb/ticket/listings_pipe.txt'
credentials 'aws_access_key_id=<access-key-id>;aws_secret_access_key=<secret-access-key>'
delimiter '|';
copy sales from 's3://awssampledb/ticket/sales_tab.txt'
credentials 'aws_access_key_id=<access-key-id>;aws_secret_access_key=<secret-access-key>'
delimiter '\t' timeformat 'MM/DD/YYYY HH:MI:SS';

3. Verify the load results.

Use the following SELECT statements to verify that the tables were created and loaded with data. The select count(*) statement returns the number of rows in the table.

<table>
<thead>
<tr>
<th>Query</th>
</tr>
</thead>
<tbody>
<tr>
<td>select count(*) from users;</td>
</tr>
<tr>
<td>select count(*) from venue;</td>
</tr>
<tr>
<td>select count(*) from category;</td>
</tr>
<tr>
<td>select count(*) from date;</td>
</tr>
<tr>
<td>select count(*) from event;</td>
</tr>
<tr>
<td>select count(*) from listing;</td>
</tr>
<tr>
<td>select count(*) from sales;</td>
</tr>
</tbody>
</table>

Step 5: Query the system tables

In addition to the tables that you create, your database contains a number of system tables. These system tables contain information about your installation and about the various queries and processes that are running on the system. You can query these system tables to collect information about your database.
Note
The description for each table in the System Tables Reference indicates whether a table is superuser visible or user visible. You must be logged in as a superuser to query tables that are superuser visible.

Amazon Redshift provides access to the following types of system tables:

- **STL tables for logging (p. 584)**
  These system tables are generated from Amazon Redshift log files to provide a history of the system. Logging tables have an STL prefix.
- **STV tables for snapshot data (p. 649)**
  These tables are virtual system tables that contain snapshots of the current system data. Snapshot tables have an STV prefix.
- **System views (p. 671)**
  System views contain a subset of data found in several of the STL and STV system tables. Systems views have an SVV or SVL prefix.
- **System catalog tables (p. 687)**
  The system catalog tables store schema metadata, such as information about tables and columns. System catalog tables have a PG prefix.

You may need to specify the process ID associated with a query to retrieve system table information about that query. For information, see Determine the process ID of a running query (p. 22).

**View a list of table names**

For example, to view a list of all tables in the public schema, you can query the PG_TABLE_DEF system catalog table.

```sql
select tablename from pg_table_def where schemaname = 'public';
```

The result will look something like this:

```
tablename
---------
category
date
event
listing
sales
testtable
users
venue
```

**View database users**

You can query the PG_USER catalog to view a list of all database users, along with the user ID (USESYSID) and user privileges.
select * from pg_user;

<table>
<thead>
<tr>
<th>usename</th>
<th>usesysid</th>
<th>usecreatedb</th>
<th>usesuper</th>
<th>usecatupd</th>
<th>passwd</th>
<th>valuntil</th>
</tr>
</thead>
<tbody>
<tr>
<td>-----------+----------+-------------+----------+-----------+---------+----------</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rdsdb</td>
<td>1</td>
<td>t</td>
<td>t</td>
<td>t</td>
<td>********</td>
<td></td>
</tr>
<tr>
<td>masteruser</td>
<td>100</td>
<td>t</td>
<td>t</td>
<td>f</td>
<td>********</td>
<td></td>
</tr>
<tr>
<td>dwuser</td>
<td>101</td>
<td>f</td>
<td>f</td>
<td>f</td>
<td>********</td>
<td></td>
</tr>
<tr>
<td>simpleuser</td>
<td>102</td>
<td>f</td>
<td>f</td>
<td>f</td>
<td>********</td>
<td></td>
</tr>
<tr>
<td>poweruser</td>
<td>103</td>
<td>f</td>
<td>t</td>
<td>f</td>
<td>********</td>
<td></td>
</tr>
<tr>
<td>dbuser</td>
<td>104</td>
<td>t</td>
<td>f</td>
<td>f</td>
<td>********</td>
<td></td>
</tr>
</tbody>
</table>

(6 rows)

View recent queries

In the previous example, you found that the user ID (USESYSID) for masteruser is 100. To list the five most recent queries executed by masteruser, you can query the STL_QLOG view. The SVL_QLOG view is a friendlier subset of information from the STL_QUERY table. You can use this view to find the query ID (QUERY) or process ID (PID) for a recently run query or to see how long it took a query to complete. SVL_QLOG includes the first 60 characters of the query string (SUBSTRING) to help you locate a specific query. Use the LIMIT clause with your SELECT statement to limit the results to five rows.

select query, pid, elapsed, substring from svl_qlog
where userid = 100
order by starttime desc
limit 5;

The result will look something like this:

<table>
<thead>
<tr>
<th>query</th>
<th>pid</th>
<th>elapsed</th>
<th>substring</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>187752</td>
<td>18921</td>
<td>18465685</td>
</tr>
<tr>
<td></td>
<td>204168</td>
<td>5117</td>
<td>59603</td>
</tr>
<tr>
<td></td>
<td>187561</td>
<td>17046</td>
<td>1003052</td>
</tr>
<tr>
<td></td>
<td>187549</td>
<td>17046</td>
<td>1108584</td>
</tr>
<tr>
<td></td>
<td>187468</td>
<td>17046</td>
<td>5670661</td>
</tr>
</tbody>
</table>

(5 rows)

Determine the process ID of a running query

In the previous example you learned how to obtain the query ID and process ID (PID) for a completed query from the SVL_QLOG view.
You might need to find the PID for a query that is still running. For example, you will need the PID if you need to cancel a query that is taking too long to run. You can query the STV.RECENTS system table to obtain a list of process IDs for running queries, along with the corresponding query string. If your query returns multiple PIDs, you can look at the query text to determine which PID you need.

To determine the PID of a running query, issue the following SELECT statement:

```
select pid, user_name, starttime, query
from stv_recents
where status='Running';
```

### Step 6: Cancel a query

If a user issues a query that is taking too long too or is consuming excessive cluster resources, you might need to cancel the query. For example, a user might want to create a list of ticket sellers that includes the seller’s name and quantity of tickets sold. The following query selects data from the SALES table and joins the two tables by matching SELLERID and USERID in the WHERE clause.

```
select sellerid, firstname, lastname, sum(qtysold)
from sales, users
where sales.sellerid = users.userid
group by sellerid, firstname, lastname
order by 4 desc;
```

**Note**

This is a complex query. For this tutorial, you don’t need to worry about how this query is constructed.

The previous query runs in seconds and returns 2,102 rows.

Suppose the user forgets to put in the WHERE clause.

```
select sellerid, firstname, lastname, sum(qtysold)
from sales, users
group by sellerid, firstname, lastname
order by 4 desc;
```

The result set will include all of the rows in the SALES table multiplied by all the rows in the USERS table (49989*3766). This is called a Cartesian join, and it is not recommended. The result is over 188 million rows and takes a long time to run.

To cancel a running query, use the CANCEL command with the query's PID.

To find the process ID, query the STV.RECENTS table, as shown in the previous step. The following example shows how you can make the results more readable by using the TRIM function to trim trailing spaces and by showing only the first 20 characters of the query string.

```
select pid, trim(user_name), starttime, substring(query,1,20)
from stv_recents
where status='Running';
```

The result looks something like this:
To cancel the query with PID 18764, issue the following command:

```sql
cancel 18764;
```

**Note**
The CANCEL command will not abort a transaction. To abort or roll back a transaction, you must use the ABORT or ROLLBACK command. To cancel a query associated with a transaction, first cancel the query then abort the transaction.

If the query that you canceled is associated with a transaction, use the ABORT or ROLLBACK command to cancel the transaction and discard any changes made to the data:

```sql
abort;
```

Unless you are signed on as a superuser, you can cancel only your own queries. A superuser can cancel all queries.

### Cancel a query from another session

If your query tool does not support running queries concurrently, you will need to start another session to cancel the query. For example, SQLWorkbench, which is the query we use in the Amazon Redshift Getting Started Guide, does not support multiple concurrent queries. To start another session using SQLWorkbench, select File, New Window and connect using the same connection parameters. Then you can find the PID and cancel the query.

### Cancel a query using the Superuser queue

If your current session has too many queries running concurrently, you might not be able to run the CANCEL command until another query finishes. In that case, you will need to issue the CANCEL command using a different workload management query queue.

Workload management enables you to execute queries in different query queues so that you don’t need to wait for another query to complete. The workload manager creates a separate queue, called the Superuser queue, that you can use for troubleshooting. To use the Superuser queue, you must be logged on a superuser and set the query group to ‘superuser’ using the SET command. After running your commands, reset the query group using the RESET command.

To cancel a query using the Superuser queue, issue these commands:

```sql
set query_group to 'superuser';
cancel 18764;
reset query_group;
```

For information about managing query queues, see Implementing workload management (p. 196).
Step 7: Clean up your resources

If you deployed a cluster in order to complete this exercise, when you are finished with the exercise, you should delete the cluster so that it will stop accruing charges to your AWS account.

To delete the cluster, follow the steps in Deleting a Cluster in the Amazon Redshift Management Guide.

If you want to keep the cluster, you might want to keep the sample data for reference. Most of the examples in this guide use the tables you created in this exercise. The size of the data will not have any significant effect on your available storage.

If you want to keep the cluster, but want to clean up the sample data, you can run the following command to drop the TICKIT database:

```sql
drop database tickit;
```

If you didn’t create a TICKIT database, or if you don’t want to drop the database, run the following commands to drop just the tables:

```sql
drop table testtable;
drop table users;
drop table venue;
drop table category;
drop table date;
drop table event;
drop table listing;
drop table sales;
```
Amazon Redshift Best Practices

Topics
- Amazon Redshift best practices for designing tables (p. 26)
- Amazon Redshift best practices for loading data (p. 29)
- Amazon Redshift best practices for tuning query performance (p. 32)

This chapter presents best practices for designing tables, loading data into tables, and tuning query performance.

Amazon Redshift is not the same as other SQL database systems. To fully realize the benefits of the Amazon Redshift architecture, you must specifically design, build, and load your tables to leverage massively parallel processing, columnar data storage, and columnar data compression. If your data loading and query execution times are longer than you expect, or longer than you want, you might be overlooking key information.

If you are an experienced SQL database developer, we strongly recommend that you review this chapter before you begin developing your Amazon Redshift data warehouse.

If you are new to developing SQL databases, this is not the best place to start. We recommend that you begin by reading Getting Started Using Databases (p. 15) and trying the examples yourself.

This chapter provides an overview of the most important development principles, along with specific tips, examples, and best practices for implementing those principles. No single practice can apply to every application. You should evaluate all of your options before finalizing a database design. For more information, refer to Designing Tables (p. 90), Loading Data (p. 117), Tuning Query Performance (p. 180), and the reference chapters.

Amazon Redshift best practices for designing tables

Topics
- Take the Tuning Table Design tutorial (p. 27)
- Choose the best sort key (p. 27)
- Choose the best distribution style (p. 27)
- Let COPY choose compression encodings (p. 28)
As you plan your database, there are key table design decisions that will heavily influence overall query performance. These design choices also have a significant effect on storage requirements, which in turn affects query performance by reducing the number of I/O operations and minimizing the memory required to process queries.

This section summarizes the most important design decisions and presents best practices for optimizing query performance. Designing Tables (p. 90) provides more detailed explanations and examples of table design options.

**Take the Tuning Table Design tutorial**

Tutorial: Tuning Table Design (p. 35) walks you step-by-step through the process of choosing sort keys, distribution styles, and compression encodings, and shows you how to compare system performance before and after tuning.

### Choose the best sort key

Amazon Redshift stores your data on disk in sorted order according to the sort key. The Redshift query optimizer uses sort order when it determines optimal query plans.

- If recent data is queried most frequently, specify the timestamp column as the leading column for the sort key.

  Queries will be more efficient because they can skip entire blocks that fall outside the time range.

- If you do frequent range filtering or equality filtering on one column, specify that column as the sort key.

  Redshift can skip reading entire blocks of data for that column because it keeps track of the minimum and maximum column values stored on each block and can skip blocks that don't apply to the predicate range.

- If you frequently join a table, specify the join column as both the sort key and the distribution key.

  This enables the query optimizer to choose a sort merge join instead of a slower hash join. Because the data is already sorted on the join key, the query optimizer can bypass the sort phase of the sort merge join.

For more information about choosing and specifying sort keys, see Tutorial: Tuning Table Design (p. 35) and Choosing sort keys (p. 113).

### Choose the best distribution style

When you execute a query, the query optimizer redistributes the rows to the compute nodes as needed to perform any joins and aggregations. The goal in selecting a table distribution style is to minimize the impact of the redistribution step by locating the data where it needs to be before the query is executed.

1. **Distribute the fact table and one dimension table on their common columns.**

   Your fact table can have only one distribution key. Any tables that join on another key will not be colocated with the fact table. Choose one dimension to collocate based on how frequently it is joined.
and the size of the joining rows. Designate both the dimension table's primary key and the fact table's corresponding foreign key as the DISTKEY.

2. **Choose the largest dimension based on the size of the filtered data set.**

   Only the rows that are used in the join need to be distributed, so consider the size of the of the data set after filtering, not the size of the table.

3. **Change some dimension tables to use ALL distribution.**

   If a dimension table cannot be collocated with the fact table or other important joining tables, you can improve query performance significantly by distributing the entire table to all of the nodes. Using ALL distribution multiplies storage space requirements and increases load times and maintenance operations, so you should weigh all factors before choosing ALL distribution.

For more information about choosing distribution styles, see Tutorial: Tuning Table Design (p. 35) and Choosing a data distribution style (p. 101).

**Let COPY choose compression encodings**

You can specify compression encodings when you create a table, but most cases, automatic compression produces the best results.

The COPY command will analyze your data and apply compression encodings to an empty table automatically as part of the load operation. You can turn off automatic compression, but as a rule it's not necessary. COPY disables automatic compression by default if the table already has compression encodings or if it is not empty. For more information, see Tutorial: Tuning Table Design (p. 35) and Loading tables with automatic compression (p. 143).

**Define primary key and foreign key constraints**

Define primary key and foreign key constraints between tables wherever appropriate. Even though they are informational only, the query optimizer uses those constraints to generate more efficient query plans.

Do not define primary key and foreign key constraints unless your application enforces the constraints. Amazon Redshift does not enforce unique, primary-key, and foreign-key constraints.

See Defining constraints (p. 113) for additional information about how Amazon Redshift uses constraints.

**Use the smallest possible column size**

Don’t make it a practice to use the maximum column size for convenience.

Instead, consider the largest values you are likely to store in a VARCHAR column, for example, and size your columns accordingly. Because Amazon Redshift compresses column data very effectively, creating columns much larger than necessary has minimal impact on the size of data tables. During processing for complex queries, however, intermediate query results might need to be stored in temporary tables. Because temporary tables are not compressed, unnecessarily large columns consume excessive memory and temporary disk space, which can affect query performance.

**Use date/time data types for date columns**

Amazon Redshift stores DATE and TIMESTAMP data more efficiently than CHAR or VARCHAR, which results in better query performance. Use the DATE or TIMESTAMP data type, depending on the resolution you need, rather than a character type when storing date/time information. For more information, see Datetime types (p. 226).
Amazon Redshift best practices for loading data

Topics
- Take the Loading Data tutorial (p. 29)
- Take the Tuning Table Design tutorial (p. 29)
- Use a COPY command to load data (p. 29)
- Use a single COPY command to load from multiple files (p. 30)
- Split your load data into multiple files (p. 30)
- Compress your data files with gzip or lzop (p. 30)
- Use a manifest file (p. 30)
- Verify data files before and after a load (p. 30)
- Use a multi-row insert (p. 31)
- Use a bulk insert (p. 31)
- Load data in sort key order (p. 31)
- Load data in sequential blocks (p. 31)
- Use time-series tables (p. 32)
- Use a staging table to perform a merge (upsert) (p. 32)
- Schedule around maintenance windows (p. 32)

Loading very large data sets can take a long time and consume a lot of computing resources. How your data is loaded can also affect query performance. This section presents best practices for loading data efficiently using COPY commands, bulk inserts, and staging tables.

Take the Loading Data tutorial

Tutorial: Loading Data from Amazon S3 (p. 62) walks you beginning to end through the steps to upload data to an Amazon S3 bucket and then use the COPY command to load the data into your tables. The tutorial includes help with troubleshooting load errors and compares the performance difference between loading from a single file and loading from multiple files.

Take the Tuning Table Design tutorial

Data loads are heavily influenced by table design, especially compression encodings and distribution styles. Tutorial: Tuning Table Design (p. 35) walks you step-by-step through the process of choosing sort keys, distribution styles, and compression encodings, and shows you how to compare system performance before and after tuning.

Use a COPY command to load data

The COPY command loads data in parallel from Amazon S3, Amazon EMR, Amazon DynamoDB, or multiple data sources on remote hosts. COPY loads large amounts of data much more efficiently than using INSERT statements, and stores the data more effectively as well.

For more information about using the COPY command, see Loading data from Amazon S3 (p. 119) and Loading data from an Amazon DynamoDB table (p. 140).
Use a single COPY command to load from multiple files

Amazon Redshift automatically loads in parallel from multiple data files.

If you use multiple concurrent COPY commands to load one table from multiple files, Amazon Redshift is forced to perform a serialized load, which is much slower and requires a VACUUM at the end if the table has a sort column defined. For more information about using COPY to load data in parallel, see Loading data from Amazon S3 (p. 119).

Split your load data into multiple files

The COPY command loads the data in parallel from multiple files, dividing the workload among the nodes in your cluster. When you load all the data from a single large file, Amazon Redshift is forced to perform a serialized load, which is much slower. For more information about how to split your data into files and examples of using COPY to load data, see Loading data from Amazon S3 (p. 119).

Compress your data files with gzip or lzop

We strongly recommend that you individually compress your load files using gzip or lzop when you have large datasets.

Specify the GZIP or LZOP option with the COPY command. This example loads the TIME table from a pipe-delimited lzop file:

```
copy time
from 's3://mybucket/data/timerows.lzo'
credentials 'aws_access_key_id=<access-key-id>;aws_secret_access_key=<secret-access-key>'
lzop
delimiter '|';
```

Use a manifest file

Amazon S3 provides eventual consistency for some operations, so it is possible that new data will not be available immediately after the upload, which could result in an incomplete data load or loading stale data. You can manage data consistency by using a manifest file to load data and using a named endpoint for Amazon S3 buckets in the US Standard region. For more information, see Managing data consistency (p. 121).

Verify data files before and after a load

When you load data from Amazon S3, first upload your files to your Amazon S3 bucket, then verify that the bucket contains all the correct files, and only those files. For more information, see Verifying that the correct files are present in your bucket (p. 123).

After the load operation is complete, query the STL_LOAD_COMMITS (p. 606) system table to verify that the expected files were loaded. For more information, see Verifying that the data was loaded correctly (p. 142).
Use a multi-row insert

If a COPY command is not an option and you require SQL inserts, use a multi-row insert whenever possible. Data compression is inefficient when you add data only one row or a few rows at a time.

Multi-row inserts improve performance by batching up a series of inserts. The following example inserts three rows into a four-column table using a single INSERT statement. This is still a small insert, shown simply to illustrate the syntax of a multi-row insert.

```sql
insert into category_stage values
(default, default, default, default),
(20, default, 'Country', default),
(21, 'Concerts', 'Rock', default);
```

See INSERT (p. 349) for more details and examples.

Use a bulk insert

Use a bulk insert operation with a SELECT clause for high performance data insertion.

Use the INSERT (p. 349) and CREATE TABLE AS (p. 320) commands when you need to move data or a subset of data from one table into another.

For example, the following INSERT statement selects all of the rows from the CATEGORY table and inserts them into the CATEGORY_STAGE table.

```sql
insert into category_stage
(select * from category);
```

The following example creates CATEGORY_STAGE as a copy of CATEGORY and inserts all of the rows in CATEGORY into CATEGORY_STAGE.

```sql
create table category_stage as
select * from category;
```

Load data in sort key order

Load your data in sort key order to avoid needing to vacuum.

As long as each batch of new data follows the existing rows in your table, your data will be properly stored in sort order, and you will not need to run a vacuum. You don’t need to presort the rows in each load because COPY sorts each batch of incoming data as it loads.

For example, suppose you load data every day based on the current day's activity. If your sort key is a timestamp column, your data is stored in sort order because the current day's data is always appended at the end of the previous day's data.

Load data in sequential blocks

If you need to add a large quantity of data, load the data in sequential blocks according to sort order to eliminate the need to vacuum.
For example, suppose you need to load a table with events from January, 2012 to December, 2012. Load
the rows for January, then February, and so on. Your table will be completely sorted when your load
completes and you will not need to run a vacuum. For more information, see Use time-series tables (p.?).

When loading very large data sets, the space required to sort might exceed the total available space. By
loading data in smaller blocks, you’ll use much less intermediate sort space during each load. In addition,
loading smaller blocks will make it easier to restart if the COPY fails and is rolled back.

**Use time-series tables**

If your data has a fixed retention period, we strongly recommend that you organize your data as a sequence
of time-series tables, where each table is identical but contains data for different time ranges.

You can easily remove old data simply by executing a DROP TABLE on the corresponding tables, which
is much faster than running a large scale DELETE, and also saves you from having to run a subsequent
VACUUM to reclaim space. You can create a UNION ALL view to hide the fact that the data is stored in
different tables. When you delete old data, simply refine your UNION ALL view to remove the dropped
tables. Similarly, as you load new time periods into new tables, add the new tables to the view.

If you use time-series tables with a timestamp column for the sort key, you effectively load your data in
sort key order, which eliminates the need to vacuum to resort the data. For more information, see Load
data in sort key order (p. 31).

**Use a staging table to perform a merge (upsert)**

You can efficiently update and insert new data by loading your data into a staging table first.

You can efficiently update and insert new data by loading your data into a staging table first. While Amazon
Redshift does not support a single merge statement (update or insert, also known as an upsert) to insert
and update data from a single data source, you can effectively perform an merge operation by loading
your data into a staging table and then joining the staging table with your target table for an UPDATE
statement and an INSERT statement. For instructions, see Updating and inserting new data (p. 150).

**Schedule around maintenance windows**

If a scheduled maintenance occurs while a query is running, the query is terminated and rolled back and
you will need to restart it. Schedule long-running operations, such as large data loads or VACUUM
operation, to avoid maintenance windows. You can also minimize the risk, and make restarts easier when
they are needed, by performing data loads in smaller increments and managing the size of your VACUUM
operations. For more information, see Load data in sequential blocks (p. 31) and Vacuuming tables (p. 161).

**Amazon Redshift best practices for tuning query performance**

- Design for performance (p. 33)
- Take the Tuning Table Design tutorial (p. 33)
- Vacuum your database (p. 33)
- Avoid long vacuums by using a deep copy (p. 33)
- Increase the available memory (p. 33)
- Maintain up-to-date table statistics (p. 34)
• Specify redundant predicates on the sort column (p. 34)

If you designed and built your database according to the principles in the Designing Tables section of this guide, your queries will take full advantage of parallel processing and should run efficiently without significant tuning. If some of your queries are taking too long to run, follow the steps in this section to optimize query performance.

Design for performance

The most important factor in query performance is the design of your tables. When you create your tables, consider carefully your choice of columns, data types, sort keys, distribution key, compression encodings, and constraints. For step-by-step tutorial on table design, see Tutorial: Tuning Table Design (p. 35). Review Best practices for designing tables (p. 26) for information about some of the most important considerations.

Take the Tuning Table Design tutorial

Optimal query performance depends on optimal table design. Tutorial: Tuning Table Design (p. 35) walks you step-by-step through the process of choosing sort keys, distribution styles, and compression encodings, and shows you how to compare system performance before and after tuning.

Vacuum your database

Run the VACUUM command whenever you add, delete, or modify a large number of rows, unless you load your data in sort key order. The VACUUM command reorganizes your data to maintain the sort order and restore performance.

You can reduce or eliminate the need to vacuum by loading data in sort key order and by loading data in sequential blocks. For more information, see Load data in sort key order (p. 31) and Load data in sequential blocks (p. 31).

For more information about how often to vacuum, see Vacuuming tables (p. 161).

Avoid long vacuums by using a deep copy

A deep copy recreates and repopulates a table by using a bulk insert, which automatically resorts the table. If a table has a large unsorted region, a deep copy is much faster than a vacuum. The trade off is that you cannot make concurrent updates during a deep copy operation, which you can do during a vacuum. For more information, see Performing a deep copy (p. 156).

Increase the available memory

If your data loads or vacuums take too long, increase the memory available to a COPY or VACUUM by increasing wlm_query_slot_count.

Loading data and vacuuming can be memory intensive. You can allocate additional memory to COPY statements or VACUUM statements, and improve their performance, by increasing the value of the wlm_query_slot_count parameter for the duration of the operation. Note, however, that increasing wlm_query_slot_count will reduce the number of other concurrent queries that you can run on your cluster. See wlm_query_slot_count (p. 697) for instructions about how to set this parameter and more details about the concurrency implications.
Maintain up-to-date table statistics

To ensure that your table statistics are current, run the ANALYZE command whenever you’ve made a significant number of changes to your data.

The Amazon Redshift query optimizer relies on statistical metadata to determine optimal query plans. When you initially load your table using COPY commands (with STATUPDATE OFF), INSERT INTO SELECT commands, or CREATE TABLE AS commands, Amazon Redshift automatically generates statistics for the table at the end of the load. However, as you add, modify, or delete data, the table’s statistics may no longer reflect the characteristics for the current data.

See Analyzing tables (p. 158) for more information and examples.

Specify redundant predicates on the sort column

Use a predicate on the leading sort column of the fact table, or the largest table, in a join. Add predicates to filter other tables that participate in the join, even when the predicates are redundant.

In a star schema or similar design, where a large fact table is joined to multiple dimension tables, when you add a predicate in the WHERE clause to filter on the sort column of the largest table, you enable the query planner to skip scanning large numbers of disk blocks. Without the filter, the query execution engine must scan the entire table, which will degrade query performance over time as the table grows.

Even if a predicate is already being applied on a table in a join query but transitively applies to another table in the query that is sorted on the column in the predicate, you can improve performance by specifying the redundant predicate. That way, when scanning the other table, Amazon Redshift can efficiently skip blocks from that table as well.

For example, suppose you want to join TAB1 and TAB2. The sort key for TAB1 is `tab1.timestamp`, and the sort key for TAB2 is `tab2.timestamp`. The following query joins the tables on their common key and filters for `tab1.timestamp > '1/1/2013'`.

```
SELECT * FROM tab1, tab2
WHERE tab1.key = tab2.key
AND tab1.timestamp > '1/1/2013';
```

If the WHERE clause doesn’t include a predicate for `tab2.timestamp`, the execution engine is forced to scan the entire table. If the join would result in values from `tab2.timestamp2` also being greater than `1/1/2013`, then add that filter also, even though it is redundant.

```
SELECT * FROM tab1, tab2
WHERE tab1.key = tab2.key
AND tab1.timestamp > '1/1/2013'
AND tab2.timestamp > '1/1/2013';
```
**Tutorial: Tuning Table Design**

In this tutorial, you will learn how to optimize the design of your tables. You will start by creating tables based on the Star Schema Benchmark (SSB) schema without sort keys, distribution styles, and compression encodings. You will load the tables with test data and test system performance. Next, you will apply best practices to recreate the tables using sort keys and distribution styles. You will load the tables with test data using automatic compression and then you will test performance again so that you can compare the performance benefits of well-designed tables.

**Estimated time:** 60 minutes

**Estimated cost:** $1.00 per hour for the cluster

**Prerequisites**

You will need your AWS credentials (access key ID and secret access key) to load test data from Amazon S3. If you need to create new access keys, go to Administering Access Keys for IAM Users.

**Steps**

- Step 1: Create a test data set (p. 36)
- Step 2: Test system performance to establish a baseline (p. 40)
- Step 3: Select sort keys (p. 44)
- Step 4: Select distribution styles (p. 45)
- Step 5: Review compression encodings (p. 48)
- Step 6: Recreate the test data set (p. 51)
- Step 7: Retest system performance after tuning (p. 54)
- Step 8: Evaluate the results (p. 58)
- Step 9: Clean up your resources (p. 60)
- Summary (p. 60)
Step 1: Create a test data set

Data warehouse databases commonly use a star schema design, in which a central fact table contains the core data for the database and several dimension tables provide descriptive attribute information for the fact table. The fact table joins each dimension table on a foreign key that matches the dimension’s primary key.

Star Schema Benchmark (SSB)

For this tutorial, you will use a set of five tables based on the Star Schema Benchmark (SSB) schema. The following diagram shows the SSB data model.

To create a test data set

You will create a set of tables without sort keys, distribution styles, or compression encodings. Then, you will load the tables with data from the SSB data set.

1. (Optional) Launch a cluster.

   If you already have a cluster that you want to use, you can skip this step. Your cluster should have at least two nodes. For the exercises in this tutorial, you will use a four-node cluster.

   To launch a dw2.large cluster with four nodes, follow the steps in Getting Started with Amazon Redshift, but select Multi Node for Cluster Type and set Number of Compute Nodes to 4.

   Follow the steps to connect to your cluster from a SQL client and test a connection. You do not need to complete the remaining steps to create tables, upload data, and try example queries.
2. Create the SSB test tables using minimum attributes.

**Note**
If the SSB tables already exist in the current database, you will need to drop the tables first. See Step 6: Recreate the test data set (p. 51) for the DROP TABLE commands.

For the purposes of this tutorial, the first time you create the tables, they will not have sort keys, distribution styles, or compression encodings.

Execute the following CREATE TABLE commands.

```
CREATE TABLE part
(
    p_partkey     INTEGER NOT NULL,
    p_name        VARCHAR(22) NOT NULL,
    p_mfgr        VARCHAR(6) NOT NULL,
    p_category    VARCHAR(7) NOT NULL,
    p_brand1      VARCHAR(9) NOT NULL,
    p_color       VARCHAR(11) NOT NULL,
    p_type        VARCHAR(25) NOT NULL,
    p_size        INTEGER NOT NULL,
    p_container   VARCHAR(10) NOT NULL
);

CREATE TABLE supplier
(
    s_suppkey   INTEGER NOT NULL,
    s_name      VARCHAR(25) NOT NULL,
    s_address   VARCHAR(25) NOT NULL,
    s_city      VARCHAR(10) NOT NULL,
    s_nation    VARCHAR(15) NOT NULL,
    s_region    VARCHAR(12) NOT NULL,
    s_phone     VARCHAR(15) NOT NULL
);
```
CREATE TABLE customer
(
  c_custkey      INTEGER NOT NULL,
  c_name         VARCHAR(25) NOT NULL,
  c_address      VARCHAR(25) NOT NULL,
  c_city         VARCHAR(10) NOT NULL,
  c_nation       VARCHAR(15) NOT NULL,
  c_region       VARCHAR(12) NOT NULL,
  c_phone        VARCHAR(15) NOT NULL,
  c_mktsegment   VARCHAR(10) NOT NULL
);

CREATE TABLE dwdate
(
  d_datekey            INTEGER NOT NULL,
  d_date               VARCHAR(19) NOT NULL,
  d_dayofweek          VARCHAR(10) NOT NULL,
  d_month              VARCHAR(10) NOT NULL,
  d_year               INTEGER NOT NULL,
  d_yearmonthnum       INTEGER NOT NULL,
  d_yearmonth          VARCHAR(8) NOT NULL,
  d_daynuminweek       INTEGER NOT NULL,
  d_daynuminmonth      INTEGER NOT NULL,
  d_daynuminyear       INTEGER NOT NULL,
  d_monthnuminyear     INTEGER NOT NULL,
  d_weeknuminyear      INTEGER NOT NULL,
  d_sellingseason      VARCHAR(13) NOT NULL,
  d_lastdayinweekfl    VARCHAR(1) NOT NULL,
  d_lastdayinmonthfl   VARCHAR(1) NOT NULL,
  d_holidayfl          VARCHAR(1) NOT NULL,
  d_weekdayfl          VARCHAR(1) NOT NULL
);

CREATE TABLE lineorder
(
  lo_orderkey          INTEGER NOT NULL,
  lo_linenumber        INTEGER NOT NULL,
  lo_custkey           INTEGER NOT NULL,
  lo_partkey           INTEGER NOT NULL,
  lo_suppkey           INTEGER NOT NULL,
  lo_orderdate         INTEGER NOT NULL,
  lo_orderpriority     VARCHAR(15) NOT NULL,
  lo_shippriority      VARCHAR(1) NOT NULL,
  lo_quantity          INTEGER NOT NULL,
  lo_extendedprice     INTEGER NOT NULL,
  lo_ordertotalprice   INTEGER NOT NULL,
  lo_discount          INTEGER NOT NULL,
  lo_revenue           INTEGER NOT NULL,
  lo_supplycost        INTEGER NOT NULL,
  lo_tax               INTEGER NOT NULL,
  lo_commitdate        INTEGER NOT NULL,
  lo_shipmode          VARCHAR(10) NOT NULL
);

3. Load the tables using SSB sample data.

The sample data for this tutorial is provided in Amazon S3 buckets that give read access to all
authenticated AWS users, so any valid AWS credentials will work. The bucket that contains the load
data must be in the same region as your cluster.
a. Use the following table to find the correct bucket name for your region.

<table>
<thead>
<tr>
<th>Region</th>
<th>&lt;region-specific-bucket-name&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>US East (Northern Virginia)</td>
<td>awssampledb</td>
</tr>
<tr>
<td>US West (Oregon)</td>
<td>awssampledbuswest2</td>
</tr>
<tr>
<td>EU (Ireland)</td>
<td>awssampledbeuwest1</td>
</tr>
<tr>
<td>Asia Pacific (Singapore)</td>
<td>awssampledbapsoutheast1</td>
</tr>
<tr>
<td>Asia Pacific (Sydney)</td>
<td>awssampledbapsoutheast2</td>
</tr>
<tr>
<td>Asia Pacific (Tokyo)</td>
<td>awssampledapnortheast1</td>
</tr>
</tbody>
</table>

b. Create a new text file named loadssb.sql containing the following SQL.

```sql
copy customer from 's3://<region-specific-bucket-name>/ssbgz/customer'
credentials 'aws_access_key_id=<Your-Access-Key-ID>;aws_secret_access_key=<Your-Secret-Access-Key>' gzip compupdate off;

copy dwdate from 's3://<region-specific-bucket-name>/ssbgz/dwdate'
credentials 'aws_access_key_id=<Your-Access-Key-ID>;aws_secret_access_key=<Your-Secret-Access-Key>' gzip compupdate off;

copy lineorder from 's3://<region-specific-bucket-name>/ssbgz/lineorder'
credentials 'aws_access_key_id=<Your-Access-Key-ID>;aws_secret_access_key=<Your-Secret-Access-Key>' gzip compupdate off;

copy part from 's3://<region-specific-bucket-name>/ssbgz/part'
credentials 'aws_access_key_id=<Your-Access-Key-ID>;aws_secret_access_key=<Your-Secret-Access-Key>' gzip compupdate off;

copy supplier from 's3://<region-specific-bucket-name>/ssbgz/supplier'
credentials 'aws_access_key_id=<Your-Access-Key-ID>;aws_secret_access_key=<Your-Secret-Access-Key>' gzip compupdate off;
```

c. Replace `<region-specific-bucket-name>` with the name of a bucket in the same region as your cluster.

d. Replace `<Your-Access-Key-ID>` and `<Your-Secret-Access-Key>` with your own AWS account credentials. The segment of the credentials string that is enclosed in single quotes must not contain any spaces or line breaks.

e. Execute the COPY commands either by running the SQL script or by copying and pasting the commands into your SQL client.

Note
The load operation will take about 10 to 15 minutes for all five tables. This might be a good time to get a cup of tea or catch up on email.

Your results should look similar to the following.

```
Load into table 'customer' completed, 3000000 record(s) loaded successfully.
```
4. Sum the execution time for all five tables, or else note the total script execution time. You’ll record that number as Load Time in the benchmarks table in Step 2.

5. To verify that each table loaded correctly, execute the following commands.

   ```sql
   select count(*) from LINEORDER;
   select count(*) from PART;
   select count(*) from CUSTOMER;
   select count(*) from SUPPLIER;
   select count(*) from DWDATE;
   ```

   The following results table shows the number of rows for each SSB table.

<table>
<thead>
<tr>
<th>Table name</th>
<th>Rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>LINEORDER</td>
<td>600,037,902</td>
</tr>
<tr>
<td>PART</td>
<td>1,400,000</td>
</tr>
<tr>
<td>CUSTOMER</td>
<td>3,000,000</td>
</tr>
<tr>
<td>SUPPLIER</td>
<td>1,000,000</td>
</tr>
<tr>
<td>DWDATE</td>
<td>2,556</td>
</tr>
</tbody>
</table>

**Next step**

Step 2: Test system performance to establish a baseline (p. 40)

**Step 2: Test system performance to establish a baseline**

As you test system performance before and after tuning your tables, you will record the following details:

- Load time
- Storage use
- Query performance
The examples in this tutorial are based on using a four-node dw2.large cluster. Your results will be different, even if you use the same cluster configuration. System performance is influenced by many factors, and no two systems will perform exactly the same.

You will record your results using the following benchmarks table.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load time (five tables)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Storage Use</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LINEORDER</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PART</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CUSTOMER</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DWDATE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUPPLIER</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total storage</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Query execution time</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Query 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Query 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Query 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total execution time</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

To test system performance to establish a baseline

1. Note the cumulative load time for all five tables and enter it in the benchmarks table in the Before column.

   This is the value you noted in the previous step.

2. Record storage use.

   Determine how many 1 MB blocks of disk space are used for each table by querying the STV_BLOCKLIST table and record the results in your benchmarks table.

   ```sql
   select stv_tbl_perm.name as table, count(*) as mb
   from stv_blocklist, stv_tbl_perm
   where stv_blocklist.tbl = stv_tbl_perm.id
   and stv_blocklist.slice = stv_tbl_perm.slice
   and stv_tbl_perm.name in ('lineorder','part','customer','dwdate','supplier')
   group by stv_tbl_perm.name
   order by 1 asc;
   ```

   Your results should look similar to this:

<table>
<thead>
<tr>
<th>table</th>
<th>mb</th>
</tr>
</thead>
<tbody>
<tr>
<td>------</td>
<td>----</td>
</tr>
</tbody>
</table>

API Version 2012-12-01
3. Test query performance.

The first time you run a query, Amazon Redshift compiles the code, and then sends compiled code to the compute nodes. When you compare the execution times for queries, you should not use the results for the first time you execute the query. Instead, compare the times for the second execution of each query. For more information, see Benchmarking with compiled code (p. 195).

Run the following queries twice to eliminate compile time. Record the second time for each query in the benchmarks table.

```sql
-- Query 1
-- Restrictions on only one dimension.
select sum(lo_extendedprice*lo_discount) as revenue
from lineorder, dwdate
where lo_orderdate = d_datekey
and d_year = 1997
and lo_discount between 1 and 3
and lo_quantity < 24;

-- Query 2
-- Restrictions on two dimensions
select sum(lo_revenue), d_year, p_brand1
from lineorder, dwdate, part, supplier
where lo_orderdate = d_datekey
and lo_partkey = p_partkey
and lo_suppkey = s_suppkey
and p_category = 'MFGR#12'
and s_region = 'AMERICA'
group by d_year, p_brand1
order by d_year, p_brand1;

-- Query 3
-- Drill down in time to just one month
select c_city, s_city, d_year, sum(lo_revenue) as revenue
from customer, lineorder, supplier, dwdate
where lo_custkey = c_custkey
and lo_suppkey = s_suppkey
and lo_orderdate = d_datekey
and (c_city='UNITED KI1' or c_city='UNITED KI5')
and (s_city='UNITED KI1' or s_city='UNITED KI5')
and d_yearmonth = 'Dec1997'
group by c_city, s_city, d_year
order by d_year asc, revenue desc;
```

Your results for the second time will look something like this:
The following benchmarks table shows the example results for the cluster used in this tutorial.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load time (five tables)</td>
<td>10m 23s</td>
<td></td>
</tr>
</tbody>
</table>

**Storage Use**

<table>
<thead>
<tr>
<th>Table</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>LINEORDER</td>
<td>51024</td>
</tr>
<tr>
<td>PART</td>
<td>384</td>
</tr>
<tr>
<td>CUSTOMER</td>
<td>200</td>
</tr>
<tr>
<td>DWDATE</td>
<td>160</td>
</tr>
<tr>
<td>SUPPLIER</td>
<td>152</td>
</tr>
<tr>
<td><strong>Total storage</strong></td>
<td><strong>51920</strong></td>
</tr>
</tbody>
</table>

**Query execution time**

<table>
<thead>
<tr>
<th>Query</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6.97</td>
</tr>
<tr>
<td>2</td>
<td>12.81</td>
</tr>
<tr>
<td>3</td>
<td>13.39</td>
</tr>
<tr>
<td><strong>Total execution time</strong></td>
<td><strong>33.17</strong></td>
</tr>
</tbody>
</table>
Step 3: Select sort keys

When you create a table, you can specify one or more columns as the sort key. Amazon Redshift stores your data on disk in sorted order according to the sort key. How your data is sorted has an important effect on disk I/O, columnar compression, and query performance.

In this step, you choose sort keys for the SSB tables based on these best practices:

- If recent data is queried most frequently, specify the timestamp column as the leading column for the sort key.
- If you do frequent range filtering or equality filtering on one column, specify that column as the sort key.
- If you frequently join a (dimension) table, specify the join column as the sort key.

To select sort keys

1. Evaluate your queries to find timestamp columns that are used to filter the results.

   For example, LINEORDER frequently uses equality filters using lo_orderdate.

   ```sql
   where lo_orderdate = d_datekey and d_year = 1997
   ```

2. Look for columns that are used in range filters and equality filters. For example, LINEORDER also uses lo_orderdate for range filtering.

   ```sql
   where lo_orderdate = d_datekey and d_year >= 1992 and d_year <= 1997
   ```

3. Based on the first two best practices, lo_orderdate is a good choice for sort key.

   In the tuning table, specify lo_orderkey as the sort key for LINEORDER.

4. The remaining tables are dimensions, so, based on the third best practice, specify their primary keys as sort keys.

The following tuning table shows the chosen sort keys. You fill in the Distribution Style column in Step 4: Select distribution styles (p. 45).

<table>
<thead>
<tr>
<th>Table name</th>
<th>Sort Key</th>
<th>Distribution Style</th>
</tr>
</thead>
<tbody>
<tr>
<td>LINEORDER</td>
<td>lo_orderdate</td>
<td></td>
</tr>
<tr>
<td>PART</td>
<td>p_partkey</td>
<td></td>
</tr>
<tr>
<td>CUSTOMER</td>
<td>c_custkey</td>
<td></td>
</tr>
<tr>
<td>SUPPLIER</td>
<td>s_suppkey</td>
<td></td>
</tr>
<tr>
<td>DWDATE</td>
<td>d_datekey</td>
<td></td>
</tr>
</tbody>
</table>

Next step

Step 4: Select distribution styles (p. 45)
Step 4: Select distribution styles

When you load data into a table, Amazon Redshift distributes the rows of the table to each of the node slices according to the table’s distribution style. The number of slices is equal to the number of processor cores on the node. For example, the dw2.large cluster that you are using in this tutorial has four nodes, so it has eight slices. The nodes all participate in parallel query execution, working on data that is distributed across the slices.

When you execute a query, the query optimizer redistributes the rows to the compute nodes as needed to perform any joins and aggregations. Redistribution might involve either sending specific rows to nodes for joining or broadcasting an entire table to all of the nodes.

You should assign distribution styles to achieve these goals.

- Collocate the rows from joining tables
  - When the rows for joining columns are on the same slices, less data needs to be moved during query execution.
- Distribute data evenly among the slices in a cluster.
  - If data is distributed evenly, workload can be allocated evenly to all the slices.

These goals may conflict in some cases, and you will need to evaluate which strategy is the best choice for overall system performance. For example, even distribution might place all matching values for a column on the same slice. If a query uses an equality filter on that column, the slice with those values will carry a disproportionate share of the workload. If tables are collocated based on a distribution key, the rows might be distributed unevenly to the slices because the keys are distributed unevenly through the table.

In this step, you evaluate the distribution of the SSB tables with respect to the goals of data distribution, and then select the optimum distribution styles for the tables.

Distribution styles

When you create a table, you designate one of three distribution styles: KEY, ALL, or EVEN.

KEY distribution

The rows are distributed according to the values in one column. The leader node will attempt to place matching values on the same node slice. If you distribute a pair of tables on the joining keys, the leader node collocates the rows on the slices according to the values in the joining columns so that matching values from the common columns are physically stored together.

ALL distribution

A copy of the entire table is distributed to every node. Where EVEN distribution or KEY distribution place only a portion of a table’s rows on each node, ALL distribution ensures that every row is collocated for every join that the table participates in.

EVEN distribution

The rows are distributed across the slices in a round-robin fashion, regardless of the values in any particular column. EVEN distribution is appropriate when a table does not participate in joins or when there is not a clear choice between KEY distribution and ALL distribution. EVEN distribution is the default distribution style.

For more information, see Distribution styles (p. 102).
To select distribution styles

When you execute a query, the query optimizer redistributes the rows to the compute nodes as needed to perform any joins and aggregations. By locating the data where it needs to be before the query is executed, you can minimize the impact of the redistribution step.

The first goal is to distribute the data so that the matching rows from joining tables are collocated, which means that the matching rows from joining tables are located on the same node slice.

1. To look for redistribution steps in the query plan, execute an EXPLAIN command followed by the query. This example uses Query 2 from our set of test queries.

```sql
explain
select sum(lo_revenue), d_year, p_brand1
from lineorder, dwdate, part, supplier
where lo_orderdate = d_datekey
and lo_partkey = p_partkey
and lo_suppkey = s_suppkey
and p_category = 'MFGR#12'
and s_region = 'AMERICA'
group by d_year, p_brand1
order by d_year, p_brand1;
```

The following shows a portion of the query plan. Look for labels that begin with `DS_BCAST` or `DS_DIST` labels.

```
<table>
<thead>
<tr>
<th>QUERY PLAN</th>
</tr>
</thead>
<tbody>
<tr>
<td>XN Merge  (cost=1038007224737.84..1038007224738.54 rows=280 width=20)</td>
</tr>
<tr>
<td>Merge Key: dwdate.d_year, part.p_brand1</td>
</tr>
<tr>
<td>-&gt; XN Network (cost=1038007224737.84..1038007224738.54 rows=280 width=20)</td>
</tr>
<tr>
<td>Send to leader</td>
</tr>
<tr>
<td>-&gt; XN Sort (cost=1038007224737.84..1038007224738.54 rows=280 width=20)</td>
</tr>
<tr>
<td>Sort Key: dwdate.d_year, part.p_brand1</td>
</tr>
<tr>
<td>-&gt; XN HashAggregate (cost=38007224725.76..38007224726.46 rows=280)</td>
</tr>
<tr>
<td>-&gt; XN Hash Join DS_BCAST_INNER (cost=30674.95..38007188507.46)</td>
</tr>
<tr>
<td>Hash Cond: (&quot;outer&quot;.lo_orderdate = &quot;inner&quot;.d_datekey)</td>
</tr>
<tr>
<td>-&gt; XN Hash Join DS_BCAST_INNER (cost=30643.00..37598119820.65)</td>
</tr>
<tr>
<td>Hash Cond: (&quot;outer&quot;.lo_suppkey = &quot;inner&quot;.s_suppkey)</td>
</tr>
<tr>
<td>-&gt; XN Hash Join DS_BCAST_INNER (cost=17500.00..17500.00)</td>
</tr>
<tr>
<td>Hash Cond: (&quot;outer&quot;.lo_partkey = &quot;inner&quot;.p_partkey)</td>
</tr>
<tr>
<td>-&gt; XN Seq Scan on lineorder (cost=0.00..17500.00)</td>
</tr>
<tr>
<td>-&gt; XN Hash (cost=12500.00..12500.00)</td>
</tr>
<tr>
<td>Filter: ((p_category)::text = 'MFGR#12')</td>
</tr>
</tbody>
</table>
```
To select distribution styles

DS_BCAST_INNER indicates that the inner join table was broadcast to every slice. A DS_DIST_BOTH label, if present, would indicate that both the outer join table and the inner join table were redistributed across the slices. Broadcasting and redistribution can be expensive steps in terms of query performance. You want to select distribution strategies that reduce or eliminate broadcast and distribution steps. For more information about evaluating the EXPLAIN plan, see Evaluating query patterns (p. 103).

2. Distribute the fact table and one dimension table on their common columns.

The following diagram shows the relationships between the fact table, LINEORDER, and the dimension tables in the SSB schema.

```
<table>
<thead>
<tr>
<th>LINEORDER</th>
<th>CUSTOMER</th>
<th>SUPPLIER</th>
<th>DWDATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>lo_orderkey</td>
<td>c_custkey</td>
<td>s_suppkey</td>
<td>d_datekey</td>
</tr>
<tr>
<td>lo_linenumber</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lo_custkey</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lo_partkey</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lo_suppkey</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lo_orderdate</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lo_shippriority</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lo_quantity</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lo_extendedprice</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lo_ordertotalprice</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lo_discount</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lo_revenue</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lo_supplycost</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lo_tax</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lo_commdate</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lo_shipmode</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Each table can have only one distribution key, which means that only one pair of tables in the schema can be collocated on their common columns. The central fact table is the clear first choice. For the second table in the pair, choose the largest dimension that commonly joins the fact table. In this design, LINEORDER is the fact table, and PART is the largest dimension. PART joins LINEORDER on its primary key, p_partkey.

Designate lo_partkey as the distribution key for LINEORDER and p_partkey as the distribution key for PART so that the matching values for the joining keys will be collocated on the same slices when the data is loaded.

3. Change some dimension tables to use ALL distribution.

If a dimension table cannot be collocated with the fact table or other important joining tables, you can often improve query performance significantly by distributing the entire table to all of the nodes. ALL distribution guarantees that the joining rows will be collocated on every slice. You should weigh all factors before choosing ALL distribution. Using ALL distribution multiplies storage space requirements and increases load times and maintenance operations.

CUSTOMER, SUPPLIER, and DWDATE also join the LINEORDER table on their primary keys; however, LINEORDER will be collocated with PART, so you will set the remaining tables to use DISTSTYLE ALL. Because the tables are relatively small and are not updated frequently, using ALL distribution will have minimal impact on storage and load times.

4. Use EVEN distribution for the remaining tables.
All of the tables have been assigned with DISTKEY or ALL distribution styles, so you won't assign EVEN to any tables. After evaluating your performance results, you might decide to change some tables from ALL to EVEN distribution.

The following tuning table shows the chosen distribution styles.

<table>
<thead>
<tr>
<th>Table name</th>
<th>Sort Key</th>
<th>Distribution Style</th>
</tr>
</thead>
<tbody>
<tr>
<td>LINEORDER</td>
<td>lo_orderdate</td>
<td>lo_partkey</td>
</tr>
<tr>
<td>PART</td>
<td>p_partkey</td>
<td>p_partkey</td>
</tr>
<tr>
<td>CUSTOMER</td>
<td>c_custkey</td>
<td>ALL</td>
</tr>
<tr>
<td>SUPPLIER</td>
<td>s_suppkey</td>
<td>ALL</td>
</tr>
<tr>
<td>DWDATE</td>
<td>d_datekey</td>
<td>ALL</td>
</tr>
</tbody>
</table>

For more information, see Choose the best distribution style (p. 27).

Next step

Step 5: Review compression encodings (p. 48)

To review compression encodings

1. Find how much space each column uses.

Query the STV_BLOCKLIST system view to find the number of 1 MB blocks each column uses. The MAX aggregate function returns the highest block number for each column. This example uses `col < 6` in the WHERE clause to exclude system-generated columns.

Execute the following command.

```sql
select col, max(blocknum)
from stv_blocklist b, stv_tbl_perm p
where (b.tbl=p.id) and name ='lineorder'
and col < 6
group by name, col
order by col;
```
Your results will look similar to the following.

<table>
<thead>
<tr>
<th>col</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>286</td>
</tr>
<tr>
<td>1</td>
<td>286</td>
</tr>
<tr>
<td>2</td>
<td>286</td>
</tr>
<tr>
<td>3</td>
<td>286</td>
</tr>
<tr>
<td>4</td>
<td>286</td>
</tr>
<tr>
<td>5</td>
<td>286</td>
</tr>
</tbody>
</table>

2. Experiment with the different encoding methods.

In this step, you create a table with identical columns, except that each column uses a different compression encoding. Then you insert a large number of rows, using data from the `p_name` column in the `PART` table, so that every column has the same data. Finally, you will examine the table to compare the effects of the different encodings on column sizes.

a. Create a table with the encodings that you want to compare.

```sql
create table encodingshipmode (
    moderaw varchar(22) encode raw,
    modebytedict varchar(22) encode bytedict,
    modelzo varchar(22) encode lzo,
    moderunlength varchar(22) encode runlength,
    modetext255 varchar(22) encode text255,
    modetext32k varchar(22) encode text32k);
```

b. Insert the same data into all of the columns using an INSERT statement with a SELECT clause. The command will take a couple minutes to execute.

```sql
insert into encodingshipmode
select lo_shipmode as moderaw, lo_shipmode as modebytedict, lo_shipmode
    as modelzo,
    lo_shipmode as moderunlength, lo_shipmode as modetext255,
    lo_shipmode as modetext32k
from lineorder where lo_orderkey < 200000000;
```

c. Query the STV_BLOCKLIST system table to compare the number of 1 MB disk blocks used by each column.

```sql
select col, max(blocknum)
from stv_blocklist b, stv_tbl_perm p
where (b.tbl=p.id) and name = 'encodingshipmode'
and col < 6
group by name, col
order by col;
```

The query returns results similar to the following. Depending on how your cluster is configured, your results will be different, but the relative sizes should be similar.

<table>
<thead>
<tr>
<th>col</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>221</td>
</tr>
</tbody>
</table>
The columns show the results for the following encodings:

- Raw
- Bytedict
- LZO
- Runlength
- Text255
- Text32K

You can see that Bytedict encoding on the second column produced the best results for this data set, with a compression ratio of better than 8:1. Different data sets will produce different results, of course.

3. Use the ANALYZE COMPRESSION command to view the suggested encodings for an existing table.

   Execute the following command.

   ```
   analyze compression lineorder;
   ```

   Your results should look similar to the following.

<table>
<thead>
<tr>
<th>Table</th>
<th>Column</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>lineorder</td>
<td>lo_orderkey</td>
<td>delta</td>
</tr>
<tr>
<td>lineorder</td>
<td>lo_linenumber</td>
<td>delta</td>
</tr>
<tr>
<td>lineorder</td>
<td>lo_custkey</td>
<td>raw</td>
</tr>
<tr>
<td>lineorder</td>
<td>lo_partkey</td>
<td>raw</td>
</tr>
<tr>
<td>lineorder</td>
<td>lo_suppkey</td>
<td>raw</td>
</tr>
<tr>
<td>lineorder</td>
<td>lo_orderdate</td>
<td>delta32k</td>
</tr>
<tr>
<td>lineorder</td>
<td>lo_orderpriority</td>
<td>bytedict</td>
</tr>
<tr>
<td>lineorder</td>
<td>lo_shippriority</td>
<td>runlength</td>
</tr>
<tr>
<td>lineorder</td>
<td>lo_quantity</td>
<td>delta</td>
</tr>
<tr>
<td>lineorder</td>
<td>lo_extendedprice</td>
<td>lzo</td>
</tr>
<tr>
<td>lineorder</td>
<td>lo_ordertotalprice</td>
<td>lzo</td>
</tr>
<tr>
<td>lineorder</td>
<td>lo_discount</td>
<td>delta</td>
</tr>
<tr>
<td>lineorder</td>
<td>lo_revenue</td>
<td>lzo</td>
</tr>
<tr>
<td>lineorder</td>
<td>lo_supplycost</td>
<td>delta32k</td>
</tr>
<tr>
<td>lineorder</td>
<td>lo_tax</td>
<td>delta</td>
</tr>
<tr>
<td>lineorder</td>
<td>lo_commitdate</td>
<td>delta32k</td>
</tr>
<tr>
<td>lineorder</td>
<td>lo_shipmode</td>
<td>bytedict</td>
</tr>
</tbody>
</table>

   Notice that ANALYZE COMPRESSION chose BYTEDICT encoding for the `lo_shipmode` column.

   For an example that walks through choosing manually applied compression encodings, see Example: Choosing compression encodings for the CUSTOMER table (p. 99).

4. Apply automatic compression to the SSB tables.
By default, the COPY command automatically applies compression encodings when you load data into an empty table that has no compression encodings other than RAW encoding. For this tutorial, you will let the COPY command automatically select and apply optimal encodings for the tables as part of the next step, Recreate the test data set.

For more information, see Loading tables with automatic compression (p. 143).

Next step

Step 6: Recreate the test data set (p. 51)

Step 6: Recreate the test data set

Now that you have chosen the sort keys and distribution styles for each of the tables, you can create the tables using those attributes and reload the data. You will allow the COPY command to analyze the load data and apply compression encodings automatically.

To recreate the test data set

1. You need to drop the SSB tables before you run the CREATE TABLE commands.

   Execute the following commands.

   ```sql
   drop table part cascade;
drop table supplier cascade;
drop table customer cascade;
drop table dwdate cascade;
drop table lineorder cascade;
   ```

2. Create the tables with sort keys and distribution styles.

   Execute the following set of SQL CREATE TABLE commands.

   ```sql
   CREATE TABLE part (
     p_partkey    integer      not null sortkey distkey,
p    _name       varchar(22)  not null,
p    _mfr         varchar(6)      not null,
p    _category    varchar(7)      not null,
p    _brand1      varchar(9)      not null,
p    _color       varchar(11)  not null,
p    _type        varchar(25)  not null,
p    _size        integer      not null,
p    _container   varchar(10)     not null
   );

   CREATE TABLE supplier (
     s_suppkey      integer        not null sortkey,
s    _name         varchar(25)    not null,
s    _address      varchar(25)    not null,
s    _city         varchar(10)    not null,
s    _nation       varchar(15)    not null,
s    _region       varchar(12)     not null,
   );
   ```
To recreate the test data set

```sql
CREATE TABLE customer (
    c_custkey  integer  not null sortkey,
    c_name     varchar(25)    not null,
    c_address  varchar(25)    not null,
    c_city     varchar(10)    not null,
    c_nation   varchar(15)    not null,
    c_region   varchar(12)    not null,
    c_phone    varchar(15)    not null,
    c_mktsegment    varchar(10)    not null)
diststyle all;

CREATE TABLE dwdate (
    d_datekey            integer       not null sortkey,
    d_date               varchar(19)   not null,
    d_dayofweek       varchar(10)   not null,
    d_month           varchar(10)   not null,
    d_year               integer       not null,
    d_yearmonthnum       integer    not null,
    d_yearmonth          varchar(8) not null,
    d_daynuminweek       integer       not null,
    d_daynuminmonth      integer       not null,
    d_daynuminyear       integer       not null,
    d_monthnuminyear     integer       not null,
    d_weeknuminyear      integer       not null,
    d_sellingseason      varchar(13)    not null,
    d_lastdayinweekfl    varchar(1)    not null,
    d_lastdayinmonthfl   varchar(1)    not null,
    d_holidayfl          varchar(1)    not null,
    d_weekdayfl          varchar(1)    not null)
diststyle all;

CREATE TABLE lineorder (
    lo_orderkey           integer      not null sortkey,
    lo_linenumber         integer      not null,
    lo_custkey            integer      not null,
    lo_partkey            integer      not null distkey,
    lo_suppkey            integer      not null,
    lo_orderdate          integer      not null,
    lo_orderpriority      varchar(15)     not null,
    lo_shippriority       varchar(1)      not null,
    lo_quantity           integer      not null,
    lo_extendedprice      integer      not null,
    lo_ordertotalprice    integer      not null,
    lo_discount           integer      not null,
    lo_revenue            integer      not null,
    lo_supplycost         integer      not null,
    lo_tax                integer      not null,
    lo_commitdate         integer         not null,
    lo_shipmode           varchar(10)     not null

3. Load the tables using the same sample data.
   a. Open the loadssb.sql script that you created in the previous step.
```
b. Delete `compupdate off` from each COPY statement. This time, you will allow COPY to apply compression encodings.

For reference, the edited script should look like the following:

```sql
COPY customer FROM 's3://<region-specific-bucket-name>/ssbgz/customer' CREDENTIALS 'aws_access_key_id=<Your-Access-Key-ID>;aws_secret_access_key=<Your-Secret-Access-Key>' gzip;
COPY dwdate FROM 's3://<region-specific-bucket-name>/ssbgz/dwdate' CREDENTIALS 'aws_access_key_id=<Your-Access-Key-ID>;aws_secret_access_key=<Your-Secret-Access-Key>' gzip;
COPY lineorder FROM 's3://<region-specific-bucket-name>/ssbgz/lineorder' CREDENTIALS 'aws_access_key_id=<Your-Access-Key-ID>;aws_secret_access_key=<Your-Secret-Access-Key>' gzip;
COPY part FROM 's3://<region-specific-bucket-name>/ssbgz/part' CREDENTIALS 'aws_access_key_id=<Your-Access-Key-ID>;aws_secret_access_key=<Your-Secret-Access-Key>' gzip;
COPY supplier FROM 's3://<region-specific-bucket-name>/ssbgz/supplier' CREDENTIALS 'aws_access_key_id=<Your-Access-Key-ID>;aws_secret_access_key=<Your-Secret-Access-Key>' gzip;
```

c. Save the file.

d. Execute the COPY commands either by running the SQL script or by copying and pasting the commands into your SQL client.

**Note**
The load operation will take about 10 to 15 minutes. This might be a good time to get another cup of tea or feed the fish.

Your results should look similar to the following.

<table>
<thead>
<tr>
<th>Warnings:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load into table 'customer' completed, 3000000 record(s) loaded success fully.</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>Script execution finished</td>
</tr>
<tr>
<td>Total script execution time: 12m 15s</td>
</tr>
</tbody>
</table>


e. Record the load time in the benchmarks table.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load time (five tables)</td>
<td>10m 23s</td>
<td>12m 15s</td>
</tr>
<tr>
<td>Storage Use</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LINEORDER</td>
<td>51024</td>
<td></td>
</tr>
<tr>
<td>PART</td>
<td>384</td>
<td></td>
</tr>
<tr>
<td>CUSTOMER</td>
<td>200</td>
<td></td>
</tr>
</tbody>
</table>
Next step

Step 7: Retest system performance after tuning (p. 54)

Step 7: Retest system performance after tuning

After recreating the test data set with the selected sort keys, distribution styles, and compressions encodings, you will retest the system performance.

To retest system performance after tuning

1. Record storage use.

   Determine how many 1 MB blocks of disk space are used for each table by querying the STV_BLOCLLIST table and record the results in your benchmarks table.

   ```sql
   select stv_tbl_perm.name as "table", count(*) as "blocks (mb)"
   from stv_blocklist, stv_tbl_perm
   where stv_blocklist.tbl = stv_tbl_perm.id
   and stv_blocklist.slice = stv_tbl_perm.slice
   and stv_tbl_perm.name in ('customer', 'part', 'supplier', 'dwdate', 'line order')
   group by stv_tbl_perm.name
   order by 1 asc;
   ```

   Your results will look similar to this:

<table>
<thead>
<tr>
<th>table</th>
<th>blocks (mb)</th>
</tr>
</thead>
<tbody>
<tr>
<td>customer</td>
<td>604</td>
</tr>
<tr>
<td>dwdate</td>
<td>160</td>
</tr>
<tr>
<td>lineorder</td>
<td>27152</td>
</tr>
</tbody>
</table>
2. Check for distribution skew.

Uneven distribution, or data distribution skew, forces some nodes to do more work than others, which limits query performance.

To check for distribution skew, query the SVV_DISKUSAGE system view. Each row in SVV_DISKUSAGE records the statistics for one disk block. The num_values column gives the number of rows in that disk block, so \( \text{sum(num_values)} \) returns the number of rows on each slice.

Execute the following query to see the distribution for all of the tables in the SSB database.

```sql
select trim(name) as table, slice, sum(num_values) as rows, min(minvalue), max(maxvalue)
from svv_diskusage
where name in ('customer', 'part', 'supplier', 'dwdate', 'lineorder')
and col =0
group by name, slice
order by name, slice;
```

Your results will look something like this:

<table>
<thead>
<tr>
<th>table</th>
<th>slice</th>
<th>rows</th>
<th>min</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>customer</td>
<td>0</td>
<td>3000000</td>
<td>1</td>
<td>3000000</td>
</tr>
<tr>
<td>customer</td>
<td>2</td>
<td>3000000</td>
<td>1</td>
<td>3000000</td>
</tr>
<tr>
<td>customer</td>
<td>4</td>
<td>3000000</td>
<td>1</td>
<td>3000000</td>
</tr>
<tr>
<td>customer</td>
<td>6</td>
<td>3000000</td>
<td>1</td>
<td>3000000</td>
</tr>
<tr>
<td>dwdate</td>
<td>0</td>
<td>2556</td>
<td>19920101</td>
<td>19981230</td>
</tr>
<tr>
<td>dwdate</td>
<td>2</td>
<td>2556</td>
<td>19920101</td>
<td>19981230</td>
</tr>
<tr>
<td>dwdate</td>
<td>4</td>
<td>2556</td>
<td>19920101</td>
<td>19981230</td>
</tr>
<tr>
<td>dwdate</td>
<td>6</td>
<td>2556</td>
<td>19920101</td>
<td>19981230</td>
</tr>
<tr>
<td>lineorder</td>
<td>0</td>
<td>75029991</td>
<td>3</td>
<td>599999975</td>
</tr>
<tr>
<td>lineorder</td>
<td>1</td>
<td>75059242</td>
<td>7</td>
<td>60000000</td>
</tr>
<tr>
<td>lineorder</td>
<td>2</td>
<td>75238172</td>
<td>1</td>
<td>599999975</td>
</tr>
<tr>
<td>lineorder</td>
<td>3</td>
<td>75065416</td>
<td>1</td>
<td>599999973</td>
</tr>
<tr>
<td>lineorder</td>
<td>4</td>
<td>74801845</td>
<td>3</td>
<td>599999975</td>
</tr>
<tr>
<td>lineorder</td>
<td>5</td>
<td>75177053</td>
<td>1</td>
<td>599999975</td>
</tr>
<tr>
<td>lineorder</td>
<td>6</td>
<td>74631775</td>
<td>1</td>
<td>60000000</td>
</tr>
<tr>
<td>lineorder</td>
<td>7</td>
<td>75034408</td>
<td>1</td>
<td>599999974</td>
</tr>
<tr>
<td>part</td>
<td>0</td>
<td>175006</td>
<td>15</td>
<td>1399997</td>
</tr>
<tr>
<td>part</td>
<td>1</td>
<td>175199</td>
<td>1</td>
<td>1399999</td>
</tr>
<tr>
<td>part</td>
<td>2</td>
<td>175441</td>
<td>4</td>
<td>1399897</td>
</tr>
<tr>
<td>part</td>
<td>3</td>
<td>175000</td>
<td>3</td>
<td>1399995</td>
</tr>
<tr>
<td>part</td>
<td>4</td>
<td>175018</td>
<td>5</td>
<td>1399979</td>
</tr>
<tr>
<td>part</td>
<td>5</td>
<td>175091</td>
<td>11</td>
<td>1400000</td>
</tr>
<tr>
<td>part</td>
<td>6</td>
<td>174253</td>
<td>2</td>
<td>1399969</td>
</tr>
<tr>
<td>part</td>
<td>7</td>
<td>174992</td>
<td>13</td>
<td>1399996</td>
</tr>
<tr>
<td>supplier</td>
<td>0</td>
<td>1000000</td>
<td>1</td>
<td>1000000</td>
</tr>
<tr>
<td>supplier</td>
<td>2</td>
<td>1000000</td>
<td>1</td>
<td>1000000</td>
</tr>
<tr>
<td>supplier</td>
<td>4</td>
<td>1000000</td>
<td>1</td>
<td>1000000</td>
</tr>
<tr>
<td>supplier</td>
<td>6</td>
<td>1000000</td>
<td>1</td>
<td>1000000</td>
</tr>
</tbody>
</table>
(28 rows)
The following chart illustrates the distribution of the three largest tables. (The columns are not to scale.) Notice that because CUSTOMER uses ALL distribution, it was distributed to only one slice per node.

The distribution is relatively even, so you don't need to adjust for distribution skew.

3. Run an EXPLAIN command with each query to view the query plans.

The following example shows the EXPLAIN command with Query 2.

```
explain
select sum(lo_revenue), d_year, p_brand1
from lineorder, dwdate, part, supplier
where lo_orderdate = d_datekey
and lo_partkey = p_partkey
and lo_suppkey = s_suppkey
and p_category = 'MFGR#12'
and s_region = 'AMERICA'
group by d_year, p_brand1
order by d_year, p_brand1;
```

In the EXPLAIN plan for Query 2, notice that the DS_BCAST_INNER labels have been replaced by DS_DIST_ALL_NONE and DS_DIST_NONE, which means that no redistribution was required for those steps, and the query should run much more quickly.
4. Run the same test queries again.

As you did earlier, run the following queries twice to eliminate compile time. Record the second time for each query in the benchmarks table.

```
-- Query 1
-- Restrictions on only one dimension.
select sum(lo_extendedprice*lo_discount) as revenue
from lineorder, dwdate
where lo_orderdate = d_datekey
and d_year = 1997
and lo_discount between 1 and 3
and lo_quantity < 24;

-- Query 2
-- Restrictions on two dimensions
select sum(lo_revenue), d_year, p_brand1
from lineorder, dwdate, part, supplier
where lo_orderdate = d_datekey
and lo_partkey = p_partkey
and lo_suppkey = s_suppkey
and p_category = 'MFGR#12'
and s_region = 'AMERICA'
group by d_year, p_brand1
order by d_year, p_brand1;

-- Query 3
-- Drill down in time to just one month
select c_city, s_city, d_year, sum(lo_revenue) as revenue
from customer, lineorder, supplier, dwdate
where lo_custkey = c_custkey
and lo_suppkey = s_suppkey
and lo_orderdate = d_datekey
and (c_city='UNITED KI1' or c_city='UNITED KI5')
```

and (s_city='UNITED K11' or s_city='UNITED K15')
and d_yearmonth = 'Dec1997'
group by c_city, s_city, d_year
order by d_year asc, revenue desc;

The following benchmarks table shows the results based on the cluster used in this example. Your results will vary based on a number of factors, but the relative results should be similar.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>Storage Use</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LINEORDER</td>
<td>51024</td>
<td>27152</td>
</tr>
<tr>
<td>PART</td>
<td>200</td>
<td>200</td>
</tr>
<tr>
<td>CUSTOMER</td>
<td>384</td>
<td>604</td>
</tr>
<tr>
<td>DWDATE</td>
<td>160</td>
<td>160</td>
</tr>
<tr>
<td>SUPPLIER</td>
<td>152</td>
<td>236</td>
</tr>
<tr>
<td>Total storage</td>
<td>51920</td>
<td>28352</td>
</tr>
<tr>
<td>Query execution time</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Query 1</td>
<td>6.97</td>
<td>3.19</td>
</tr>
<tr>
<td>Query 2</td>
<td>12.81</td>
<td>9.02</td>
</tr>
<tr>
<td>Query 3</td>
<td>13.39</td>
<td>10.54</td>
</tr>
<tr>
<td>Total execution time</td>
<td>33.17</td>
<td>22.75</td>
</tr>
</tbody>
</table>

**Next step**

Step 8: Evaluate the results (p. 58)

**Step 8: Evaluate the results**

You tested load times, storage requirements, and query execution times before and after tuning the tables, and recorded the results.

The following table shows the example results for the cluster that was used for this tutorial. Your results will be different, but should show similar improvements.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Before</th>
<th>After</th>
<th>Change</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load time (five tables)</td>
<td>623</td>
<td>732</td>
<td>109</td>
<td>17.5%</td>
</tr>
</tbody>
</table>
## Benchmark Results

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Before</th>
<th>After</th>
<th>Change</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Storage Use</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LINEORDER</td>
<td>51024</td>
<td>27152</td>
<td>-23872</td>
<td>-46.8%</td>
</tr>
<tr>
<td>PART</td>
<td>200</td>
<td>200</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>CUSTOMER</td>
<td>384</td>
<td>604</td>
<td>220</td>
<td>57.3%</td>
</tr>
<tr>
<td>DWDATE</td>
<td>160</td>
<td>160</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>SUPPLIER</td>
<td>152</td>
<td>236</td>
<td>84</td>
<td>55.3%</td>
</tr>
<tr>
<td><strong>Total storage</strong></td>
<td>51920</td>
<td>28352</td>
<td>-23568</td>
<td>-45.4%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Query execution time</strong></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Query 1</td>
<td>6.97</td>
<td>3.19</td>
<td>-3.78</td>
<td>-54.2%</td>
</tr>
<tr>
<td>Query 2</td>
<td>12.81</td>
<td>9.02</td>
<td>-3.79</td>
<td>-29.6%</td>
</tr>
<tr>
<td>Query 3</td>
<td>13.39</td>
<td>10.54</td>
<td>-2.85</td>
<td>-21.3%</td>
</tr>
<tr>
<td><strong>Total execution time</strong></td>
<td>33.17</td>
<td>22.75</td>
<td>-10.42</td>
<td>-31.4%</td>
</tr>
</tbody>
</table>

### Load time

Load time increased by 17.5%.

Sorting, compression, and distribution increase load time. In particular, in this case, you used automatic compression, which increases the load time for empty tables that don’t already have compression encodings. Subsequent loads to the same tables would be faster. You also increased load time by using ALL distribution. You could reduce load time by using EVEN or DISTKEY distribution instead for some of the tables, but that decision needs to be weighed against query performance.

### Storage requirements

Storage requirements were reduced by 45.4%.

Some of the storage improvement from using columnar compression was offset by using ALL distribution on some of the tables. Again, you could improve storage use by using EVEN or DISTKEY distribution instead for some of the tables, but that decision needs to be weighed against query performance.

### Distribution

You verified that there is no distribution skew as a result of your distribution choices.

By checking the EXPLAIN plan, you saw that data redistribution was eliminated for the test queries.

### Query execution time

Total query execution time was reduced by 31.4%.

The improvement in query performance was due to a combination of optimizing sort keys, distribution styles, and compression. Often, query performance can be improved even further by rewriting queries and configuring workload management (WLM). For more information, see Tuning Query Performance (p. 180).
Next step

Step 9: Clean up your resources (p. 60)

Step 9: Clean up your resources

Your cluster continues to accrue charges as long as it is running. When you have completed this tutorial, you should return your environment to the previous state by following the steps in Step 5: Revoke Access and Delete Your Sample Cluster in the Amazon Redshift Getting Started Guide.

If you want to keep the cluster, but recover the storage used by the SSB tables, execute the following commands.

```
drop table part cascade;
drop table supplier cascade;
drop table customer cascade;
drop table dwdate cascade;
drop table lineorder cascade;
```

Summary

In this tutorial, you learned how to optimize the design of your tables by applying table design best practices.

You chose sort keys for the SSB tables based on these best practices:

- If recent data is queried most frequently, specify the timestamp column as the leading column for the sort key.
- If you do frequent range filtering or equality filtering on one column, specify that column as the sort key.
- If you frequently join a (dimension) table, specify the join column as the sort key.

You applied the following best practices to improve the distribution of the tables:

- Distribute the fact table and one dimension table on their common columns
- Change some dimension tables to use ALL distribution

You evaluated the effects of compression on a table and determined that using automatic compression usually produces the best results.

For more information, see the following links:

- Best practices for designing tables (p. 26)
- Choose the best sort key (p. 27)
- Choosing a data distribution style (p. 101)
- Choosing a column compression type (p. 90)
• Analyzing table design (p. 113)

Next Step

For your next step, if you haven't done so already, we recommend taking Tutorial: Loading Data from Amazon S3 (p. 62).
Tutorial: Loading Data from Amazon S3

In this tutorial, you will walk through the process of loading data into your Amazon Redshift database tables from data files in an Amazon Simple Storage Service (Amazon S3) bucket from beginning to end.

In this tutorial, you will:

• Download data files that use CSV, character-delimited, and fixed width formats.
• Create an Amazon S3 bucket and then upload the data files to the bucket.
• Launch an Amazon Redshift cluster and create database tables.
• Use COPY commands to load the tables from the data files on Amazon S3.
• Troubleshoot load errors and modify your COPY commands to correct the errors.

Estimated time: 60 minutes

Estimated cost: $1.00 per hour for the cluster

Prerequisites

You will need the following prerequisites:

• An AWS account to launch an Amazon Redshift cluster and to create a bucket in Amazon S3.
• Your AWS credentials (an access key ID and secret access key) to load test data from Amazon S3. If you need to create new access keys, go to Administering Access Keys for IAM Users.

This tutorial is designed so that it can be taken by itself. In addition to this tutorial, we recommend completing the following tutorials to gain a more complete understanding of how to design and use Amazon Redshift databases:

• Getting Started with Amazon Redshift walks you through the process of creating an Amazon Redshift cluster and loading sample data.
• Tutorial: Tuning Table Design (p. 35) walks you step by step through the process of designing and tuning tables, including choosing sort keys, distribution styles, and compression encodings, and evaluating system performance before and after tuning.
Overview

You can add data to your Amazon Redshift tables either by using an INSERT command or by using a COPY command. At the scale and speed of an Amazon Redshift data warehouse, the COPY command is many times faster and more efficient than INSERT commands.

The COPY command uses the Amazon Redshift massively parallel processing (MPP) architecture to read and load data in parallel from multiple data sources. You can load from data files on Amazon S3, Amazon EMR, or any remote host accessible through a Secure Shell (SSH) connection, or you can load directly from an Amazon DynamoDB table.

In this tutorial, you will use the COPY command to load data from Amazon S3. Many of the principles presented here apply to loading from other data sources as well.

To learn more about using the COPY command, see these resources:

• Best practices for loading data (p. 29)
• Loading data from Amazon EMR (p. 127)
• Loading data from remote hosts (p. 134)
• Loading data from an Amazon DynamoDB table (p. 140)

Steps

• Step 1: Launch a Cluster (p. 63)
• Step 2: Download the Data Files (p. 64)
• Step 3: Upload the Files to an Amazon S3 Bucket (p. 65)
• Step 4: Create the Sample Tables (p. 66)
• Step 5: Run the COPY Commands (p. 69)
• Step 6: Vacuum and Analyze the Database (p. 81)
• Step 7: Clean Up Your Resources (p. 81)

Step 1: Launch a Cluster

If you already have a cluster that you want to use, you can skip this step.

For the exercises in this tutorial, you will use a four-node cluster. Follow the steps in Getting Started with Amazon Redshift, but select Multi Node for Cluster Type and set Number of Compute Nodes to 4.
Follow the Getting Started steps to connect to your cluster from a SQL client and test a connection. You do not need to complete the remaining Getting Started steps to create tables, upload data, and try example queries.

**Next Step**

Step 2: Download the Data Files (p. 64)

**Step 2: Download the Data Files**

In this step, you will download a set of sample data files to your computer. In the next step, you will upload the files to an Amazon S3 bucket.

**To download the data files**

1. Download the zipped file from the following link: LoadingDataSampleFiles.zip
2. Extract the files to a folder on your computer.
3. Verify that your folder contains the following files.

   ```
   customer-fw-manifest
   customer-fw.tbl-000
   customer-fw.tbl-000.bak
   customer-fw.tbl-001
   customer-fw.tbl-002
   customer-fw.tbl-003
   customer-fw.tbl-004
   customer-fw.tbl-005
   customer-fw.tbl-006
   customer-fw.tbl-007
   customer-fw.tbl.log
   ```
Next Step

Step 3: Upload the Files to an Amazon S3 Bucket (p. 65)

Step 3: Upload the Files to an Amazon S3 Bucket

In this step, you create an Amazon S3 bucket and upload the data files to the bucket.

To upload the files to an Amazon S3 bucket

1. Create a bucket in Amazon S3.

   1. Sign in to the AWS Management Console, and open the Amazon S3 console.
   2. Click Create Bucket.
   3. In the Bucket Name box of the Create a Bucket dialog box, type a bucket name.
      The bucket name you choose must be unique among all existing bucket names in Amazon S3. One way to help ensure uniqueness is to prefix your bucket names with the name of your organization. Bucket names must comply with certain rules. For more information, go to Bucket Restrictions and Limitations in the Amazon Simple Storage Service Developer Guide.
   4. Select a region.
      Create the bucket in the same region as your cluster. If your cluster is in the US East (N. Virginia) region, click US Standard.
   5. Click Create.
      When Amazon S3 successfully creates your bucket, the console displays your empty bucket in the Buckets panel.

2. Create a folder.

   1. Click the name of the new bucket.
   2. Click the Actions button, and click Create Folder in the drop-down list.
   3. Name the new bucket load.
Note
The bucket that you created is not in a sandbox. In this exercise, you will add objects to a real bucket, and you will be charged a nominal amount for the time that you store the objects in the bucket. For more information about Amazon S3 pricing, go to the Amazon S3 Pricing page.

3. Upload the data files the new Amazon S3 bucket.
   1. Click the name of the data folder.
   2. In the Upload - Select Files wizard, click Add Files.
      A file selection dialog box opens.
   3. Select all of the files you downloaded and extracted, and then click Open.
   4. Click Start Upload.

User Credentials
The Amazon Redshift COPY command must have access to read the file objects in the Amazon S3 bucket. If you use the same user credentials to create the Amazon S3 bucket and to run the Amazon Redshift COPY command, the COPY command will have all necessary permissions. If you want to use different user credentials, you can grant access by using the Amazon S3 access controls. The Amazon Redshift COPY command requires at least ListBucket and GetObject permissions to access the file objects in the Amazon S3 bucket. For more information about controlling access to Amazon S3 resources, go to Managing Access Permissions to Your Amazon S3 Resources.

Next Step
Step 4: Create the Sample Tables (p. 66)

Step 4: Create the Sample Tables
For this tutorial, you will use a set of five tables based on the Star Schema Benchmark (SSB) schema. The following diagram shows the SSB data model.
If the SSB tables already exist in the current database, you will need to drop the tables to remove them from the database before you create them using the CREATE TABLE commands in the next step. The tables used in this tutorial might have different attributes than the existing tables.

To create the sample tables

1. To drop the SSB tables, execute the following commands.

   ```sql
   drop table part cascade;
drop table supplier;
drop table customer;
drop table dwdate;
drop table lineorder;
   ```

2. Execute the following CREATE TABLE commands.

   ```sql
   CREATE TABLE part
   (p_partkey     INTEGER NOT NULL,
p_name        VARCHAR(22) NOT NULL,
p_mfgr        VARCHAR(6),
p_category    VARCHAR(7) NOT NULL,
p_brand1      VARCHAR(9) NOT NULL,
p_color       VARCHAR(11) NOT NULL,
p_type        VARCHAR(25) NOT NULL,
p_size        INTEGER NOT NULL,
p_container   VARCHAR(10) NOT NULL);

   CREATE TABLE supplier
   (s_suppkey   INTEGER NOT NULL,
s_name      VARCHAR(25) NOT NULL,
s_address   VARCHAR(15),
s_city      VARCHAR(10),
s_nation     VARCHAR(10),
s_region     VARCHAR(5),
s_phone      VARCHAR(15),
s_comment    VARCHAR(111));

   CREATE TABLE dwdate
   (d_datekey
   d_date, d_dayofweek, d_month, d_year,
d_yearmonthnum, d_yearmonthdow, d_weeknuminyear,
d_daynuminyear, d_numdaysinyear, d_lastdayinweek,
d_lastdayinmonth, d_holiday, d_weekday);
CREATE TABLE customer
(
    c_custkey      INTEGER NOT NULL,
    c_name         VARCHAR(25) NOT NULL,
    c_address      VARCHAR(25) NOT NULL,
    c_city         VARCHAR(10) NOT NULL,
    c_nation       VARCHAR(15) NOT NULL,
    c_region       VARCHAR(12) NOT NULL,
    c_phone        VARCHAR(15) NOT NULL,
    c_mktsegment   VARCHAR(10) NOT NULL
);

CREATE TABLE dwdate
(
    d_datekey            INTEGER NOT NULL,
    d_date               VARCHAR(19) NOT NULL,
    d_dayofweek          VARCHAR(10) NOT NULL,
    d_month              VARCHAR(10) NOT NULL,
    d_year               INTEGER NOT NULL,
    d_yearmonthnum       INTEGER NOT NULL,
    d_yearmonth          VARCHAR(8) NOT NULL,
    d_daynuminweek       INTEGER NOT NULL,
    d_daynuminmonth      INTEGER NOT NULL,
    d_daynuminyear       INTEGER NOT NULL,
    d_monthnuminyear     INTEGER NOT NULL,
    d_weeknuminyear      INTEGER NOT NULL,
    d_sellingseason      VARCHAR(13) NOT NULL,
    d_lastdayinweekfl    VARCHAR(1) NOT NULL,
    d_lastdayinmonthfl   VARCHAR(1) NOT NULL,
    d_holidayfl          VARCHAR(1) NOT NULL,
    d_weekdayfl          VARCHAR(1) NOT NULL
);

CREATE TABLE lineorder
(
    lo_orderkey          INTEGER NOT NULL,
    lo_linenumber        INTEGER NOT NULL,
    lo_custkey           INTEGER NOT NULL,
    lo_partkey           INTEGER NOT NULL,
    lo_suppkey           INTEGER NOT NULL,
    lo_orderdate         INTEGER NOT NULL,
    lo_orderpriority     VARCHAR(15) NOT NULL,
    lo_shippriority      VARCHAR(1) NOT NULL,
    lo_quantity          INTEGER NOT NULL,
    lo_extendedprice     INTEGER NOT NULL,
    lo_ordertotalprice   INTEGER NOT NULL,
    lo_discount          INTEGER NOT NULL,
    lo_revenue           INTEGER NOT NULL,
    lo_supplycost        INTEGER NOT NULL,
    lo_tax               INTEGER NOT NULL,
    lo_commitdate        INTEGER NOT NULL,
    lo_supplycost        INTEGER NOT NULL
);
You will run COPY commands to load each of the tables in the SSB schema. The COPY command examples demonstrate loading from different file formats, using several COPY command options, and troubleshooting load errors.

Topics

• COPY Command Syntax (p. 69)
• Loading the SSB Tables (p. 70)

COPY Command Syntax

The basic COPY (p. 276) command syntax is as follows.

```
COPY table_name [ column_list ] FROM data_source CREDENTIALS access_credentials
[options]
```

To execute a COPY command, you provide the following values.

**Table name**

The target table for the COPY command. The table must already exist in the database. The table can be temporary or persistent. The COPY command appends the new input data to any existing rows in the table.

**Column list**

By default, COPY loads fields from the source data to the table columns in order. You can optionally specify a column list, that is a comma-separated list of column names, to map data fields to specific columns. You will not use column lists in this tutorial. For more information, see column list (p. 277) in the COPY command reference.

**Data source**

You can use the COPY command to load data from an Amazon S3 bucket, an Amazon EMR cluster, a remote host using an SSH connection, or an Amazon DynamoDB table. For this tutorial, you will load from data files in an Amazon S3 bucket. When loading from Amazon S3, you must provide the name of the bucket and the location of the data files, by providing either an object path for the data files or the location of a manifest file that explicitly lists each data file and its location.

• Key prefix

An object stored in Amazon S3 is uniquely identified by an object key, which includes the bucket name, folder names, if any, and the object name. A key prefix refers to a set of objects with the same prefix.
The object path is a key prefix that the COPY command uses to load all objects that share the key prefix. For example, the key prefix `custdata.txt` can refer to a single file or to a set of files, including `custdata.txt.001`, `custdata.txt.002`, and so on.

- **Manifest file**

  If you need to load files with different prefixes, for example, from multiple buckets or folders, or if you need to exclude files that share a prefix, you can use a manifest file. A manifest file explicitly lists each load file and its unique object key. You will use a manifest file to load the PART table later in this tutorial.

---

**Credentials**

To access the AWS resources that contain the data to load, you must provide AWS access credentials (that is, an access key ID and a secret access key) for an AWS user or an IAM user with sufficient privileges. To load data from Amazon S3, the credentials must include ListBucket and GetObject permissions. Additional credentials are required if your data is encrypted or if you are using temporary access credentials. For more information, see CREDENTIALS (p. 281) in the COPY command reference. For more information about managing access, go to Managing Access Permissions to Your Amazon S3 Resources. If you do not have an access key ID and secret access key, you will need to get them. For more information, go to Administering Access Keys for IAM Users.

---

**Options**

You can specify a number of parameters with the COPY command to specify file formats, manage data formats, manage errors, and control other features. In this tutorial, you will use the following COPY command options and features:

- Key Prefix (p. 71)
- CSV Format (p. 71)
- NULL AS (p. 73)
- REGION (p. 74)
- Fixed-Width Format (p. 74)
- MAXERROR (p. 75)
- ACCEPTINVCHARS (p. 77)
- MANIFEST (p. 77)
- DATEFORMAT (p. 79)
- GZIP and LZOP (p. 79)
- COMPUPDATE (p. 79)
- Multiple Files (p. 79)

---

**Loading the SSB Tables**

You will use the following COPY commands to load each of the tables in the SSB schema. The command to each table demonstrates different COPY options and troubleshooting techniques.

To load the SSB tables, follow these steps:

1. Replace the Bucket Name and AWS Credentials (p. 71)
2. Load the PART Table Using NULL AS (p. 71)
3. Load the SUPPLIER table Using REGION (p. 73)
4. Load the CUSTOMER Table Using MANIFEST (p. 74)
5. Load the DWDATE Table Using DATEFORMAT (p. 78)
6. Load the LINEORDER Table Using Multiple Files (p. 79)
Replace the Bucket Name and AWS Credentials

The COPY commands in this tutorial are presented in the following format.

```sql
COPY table from 's3://<your-bucket-name>/load/key_prefix'
credentials 'aws_access_key_id=<Your-Access-Key-ID>;aws_secret_access_key=<Your-Secret-Access-Key>'
options;
```

For each COPY command, do the following:

1. Replace `<your-bucket-name>` with the name of a bucket in the same region as your cluster.
2. Replace `<Your-Access-Key-ID>` and `<Your-Secret-Access-Key>` with your own AWS IAM account credentials. The segment of the credentials string that is enclosed in single quotation marks must not contain any spaces or line breaks.

Load the PART Table Using NULL AS

In this step, you will use the CSV and NULL AS options to load the PART table.

The COPY command can load data from multiple files in parallel, which is much faster than loading from a single file. To demonstrate this principle, the data for each table in this tutorial is split into eight files, even though the files are very small. In a later step, you will compare the time difference between loading from a single file and loading from multiple files. For more information, see Split your load data into multiple files (p. 30).

**Key Prefix**

You can load from multiple files by specifying a key prefix for the file set, or by explicitly listing the files in a manifest file. In this step, you will use a key prefix. In a later step, you will use a manifest file. The key prefix `s3://mybucket/load/part-csv.tbl` loads the following set of the files in the load folder.

<table>
<thead>
<tr>
<th>File Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>part-csv.tbl-000</td>
</tr>
<tr>
<td>part-csv.tbl-001</td>
</tr>
<tr>
<td>part-csv.tbl-002</td>
</tr>
<tr>
<td>part-csv.tbl-003</td>
</tr>
<tr>
<td>part-csv.tbl-004</td>
</tr>
<tr>
<td>part-csv.tbl-005</td>
</tr>
<tr>
<td>part-csv.tbl-006</td>
</tr>
<tr>
<td>part-csv.tbl-007</td>
</tr>
</tbody>
</table>

**CSV Format**

CSV, which stands for comma separated values, is a common format used for importing and exporting spreadsheet data. CSV is more flexible than comma-delimited format because it enables you to include quoted strings within fields. The default quote character for COPY from CSV format is a double quotation mark ("), but you can specify another quote character by using the QUOTE AS option. When you use the quote character within the field, escape the character with an additional quote character.

The following excerpt from a CSV-formatted data file for the PART table shows strings enclosed in double quotation marks ("LARGE ANODIZED BRASS") and a string enclosed in two double quotation marks within a quoted string ("MEDIUM ""BURNISHED"" TIN").
The data for the PART table contains characters that will cause COPY to fail. In this exercise, you will troubleshoot the errors and correct them.

To load data that is in CSV format, add `csv` to your COPY command. Execute the following command to load the PART table.

```
copy part from 's3://<your-bucket-name>/load/part-csv.tbl' credentials 'aws_access_key_id=<Your-Access-Key-ID>;aws_secret_access_key=<Your-Secret-Access-Key>' csv;
```

You should get an error message similar to the following.

```
An error occurred when executing the SQL command:
copy part from 's3://<your-bucket-name>/load/part-csv.tbl' credentials' ...

ERROR: Load into table 'part' failed. Check 'stl_load_errors' system table for details. [SQL State=XX000]

Execution time: 1.46s
1 statement(s) failed.
1 statement(s) failed.
```

To get more information about the error, query the STL_LOAD_ERRORS table. The following query uses the SUBSTRING function to shorten columns for readability and uses LIMIT 10 to reduce the number of rows returned. You can adjust the values in `substring(filename,22,25)` to allow for the length of your bucket name.

```
select query, substring(filename,22,25) as filename,line_number as line,substring(colname,0,12) as column, type, position as pos, substring(raw_line,0,30) as line_text,substring(raw_field_value,0,15) as field_text,substring(err_reason,0,45) as reason from stl_load_errors order by query desc limit 10;
```

```
query |    filename      | line |  column   |    type    | pos |
--------+-------------------------+-----------+------------+------------+-----
333765 | part-csv.tbl-000 |    1 |           |            |   0 |
line_text        | field_text |                    reason
------------------+------------+----------------------------------------------
15,NUL next,     |            | Missing newline: Unexpected character 0x2c f
```

API Version 2012-12-01
NULL AS

The *part-csv.tbl* data files use the NULL terminator character (\x000 or \x0) to indicate NULL values.

**Note**

Despite very similar spelling, NUL and NULL are not the same. NUL is a UTF-8 character with codepoint x000 that is often used to indicate end of record (EOR). NULL is a SQL value that represents an absence of data.

By default, COPY treats a NUL terminator character as an EOR character and terminates the record, which often results in unexpected results or an error. Because there is no single standard method of indicating NULL in text data, the NULL AS COPY command option enables you to specify which character to substitute with NULL when loading the table. In this example, you want COPY to treat the NUL terminator character as a NULL value.

**Note**

The table column that receives the NULL value must be configured as *nullable*. That is, it must not include the NOT NULL constraint in the CREATE TABLE specification.

To load PART using the NULL AS option, execute the following COPY command.

```
copy part from 's3://<your-bucket-name>/load/part-csv.tbl' credentials 'aws_access_key_id=<Your-Access-Key-ID>;aws_secret_access_key=<Your-Secret-Access-Key>' csv null as '\000';
```

To verify that COPY loaded NULL values, execute the following command to select only the rows that contain NULL.

```
select p_partkey, p_name, p_mfgr, p_category from part where p_mfgr is null;
```

<table>
<thead>
<tr>
<th>p_partkey</th>
<th>p_name</th>
<th>p_mfgr</th>
<th>p_category</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>NUL next</td>
<td></td>
<td>MFGR#47</td>
</tr>
<tr>
<td>81</td>
<td>NUL next</td>
<td></td>
<td>MFGR#23</td>
</tr>
<tr>
<td>133</td>
<td>NUL next</td>
<td></td>
<td>MFGR#44</td>
</tr>
</tbody>
</table>

(2 rows)

## Load the SUPPLIER table Using REGION

In this step you will use the DELIMITER and REGION options to load the SUPPLIER table.

### Character-Delimited Format

The fields in a character-delimited file are separated by a specific character, such as a pipe character (|), a comma (,) or a tab (\t). Character-delimited files can use any single ASCII character, including one of the nonprinting ASCII characters, as the delimiter. You specify the delimiter character by using the DELIMITER option. The default delimiter is a pipe character (|).

The following excerpt from the data for the SUPPLIER table uses pipe-delimited format.

```
1|1|257368|465569|41365|19950218|2-HIGH|0|17|2608718|9783671|4|2504369|92072|2|19950331|TRUCK
1|2|257368|201928|8146|19950218|2-HIGH|0|36|6587676|9783671|9|5994785|109794|6|19950416|MAIL
```
Whenever possible, you should locate your load data in the same AWS region as your Amazon Redshift cluster. If your data and your cluster are in the same region, you reduce latency, minimize eventual consistency issues, and avoid cross-region data transfer costs. For more information, see Best practices for loading data (p. 29).

If you must load data from a different AWS region, use the REGION option to specify the AWS region in which the load data is located. If you specify a region, all of the load data, including manifest files, must be in the named region. For more information, see REGION (p. 285).

If your cluster is in the US East (N. Virginia) region, execute the following command to load the SUPPLIER table from pipe-delimited data in an Amazon S3 bucket located in the US West (Oregon) region. For this example, do not change the bucket name.

```
copy supplier from 's3://awssampledbuswest2/ssbgz/supplier.tbl'
    credentials 'aws_access_key_id=<Your-Access-Key-ID>
    ;aws_secret_access_key=<Your-Secret-Access-Key>'
    delimiter '|'
    gzip
    region 'us-west-2';
```

If your cluster is not in the US East (N. Virginia) region, execute the following command to load the SUPPLIER table from pipe-delimited data in an Amazon S3 bucket located in the US East (N. Virginia) region. For this example, do not change the bucket name.

```
copy supplier from 's3://awssampledb/ssbgz/supplier.tbl'
    credentials 'aws_access_key_id=<Your-Access-Key-ID>
    ;aws_secret_access_key=<Your-Secret-Access-Key>'
    delimiter '|'
    gzip
    region 'us-east-1';
```

Load the CUSTOMER Table Using MANIFEST

In this step, you will use the FIXEDWIDTH, MAXERROR, ACCEPTINVCHARS, and MANIFEST options to load the CUSTOMER table.

The sample data for this exercise contains characters that will cause errors when COPY attempts to load them. You will use the MAXERRORS option and the STL_LOAD_ERRORS system table to troubleshoot the load errors and then use the ACCEPTINVCHARS and MANIFEST options to eliminate the errors.

Fixed-Width Format

Fixed-width format defines each field as a fixed number of characters, rather than separating fields with a delimeter. The following excerpt from the data for the CUSTOMER table uses fixed-width format.

<table>
<thead>
<tr>
<th></th>
<th>Customer#000000001</th>
<th>IVhzIApeRb</th>
<th>MOROCCO</th>
<th>0MOROCCO</th>
<th>AFRICA 25-705</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Customer#000000002</td>
<td>XSTf4,NCwDNae6tE</td>
<td>JORDAN</td>
<td>6JORDAN</td>
<td>MIDDLE EAST 23-453</td>
</tr>
<tr>
<td>3</td>
<td>Customer#000000003</td>
<td>MG9kdTD</td>
<td>ARGENTINA5ARGENTINAAMERICA 11-783</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The order of the label/width pairs must match the order of the table columns exactly. For more information, see FIXEDWIDTH (p. 282).
The fixed-width specification string for the CUSTOMER table data is as follows.

```sql
fixedwidth 'c_custkey:10, c_name:25, c_address:25, c_city:10, c_nation:15, c_region :12, c_phone:15,c_mktsegment:10'
```

To load the CUSTOMER table from fixed-width data, execute the following command.

```sql
copy customer
from 's3://<your-bucket-name>/load/customer-fw.tbl'
credentials 'aws_access_key_id=<Your-Access-Key-ID>;aws_secret_access_key=<Your-Secret-Access-Key>
fixedwidth 'c_custkey:10, c_name:25, c_address:25, c_city:10, c_nation:15, c_region :12, c_phone:15,c_mktsegment:10';
```

You should get an error message, similar to the following.

```
An error occurred when executing the SQL command:
copy customer
from 's3://mybucket/load/customer-fw.tbl'
credentials"aws_access_key_id=..."
ERROR: Load into table 'customer' failed. Check 'stl_load_errors' system table for details. [SQL State=XX000]
Execution time: 2.95s
1 statement(s) failed.
```

**MAXERROR**

By default, the first time COPY encounters an error, the command fails and returns an error message. To save time during testing, you can use the MAXERROR option to instruct COPY to skip a specified number of errors before it fails. Because we expect errors the first time we test loading the CUSTOMER table data, add `maxerror 10` to the COPY command.

To test using the FIXEDWIDTH and MAXERROR options, execute the following command.

```sql
copy customer
from 's3://<your-bucket-name>/load/customer-fw.tbl'
credentials 'aws_access_key_id=<Your-Access-Key-ID>;aws_secret_access_key=<Your-Secret-Access-Key>
fixedwidth 'c_custkey:10, c_name:25, c_address:25, c_city:10, c_nation:15, c_region :12, c_phone:15,c_mktsegment:10'
maxerror 10;
```

This time, instead of an error message, you get a warning message similar to the following.

```
Warnings:
Load into table 'customer' completed, 112497 record(s) loaded successfully.
Load into table 'customer' completed, 7 record(s) could not be loaded. Check 'stl_load_errors' system table for details.
```

The warning indicates that COPY encountered seven errors. To check the errors, query the STL_LOAD_ERRORS table, as shown in the following example.
select query, substring(filename,22,25) as filename, line_number as line, substring(colname,0,12) as column, type, position as pos, substring(raw_line,0,30) as line_text, substring(raw_field_value,0,15) as field_text, substring(err_reason,0,45) as error_reason from stl_load_errors order by query desc, filename limit 7;

The results of the STL_LOAD_ERRORS query should look similar to the following.

<table>
<thead>
<tr>
<th>query</th>
<th>filename</th>
<th>line</th>
<th>column</th>
<th>type</th>
<th>pos</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>line_text</td>
<td>field_text</td>
<td>error_reason</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------</td>
<td>---------------------------</td>
<td>------</td>
<td>----------</td>
<td>----------</td>
<td>-----</td>
</tr>
<tr>
<td></td>
<td>--------+---------------------------+------------+------------------------------------</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>----------+---------------------------+------------+------------------------------------</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>334489</td>
<td>customer-fw.tbl.log</td>
<td>2</td>
<td>c_custkey</td>
<td>int4</td>
<td>-1</td>
</tr>
<tr>
<td></td>
<td>customer-fw.tbl</td>
<td>customer-f</td>
<td>Invalid digit, Value 'c', Pos 0,</td>
<td>Type: Integ</td>
<td></td>
</tr>
<tr>
<td>334489</td>
<td>customer-fw.tbl.log</td>
<td>6</td>
<td>c_custkey</td>
<td>int4</td>
<td>-1</td>
</tr>
<tr>
<td></td>
<td>Complete</td>
<td>Complete</td>
<td>Invalid digit, Value 'C', Pos 0,</td>
<td>Type: Integ</td>
<td></td>
</tr>
<tr>
<td>334489</td>
<td>customer-fw.tbl.log</td>
<td>3</td>
<td>c_custkey</td>
<td>int4</td>
<td>-1</td>
</tr>
<tr>
<td></td>
<td>#Total rows</td>
<td>#Total row</td>
<td>Invalid digit, Value '#', Pos 0,</td>
<td>Type: Integ</td>
<td></td>
</tr>
<tr>
<td>334489</td>
<td>customer-fw.tbl.log</td>
<td>5</td>
<td>c_custkey</td>
<td>int4</td>
<td>-1</td>
</tr>
<tr>
<td></td>
<td>#Status</td>
<td>#Status</td>
<td>Invalid digit, Value '#', Pos 0,</td>
<td>Type: Integ</td>
<td></td>
</tr>
<tr>
<td>334489</td>
<td>customer-fw.tbl.log</td>
<td>1</td>
<td>c_custkey</td>
<td>int4</td>
<td>-1</td>
</tr>
<tr>
<td></td>
<td>#Load file</td>
<td>#Load file</td>
<td>Invalid digit, Value '#', Pos 0,</td>
<td>Type: Integ</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Customer#000000001</td>
<td>.Mayag.ezR</td>
<td>String contains invalid or unsupported UTF8</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Customer#000000001</td>
<td>.Mayag.ezR</td>
<td>String contains invalid or unsupported UTF8</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(7 rows)

By examining the results, you can see that there are two messages in the error_reason column:

• Invalid digit, Value 'c', Pos 0, Type: Integ

These errors are caused by the customer-fw.tbl.log file. The problem is that it is a log file, not a data file, and should not be loaded. You can use a manifest file to avoid loading the wrong file.

• String contains invalid or unsupported UTF8

The VARCHAR data type supports multibyte UTF-8 characters up to three bytes. If the load data contains unsupported or invalid characters, you can use the ACCEPTINVCHARS option to replace each invalid character with a specified alternative character.

Another problem with the load is more difficult to detect—the load produced unexpected results. To investigate this problem, execute the following command to query the CUSTOMER table.
The rows should be unique, but there are duplicates.

Another way to check for unexpected results is to verify the number of rows that were loaded. In our case, 100000 rows should have been loaded, but the load message reported loading 112497 records. The extra rows were loaded because the COPY loaded an extraneous file, customer-fw.tbl0000.bak.

In this exercise, you will use a manifest file to avoid loading the wrong files.

ACCEPTINVCHARS

By default, when COPY encounters a character that is not supported by the column's data type, it skips the row and returns an error. For information about invalid UTF-8 characters, see Multi-byte character load errors (p. 148).

You could use the MAXERRORS option to ignore errors and continue loading, then query STL_LOAD_ERRORS to locate the invalid characters, and then fix the data files. However, MAXERRORS is best used for troubleshooting load problems and should generally not be used in a production environment.

The ACCEPTINVCHARS option is usually a better choice for managing invalid characters. ACCEPTINVCHARS instructs COPY to replace each invalid character with a specified valid character and continue with the load operation. You can specify any valid ASCII character, except NULL, as the replacement character. The default replacement character is a question mark ( ? ). COPY replaces multibyte characters with a replacement string of equal length. For example, a 4-byte character would be replaced with '????'.

COPY returns the number of rows that contained invalid UTF-8 characters, and it adds an entry to the STL_REPLACEMENTS system table for each affected row, up to a maximum of 100 rows per node slice. Additional invalid UTF-8 characters are also replaced, but those replacement events are not recorded.

ACCEPTINVCHARS is valid only for VARCHAR columns.

For this step, you will add the ACCEPTINVCHARS with the replacement character '^^'.

MANIFEST

When you COPY from Amazon S3 using a key prefix, there is a risk that you will load unwanted tables. For example, the 's3://mybucket/load/' folder contains eight data files that share the key prefix
customer-fw.tbl,customer-fw.tbl0000,customer-fw.tbl0001, and so on. However, the same folder also contains the extraneous files customer-fw.tbl.log and customer-fw.tbl-0001.bak.

To ensure that you load all of the correct files, and only the correct files, use a manifest file. The manifest is a text file in JSON format that explicitly lists the unique object key for each source file to be loaded. The file objects can be in different folders or different buckets, but they must be in the same region. For more information, see MANIFEST (p. 285).

The following shows the customer-fw-manifest text.

```json
{
    "entries": [
        {"url":"s3://<your-bucket-name>/load/customer-fw.tbl-000"},
        {"url":"s3://<your-bucket-name>/load/customer-fw.tbl-001"},
        {"url":"s3://<your-bucket-name>/load/customer-fw.tbl-002"},
        {"url":"s3://<your-bucket-name>/load/customer-fw.tbl-003"},
        {"url":"s3://<your-bucket-name>/load/customer-fw.tbl-004"},
        {"url":"s3://<your-bucket-name>/load/customer-fw.tbl-005"},
        {"url":"s3://<your-bucket-name>/load/customer-fw.tbl-006"},
        {"url":"s3://<your-bucket-name>/load/customer-fw.tbl-007"}
    ]
}
```

To load the data for the CUSTOMER table using the manifest file

1. Open the file customer-fw-manifest in a text editor.
2. Replace `<your-bucket-name>` with the name of your bucket.
3. Save the file.
4. Upload the file to the load folder on your bucket.
5. Execute the following COPY command.

```sql
copy customer from 's3://<your-bucket-name>/load/customer-fw-manifest'
credentials 'aws_access_key_id=<Your-Access-Key-ID>;aws_secret_access_key=<Your-Secret-Access-Key>'
fixedwidth 'c_custkey:10, c_name:25, c_address:25, c_city:10, c_nation:15, c_region :12, c_phone:15,c_mktsegment:10'
maxerror 10
acceptinvchars as '^'
manifest;
```

Load the DWDATE Table Using DATEFORMAT

In this step, you will use the DELIMITER and DATEFORMAT options to load the DWDATE table.

When loading DATE and TIMESTAMP columns, COPY expects the default format, which is YYYY-MM-DD for dates and YYYY-MM-DD HH:MI:SS for time stamps. If the load data does not use a default format, you can use DATEFORMAT and TIMEFORMAT to specify the format.

The following excerpt shows date formats in the DWDATE table. Notice that the date formats in column two are inconsistent.
DATEFORMAT

You can specify only one date format. If the load data contains inconsistent formats, possibly in different columns, or if the format is not known at load time, you use DATEFORMAT with the 'auto' argument. When 'auto' is specified, COPY will recognize any valid date or time format and convert it to the default format. The 'auto' option recognizes several formats that are not supported when using a DATEFORMAT and TIMEFORMAT string. For more information, see Using Automatic Recognition with DATEFORMAT and TIMEFORMAT (p. 298).

To load the DWDATe table, execute the following COPY command.

```sql
COPY dwdate FROM 's3://<your-bucket-name>/load/dwdate-tab.tbl'
    CREDENTIALS 'aws_access_key_id=<Your-Access-Key-ID>;aws_secret_access_key=<Your-Secret-Access-Key>'
    DELIMITER '	'
    DATEFORMAT 'auto';
```

Load the LINEORDER Table Using Multiple Files

This step uses the GZIP and COMPUPDATE options to load the LINEORDER table.

In this exercise, you will load the LINEORDER table from a single data file, and then load it again from multiple files in order to compare the load times for the two methods.

GZIP and LZOP

You can compress your files using either the gzip or lzop compression format. When loading from compressed files, COPY uncompresses the files during the load process. Compressing your files saves storage space and shortens upload times.

COMPUPDATE

When COPY loads an empty table with no compression encodings, it analyzes the load data to determine the optimal encodings. It then alters the table to use those encodings before beginning the load. This analysis process takes time, but it occurs, at most, once per table. To save time, you can skip this step by turning COMPUPDATE off. To enable an accurate evaluation of COPY times, you will turn COMPUPDATE off for this step.

Multiple Files

The COPY command can load data very efficiently when it loads from multiple files in parallel instead of loading from a single file. If you split your data into files so that the number of files is a multiple of the number of slices in your cluster, Amazon Redshift divides the workload and distributes the data evenly among the slices. The number of slices is equal to the number of processor cores on the node.

For example, the dw2.large compute nodes used in this tutorial have two slices each, so the four-node cluster has eight slices. In previous steps, the load data was contained in eight files, even though the files are very small. In this step, you will compare the time difference between loading from a single large file and loading from multiple files.

The files you will use for this tutorial contain about 15 million records and occupy about 1.2 GB. These files are very small in Amazon Redshift scale, but sufficient to demonstrate the performance advantage of loading from multiple files. The files are large enough that the time required to download them and
then upload them to Amazon S3 is excessive for this tutorial, so you will load the files directly from an AWS sample bucket.

The following screenshot shows the data files for LINEORDER.

To evaluate the performance of COPY with multiple files

1. Execute the following command to COPY from a single file. Do not change the bucket name.

   ```
   copy lineorder from 's3://awssampledb/load/lo/lineorder-single.tbl'
   credentials 'aws_access_key_id=<Your-Access-Key-ID>;aws_secret_access_key=<Your-Secret-Access-Key>'
   gzip
   compupdate off
   region 'us-east-1';
   ```

2. Your results should be similar to the following. Note the execution time.

   Warnings:
   Load into table 'lineorder' completed, 14996734 record(s) loaded successfully.
   0 row(s) affected.
   copy executed successfully
   Execution time: 51.56s

3. Execute the following command to COPY from multiple files. Do not change the bucket name.

   ```
   copy lineorder from 's3://awssampledb/load/lo/lineorder-multi.tbl'
   credentials 'aws_access_key_id=<Your-Access-Key-ID>;aws_secret_access_key=<Your-Secret-Access-Key>'
   gzip
   compupdate off
   region 'us-east-1';
   ```

4. Your results should be similar to the following. Note the execution time.
5. Compare execution times.

In our example, the time to load 15 million records decreased from 51.56 seconds to 17.7 seconds, a reduction of 65.7 percent.

These results are based on using a four-node cluster. If your cluster has more nodes, the time savings is multiplied. For typical Amazon Redshift clusters, with tens to hundreds of nodes, the difference is even more dramatic. If you have a single node cluster, there is little difference between the execution times.

**Next Step**

Step 6: Vacuum and Analyze the Database (p. 81)

---

**Step 6: Vacuum and Analyze the Database**

Whenever you add, delete, or modify a significant number of rows, you should run a VACUUM command and then an ANALYZE command. A vacuum recovers the space from deleted rows and restores the sort order. The ANALYZE command updates the statistics metadata, which enables the query optimizer to generate more accurate query plans. For more information, see Vacuuming tables (p. 161).

If you load the data in sort key order, a vacuum is fast. In this tutorial, you added a significant number of rows, but you added them to empty tables. That being the case, there is no need to resort, and you didn't delete any rows. COPY automatically updates statistics after loading an empty table, so your statistics should be up-to-date. However, as a matter of good housekeeping, you will complete this tutorial by vacuuming and analyzing your database.

To vacuum and analyze the database, execute the following commands.

```
vacuum;
analyze;
```

**Next Step**

Step 7: Clean Up Your Resources (p. 81)

---

**Step 7: Clean Up Your Resources**

Your cluster continues to accrue charges as long as it is running. When you have completed this tutorial, you should return your environment to the previous state by following the steps in Step 5: Revoke Access and Delete Your Sample Cluster in the Amazon Redshift Getting Started Guide.
If you want to keep the cluster, but recover the storage used by the SSB tables, execute the following commands.

```sql
drop table part;
drop table supplier;
drop table customer;
drop table dwdate;
drop table lineorder;
```

Next

Summary (p. 82)

Summary

In this tutorial, you uploaded data files to Amazon S3 and then used COPY commands to load the data from the files into Amazon Redshift tables.

You loaded data using the following formats:

- Character-delimited
- CSV
- Fixed-width

You used the STL_LOAD_ERRORS system table to troubleshoot load errors, and then used the REGION, MANIFEST, MAXERROR, ACCEPTINVCHARS, DATEFORMAT, and NULL AS options to resolve the errors.

You applied the following best practices for loading data:

- Use a COPY command to load data (p. 29)
- Split your load data into multiple files (p. 30)
- Use a single COPY command to load from multiple files (p. 30)
- Compress your data files with gzip or lzop (p. 30)
- Use a manifest file (p. 30)
- Verify data files before and after a load (p. 30)

For more information about Amazon Redshift best practices, see the following links:

- Best practices for loading data (p. 29)
- Best practices for designing tables (p. 26)
- Best practices for tuning query performance (p. 32)

Next Step

For your next step, if you haven't done so already, we recommend taking Tutorial: Tuning Table Design (p. 35).
Managing Database Security

Topics

• Amazon Redshift security overview (p. 83)
• Default database user privileges (p. 84)
• Superusers (p. 84)
• Users (p. 85)
• Groups (p. 86)
• Schemas (p. 86)
• Example for controlling user and group access (p. 87)

You manage database security by controlling which users have access to which database objects.

Access to database objects depends on the privileges that you grant to user accounts or groups. The following guidelines summarize how database security works:

• By default, privileges are granted only to the object owner.
• Amazon Redshift database users are named user accounts that can connect to a database. A user account is granted privileges explicitly, by having those privileges assigned directly to the account, or implicitly, by being a member of a group that is granted privileges.
• Groups are collections of users that can be collectively assigned privileges for easier security maintenance.
• Schemas are collections of database tables and other database objects. Schemas are similar to operating system directories, except that schemas cannot be nested. Users can be granted access to a single schema or to multiple schemas.

For examples of security implementation, see Example for controlling user and group access (p. 87).

Amazon Redshift security overview

Amazon Redshift database security is distinct from other types of Amazon Redshift security. In addition to database security, which is described in this section, Amazon Redshift provides these features to manage security:
• **Sign-in credentials** — Access to your Amazon Redshift Management Console is controlled by your AWS account privileges. For more information, see Sign-In Credentials.

• **Access management** — To control access to specific Amazon Redshift resources, you define AWS Identity and Access Management (IAM) accounts. For more information, see Controlling Access to Amazon Redshift Resources.

• **Cluster security groups** — To grant other users inbound access to an Amazon Redshift cluster, you define a cluster security group and associate it with a cluster. For more information, see Amazon Redshift Cluster Security Groups.

• **VPC** — To protect access to your cluster by using a virtual networking environment, you can launch your cluster in a Virtual Private Cloud (VPC). For more information, see Managing Clusters in Virtual Private Cloud (VPC).

• **Cluster encryption** — To encrypt the data in all your user-created tables, you can enable cluster encryption when you launch the cluster. For more information, see Amazon Redshift Clusters.

• **SSL connections** — To encrypt the connection between your SQL client and your cluster, you can use secure sockets layer (SSL) encryption. For more information, see Connect to Your Cluster Using SSL.

• **Load data encryption** — To encrypt your table load data files when you upload them to Amazon S3, you can use either server-side encryption or client-side encryption. When you load from server-side encrypted data, Amazon S3 handles decryption transparently. When you load from client-side encrypted data, the Amazon Redshift COPY command decrypts the data as it loads the table. For more information, see Uploading encrypted data to Amazon S3 (p. 122).

• **Data in transit** — To protect your data in transit within the AWS cloud, Amazon Redshift uses hardware accelerated SSL to communicate with Amazon S3 or Amazon DynamoDB for COPY, UNLOAD, backup, and restore operations.

## Default database user privileges

When you create a database object, you are its owner. By default, only a superuser or the owner of an object can query, modify, or grant privileges on the object. For users to use an object, you must grant the necessary privileges to the user or the group that contains the user. Database superusers have the same privileges as database owners.

Amazon Redshift supports the following privileges: SELECT, INSERT, UPDATE, DELETE, REFERENCES, CREATE, TEMPORARY, EXECUTE, and USAGE. Different privileges are associated with different object types. For information on database object privileges supported by Amazon Redshift, see the GRANT (p. 346) command.

The right to modify or destroy an object is always the privilege of the owner only.

To revoke a privilege that was previously granted, use the REVOKE (p. 356) command. The privileges of the object owner, such as DROP, GRANT, and REVOKE privileges, are implicit and cannot be granted or revoked. Object owners can revoke their own ordinary privileges, for example, to make a table read-only for themselves as well as others. Superusers retain all privileges regardless of GRANT and REVOKE commands.

## Superusers

Database superusers have the same privileges as database owners for all databases.

The *masteruser*, which is the user you created when you launched the cluster, is a superuser.

You must be a superuser to create a superuser.
Amazon Redshift system tables and system views are designated either "superuser visible" or "user visible." Only superusers can query system tables and system views that are designated "superuser visible." For information, see System tables and views (p. 582).

Superusers can query all PostgreSQL catalog tables. For information, see System catalog tables (p. 687).

A database superuser bypasses all permission checks. Be very careful when using a superuser role. We recommend that you do most of your work as a role that is not a superuser. Superusers retain all privileges regardless of GRANT and REVOKE commands.

To create a new database superuser, log on to the database as a superuser and issue a CREATE USER command or an ALTER USER command with the CREATEUSER privilege.

```
create user adminuser createuser password '1234Admin';
alter user adminuser createuser;
```

Users

Amazon Redshift user accounts can only be created and dropped by a database superuser. Users are authenticated when they login to Amazon Redshift. They can own databases and database objects (for example, tables) and can grant privileges on those objects to users, groups, and schemas to control who has access to which object. Users with CREATE DATABASE rights can create databases and grant privileges to those databases. Superusers have database ownership privileges for all databases.

Creating, altering, and deleting users

Database users accounts are global across a data warehouse cluster (and not per individual database).

- To create a user use the CREATE USER (p. 325) command.
- To create a superuser use the CREATE USER (p. 325) command with the CREATUSER option.
- To remove an existing user, use the DROP USER (p. 336) command.
- To make changes to a user account, such as changing a password, use the ALTER USER (p. 266) command.
- To view a list of users, query the PG_USER catalog table:

```
select * from pg_user;
```

<table>
<thead>
<tr>
<th>username</th>
<th>usesysid</th>
<th>usecreatedb</th>
<th>usesuper</th>
<th>usecatupd</th>
<th>passwd</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rdsdb</td>
<td>1</td>
<td>t</td>
<td>t</td>
<td>t</td>
<td>********</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>masteruser</td>
<td>100</td>
<td>t</td>
<td>t</td>
<td>f</td>
<td>********</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dwuser</td>
<td>101</td>
<td>f</td>
<td>f</td>
<td>f</td>
<td>********</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>simpleuser</td>
<td>102</td>
<td>f</td>
<td>f</td>
<td>f</td>
<td>********</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>poweruser</td>
<td>103</td>
<td>f</td>
<td>t</td>
<td>f</td>
<td>********</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dbuser</td>
<td>104</td>
<td>t</td>
<td>f</td>
<td>f</td>
<td>********</td>
</tr>
</tbody>
</table>

API Version 2012-12-01
85
Groups

Groups are collections of users who are all granted whatever privileges are associated with the group. You can use groups to assign privileges by role. For example, you can create different groups for sales, administration, and support and give the users in each group the appropriate access to the data they require for their work. You can grant or revoke privileges at the group level, and those changes will apply to all members of the group, except for superusers.

To view all user groups, query the PG_GROUP system catalog table:

```
select * from pg_group;
```

Creating, altering, and deleting groups

Any user can create groups and alter or drop groups they own.

You can perform the following actions:

- To create a group, use the CREATE GROUP (p. 308) command.
- To add users to or remove users from an existing group, use the ALTER GROUP (p. 259) command.
- To delete a group, use the DROP GROUP (p. 333) command. This command only drops the group, not its member users.

Schemas

A database contains one or more named schemas. Each schema in a database contains tables and other kinds of named objects. By default, a database has a single schema, which is named PUBLIC. You can use schemas to group database objects under a common name. Schemas are similar to operating system directories, except that schemas cannot be nested.

Identical database object names can be used in different schemas in the same database without conflict. For example, both MY_SCHEMA and YOUR_SCHEMA can contain a table named MYTABLE. Users with the necessary privileges can access objects across multiple schemas in a database.

By default, an object is created within the first schema in the search path of the database. For information, see Search path (p. 87) later in this section.

Schemas can help with organization and concurrency issues in a multi-user environment in the following ways:

- To allow many developers to work in the same database without interfering with each other.
- To organize database objects into logical groups to make them more manageable.
- To give applications the ability to put their objects into separate schemas so that their names will not collide with the names of objects used by other applications.
Creating, altering, and deleting schemas

Any user can create groups and alter or drop groups they own.

You can perform the following actions:

• To create a schema, use the CREATE SCHEMA (p. 309) command.
• To change the owner of a schema, use the ALTER SCHEMA (p. 260) command.
• To delete a schema and its objects, use the DROP SCHEMA (p. 334) command.
• To create a table within a schema, create the table with the format `schema_name.table_name`.

To view all user groups, query the PG_NAMESPACE system catalog table:

```
select * from pg_namespace;
```

Search path

The search path is defined in the search_path parameter with a comma-separated list of schema names. The search path specifies the order in which schemas are searched when an object, such as a table or function, is referenced by a simple name that does not include a schema qualifier.

If an object is created without specifying a target schema, the object is added to the first schema that is listed in search path. When objects with identical names exist in different schemas, an object name that does not specify a schema will refer to the first schema in the search path that contains an object with that name.

To change the default schema for the current session, use the SET (p. 389) command.

For more information, see the search_path (p. 696) description in the Configuration Reference.

Schema-based privileges

Schema-based privileges are determined by the owner of the schema:

• By default, all users have CREATE and USAGE privileges on the PUBLIC schema of a database. To disallow users from creating objects in the PUBLIC schema of a database, use the REVOKE (p. 356) command to remove that privilege.
• Unless they are granted the USAGE privilege by the object owner, users cannot access any objects in schemas they do not own.
• If users have been granted the CREATE privilege to a schema that was created by another user, those users can create objects in that schema.

Example for controlling user and group access

This example creates user groups and user accounts and then grants them various privileges for an Amazon Redshift database that connects to a web application client. This example assumes three groups of users: regular users of a web application, power users of a web application, and web developers.

1. Create the groups where the user accounts will be assigned. The following set of commands creates three different user groups:
create group webappusers;
create group webpowerusers;
create group webdevusers;

2. Create several database user accounts with different privileges and add them to the groups.
   a. Create two users and add them to the WEBAPPUSERS group:

```sql
create user webappuser1 password 'webAppuser1pass'
in group webappusers;
create user webappuser2 password 'webAppuser2pass'
in group webappusers;
```

b. Create an account for a web developer and adds it to the WEBDEVUSERS group:

```sql
create user webdevuser1 password 'webDevuser2pass'
in group webdevusers;
```

c. Create a superuser account. This user will have administrative rights to create other users:

```sql
create user webappadmin  password 'webAppadminpass1'
createuser;
```

3. Create a schema to be associated with the database tables used by the web application, and grant
   the various user groups access to this schema:
   a. Create the WEBAPP schema:

```sql
create schema webapp;
```

b. Grant USAGE privileges to the WEBAPPUSERS group:

```sql
grant usage on schema webapp to group webappusers;
```

c. Grant USAGE privileges to the WEBPOWERUSERS group:

```sql
grant usage on schema webapp to group webpowerusers;
```

d. Grant ALL privileges to the WEBDEVUSERS group:

```sql
grant all on schema webapp to group webdevusers;
```

The basic users and groups are now set up. You can now make changes to alter the users and groups.

4. For example, the following command alters the search_path parameter for the WEBAPPUSER1.

```sql
alter user webappuser1 set search_path to webapp, public;
```
The SEARCH_PATH specifies the schema search order for database objects, such as tables and functions, when the object is referenced by a simple name with no schema specified.

5. You can also add users to a group after creating the group, such as adding WEBAPPUSER2 to the WEBPOWERUSERS group:

```
alter group webpowerusers add user webappuser2;
```
A data warehouse system has very different design goals as compared to a typical transaction-oriented relational database system. An online transaction processing (OLTP) application is focused primarily on single row transactions, inserts, and updates. Amazon Redshift is optimized for very fast execution of complex analytic queries against very large data sets. Because of the massive amount of data involved in data warehousing, you must specifically design your database to take full advantage of every available performance optimization.

This section explains how to choose and implement compression encodings, data distribution keys, sort keys, and table constraints, and it presents best practices for making these design decisions.

Choosing a column compression type

Compression is a column-level operation that reduces the size of data when it is stored. Compression conserves storage space and reduces the size of data that is read from storage, which reduces the amount of disk I/O and therefore improves query performance.

By default, Amazon Redshift stores data in its raw, uncompressed format. You can apply a compression type, or *encoding*, to the columns in a table manually when you create the table, or you can use the COPY command to analyze and apply compression automatically. For details about applying automatic compression, see Loading tables with automatic compression (p. 143).
**Note**

We strongly recommend using the COPY command to apply automatic compression.

You might choose to apply compression encodings manually if the new table shares the same data characteristics as another table, or if in testing you discover that the compression encodings that are applied during automatic compression are not the best fit for your data. If you choose to apply compression encodings manually, you can run the ANALYZE COMPRESSION (p. 269) command against an already populated table and use the results to choose compression encodings.

To apply compression manually, you specify compression encodings for individual columns as part of the CREATE TABLE statement. The syntax is as follows:

```sql
CREATE TABLE table_name (column_name data_type ENCODE encoding-type) [, ...]
```

Where `encoding-type` is taken from the keyword table in the following section.

For example, the following statement creates a two-column table, PRODUCT. When data is loaded into the table, the PRODUCT_ID column is not compressed, but the PRODUCT_NAME column is compressed, using the byte dictionary encoding (BYTEDICT).

```sql
create table product(
  product_id int,
  product_name char(20) encode bytedict);
```

You cannot change the compression encoding for a column after the table is created. You can specify the encoding for a column when it is added to a table using the ALTER TABLE command.

```sql
ALTER TABLE table-name ADD [ COLUMN ] column_name column_type
```

### Compression encodings

**Topics**

- Raw encoding (p. 92)
- Byte-dictionary encoding (p. 92)
- Delta encoding (p. 93)
- LZO encoding (p. 94)
- Mostly encoding (p. 94)
- Runlength encoding (p. 96)
- Text255 and text32k encodings (p. 97)

A compression encoding specifies the type of compression that is applied to a column of data values as rows are added to a table.

The following table identifies the supported compression encodings and the data types that support the encoding.

<table>
<thead>
<tr>
<th>Encoding type</th>
<th>Keyword in CREATE TABLE and ALTER TABLE</th>
<th>Data types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Raw (no compression)</td>
<td>RAW</td>
<td>All</td>
</tr>
</tbody>
</table>
### Encoding type | Keyword in CREATE TABLE and ALTER TABLE | Data types
---|---|---
Byte dictionary | BYTEDICT | All except BOOLEAN
Delta | DELTA | SMALLINT, INT, BIGINT, DATE, TIMESTAMP, DECIMAL
| DELTA32K | INT, BIGINT, DATE, TIMESTAMP, DECIMAL
LZO | LZO | All except BOOLEAN, REAL, and DOUBLE PRECISION
Mostlyn | MOSTLY8 | SMALLINT, INT, BIGINT, DECIMAL
| MOSTLY16 | INT, BIGINT, DECIMAL
| MOSTLY32 | BIGINT, DECIMAL
Run-length | RUNLENGTH | All
Text | TEXT255 | VARCHAR only
| TEXT32K | VARCHAR only

## Raw encoding

Raw encoding is the default storage method. With raw encoding, data is stored in raw, uncompressed form.

### Byte-dictionary encoding

In byte dictionary encoding, a separate dictionary of unique values is created for each block of column values on disk. (An Amazon Redshift disk block occupies 1 MB.) The dictionary contains up to 256 one-byte values that are stored as indexes to the original data values. If more than 256 values are stored in a single block, the extra values are written into the block in raw, uncompressed form. The process repeats for each disk block.

This encoding is very effective when a column contains a limited number of unique values. This encoding is optimal when the data domain of a column is fewer than 256 unique values. Byte-dictionary encoding is especially space-efficient if a CHAR column holds long character strings.

**Note**

Byte-dictionary encoding is not always effective when used with VARCHAR columns. Using BYTEDICT with large VARCHAR columns might cause excessive disk usage. We strongly recommend using a different encoding, such as LZO, for VARCHAR columns.

Suppose a table has a COUNTRY column with a CHAR(30) data type. As data is loaded, Amazon Redshift creates the dictionary and populates the COUNTRY column with the index value. The dictionary contains the indexed unique values, and the table itself contains only the one-byte subscripts of the corresponding values.

**Note**

Trailing blanks are stored for fixed-length character columns. Therefore, in a CHAR(30) column, every compressed value saves 29 bytes of storage when you use the byte-dictionary encoding.

The following table represents the dictionary for the COUNTRY column:
The following table represents the values in the COUNTRY column:

<table>
<thead>
<tr>
<th>Original data value</th>
<th>Original size (fixed length, 30 bytes per value)</th>
<th>Compressed value (index)</th>
<th>New size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>England</td>
<td>30</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>England</td>
<td>30</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>United States of America</td>
<td>30</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>United States of America</td>
<td>30</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Venezuela</td>
<td>30</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Sri Lanka</td>
<td>30</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Argentina</td>
<td>30</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>Japan</td>
<td>30</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>Sri Lanka</td>
<td>30</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Argentina</td>
<td>30</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Totals</td>
<td>300</td>
<td></td>
<td>10</td>
</tr>
</tbody>
</table>

The total compressed size in this example is calculated as follows: 6 different entries are stored in the dictionary (6 * 30 = 180), and the table contains 10 1-byte compressed values, for a total of 190 bytes.

**Delta encoding**

Delta encodings are very useful for datetime columns.

Delta encoding compresses data by recording the difference between values that follow each other in the column. This difference is recorded in a separate dictionary for each block of column values on disk. (An Amazon Redshift disk block occupies 1 MB.) For example, if the column contains 10 integers in sequence from 1 to 10, the first will be stored as a 4-byte integer (plus a 1-byte flag), and the next 9 will each be stored as a byte with the value 1, indicating that it is one greater than the previous value.
Delta encoding comes in two variations:

- DELTA records the differences as 1-byte values (8-bit integers)
- DELTA32K records differences as 2-byte values (16-bit integers)

If most of the values in the column could be compressed by using a single byte, the 1-byte variation is very effective; however, if the deltas are larger, this encoding, in the worst case, is somewhat less effective than storing the uncompressed data. Similar logic applies to the 16-bit version.

If the difference between two values exceeds the 1-byte range (DELTA) or 2-byte range (DELTA32K), the full original value is stored, with a leading 1-byte flag. The 1-byte range is from -127 to 127, and the 2-byte range is from -32K to 32K.

The following table shows how a delta encoding works for a numeric column:

<table>
<thead>
<tr>
<th>Original data value</th>
<th>Original size (bytes)</th>
<th>Difference (delta)</th>
<th>Compressed value</th>
<th>Compressed size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>1</td>
<td>1+4 (flag + actual value)</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>50</td>
<td>4</td>
<td>45</td>
<td>45</td>
<td>1</td>
</tr>
<tr>
<td>200</td>
<td>4</td>
<td>150</td>
<td>150</td>
<td>1+4 (flag + actual value)</td>
</tr>
<tr>
<td>185</td>
<td>4</td>
<td>-15</td>
<td>-15</td>
<td>1</td>
</tr>
<tr>
<td>220</td>
<td>4</td>
<td>35</td>
<td>35</td>
<td>1</td>
</tr>
<tr>
<td>221</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Totals</td>
<td>28</td>
<td></td>
<td></td>
<td>15</td>
</tr>
</tbody>
</table>

LZO encoding

LZO encoding provides a very high compression ratio with good performance. LZO encoding works especially well for CHAR and VARCHAR columns that store very long character strings, especially free form text, such as product descriptions, user comments, or JSON strings.

Note
COPY with automatic compression currently supports LZO encoding only for CHAR and VARCHAR; however, ANALYZE COMPRESSION (p. 269) fully supports LZO encoding. For more information, see Loading tables with automatic compression (p. 143).

Mostly encoding

Mostly encodings are useful when the data type for a column is larger than most of the stored values require. By specifying a mostly encoding for this type of column, you can compress the majority of the values in the column to a smaller standard storage size. The remaining values that cannot be compressed are stored in their raw form. For example, you can compress a 16-bit column, such as an INT2 column, to 8-bit storage.

In general, the mostly encodings work with the following data types:

- SMALLINT/INT2 (16-bit)
- INTEGER/INT (32-bit)
- BIGINT/INT8 (64-bit)
- DECIMAL/NUMERIC (64-bit)

Choose the appropriate variation of the mostly encoding to suit the size of the data type for the column. For example, apply MOSTLY8 to a column that is defined as a 16-bit integer column. Applying MOSTLY16 to a column with a 16-bit data type or MOSTLY32 to a column with a 32-bit data type is disallowed.

Mostly encodings might be less effective than no compression when a relatively high number of the values in the column cannot be compressed. Before applying one of these encodings to a column, check that most of the values that you are going to load now (and are likely to load in the future) fit into the ranges shown in the following table.

<table>
<thead>
<tr>
<th>Encoding</th>
<th>Compressed Storage Size</th>
<th>Range of values that can be compressed (values outside the range are stored raw)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOSTLY8</td>
<td>1 byte (8 bits)</td>
<td>-128 to 127</td>
</tr>
<tr>
<td>MOSTLY16</td>
<td>2 bytes (16 bits)</td>
<td>-32768 to 32767</td>
</tr>
<tr>
<td>MOSTLY32</td>
<td>4 bytes (32 bits)</td>
<td>-2147483648 to +2147483647</td>
</tr>
</tbody>
</table>

**Note**
For decimal values, ignore the decimal point to determine whether the value fits into the range. For example, 1,234.56 is treated as 123456 and can be compressed in a MOSTLY32 column.

For example, the VENUEID column in the VENUE table is defined as a raw integer column, which means that its values consume 4 bytes of storage. However, the current range of values in the column is 0 to 309. Therefore, re-creating and reloading this table with MOSTLY16 encoding for VENUEID would reduce the storage of every value in that column to 2 bytes.

If the VENUEID values referenced in another table were mostly in the range of 0 to 127, it might make sense to encode that foreign-key column as MOSTLY8. Before making the choice, you would have to run some queries against the referencing table data to find out whether the values mostly fall into the 8-bit, 16-bit, or 32-bit range.

The following table shows compressed sizes for specific numeric values when the MOSTLY8, MOSTLY16, and MOSTLY32 encodings are used:

<table>
<thead>
<tr>
<th>Original value</th>
<th>Original INT or BIGINT size (bytes)</th>
<th>MOSTLY8 compressed size (bytes)</th>
<th>MOSTLY16 compressed size (bytes)</th>
<th>MOSTLY32 compressed size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>10</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>100</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>
### Runlength encoding

Runlength encoding replaces a value that is repeated consecutively with a token that consists of the value and a count of the number of consecutive occurrences (the length of the run). A separate dictionary of unique values is created for each block of column values on disk. (An Amazon Redshift disk block occupies 1 MB.) This encoding is best suited to a table in which data values are often repeated consecutively, for example, when the table is sorted by those values.

For example, if a column in a large dimension table has a predictably small domain, such as a COLOR column with fewer than 10 possible values, these values are likely to fall in long sequences throughout the table, even if the data is not sorted.

We do not recommend applying runlength encoding on any column that is designated as a sort key. Range-restricted scans perform better when blocks contain similar numbers of rows. If sort key columns are compressed much more highly than other columns in the same query, range-restricted scans might perform poorly.

The following table uses the COLOR column example to show how the runlength encoding works:

<table>
<thead>
<tr>
<th>Original data value</th>
<th>Original size (bytes)</th>
<th>Compressed value (token)</th>
<th>Compressed size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blue</td>
<td>4</td>
<td>(2,Blue)</td>
<td>5</td>
</tr>
<tr>
<td>Blue</td>
<td>4</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>Green</td>
<td>5</td>
<td>(3,Green)</td>
<td>6</td>
</tr>
<tr>
<td>Green</td>
<td>5</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>Green</td>
<td>5</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>Blue</td>
<td>4</td>
<td>(1,Blue)</td>
<td>5</td>
</tr>
<tr>
<td>Yellow</td>
<td>6</td>
<td>(4,Yellow)</td>
<td>7</td>
</tr>
<tr>
<td>Yellow</td>
<td>6</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>Yellow</td>
<td>6</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>Yellow</td>
<td>6</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>Totals</td>
<td>51</td>
<td></td>
<td>23</td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th>Original value</th>
<th>Original INT or BIGINT size (bytes)</th>
<th>MOSTLY8 compressed size (bytes)</th>
<th>MOSTLY16 compressed size (bytes)</th>
<th>MOSTLY32 compressed size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>4</td>
<td>Same as raw data size</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>10000</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>20000</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>40000</td>
<td>8</td>
<td>Same as raw data size</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>100000</td>
<td>8</td>
<td>4</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>20000000000</td>
<td>8</td>
<td>4</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>
Text255 and text32k encodings

Text255 and text32k encodings are useful for compressing VARCHAR columns in which the same words recur often. A separate dictionary of unique words is created for each block of column values on disk. (An Amazon Redshift disk block occupies 1 MB.) The dictionary contains the first 245 unique words in the column. Those words are replaced on disk by a one-byte index value representing one of the 245 values, and any words that are not represented in the dictionary are stored uncompressed. The process repeats for each 1 MB disk block. If the indexed words occur frequently in the column, the column will yield a high compression ratio.

For the text32k encoding, the principle is the same, but the dictionary for each block does not capture a specific number of words. Instead, the dictionary indexes each unique word it finds until the combined entries reach a length of 32K, minus some overhead. The index values are stored in two bytes.

For example, consider the VENUENAME column in the VENUE table. Words such as Arena, Center, and Theatre recur in this column and are likely to be among the first 245 words encountered in each block if text255 compression is applied. If so, this column will benefit from compression because every time those words appear, they will occupy only 1 byte of storage (instead of 5, 6, or 7 bytes, respectively).

Testing compression encodings

If you decide to manually specify column encodings, you might want to test different encodings with your data.

Note

We recommend that you use the COPY command to load data whenever possible, and allow the COPY command to choose the optimal encodings based on your data. Alternatively, you can use the ANALYZE COMPRESSION (p. 269) command to view the suggested encodings for existing data. For details about applying automatic compression, see Loading tables with automatic compression (p. 143).

To perform a meaningful test of data compression, you need a large number of rows. For this example, we will create a table by using a CREATE TABLE AS statement that selects from two tables; VENUE and LISTING. We will leave out the WHERE clause that would normally join the two tables; the result is that each row in the VENUE table is joined to all of the rows in the LISTING table, for a total of over 32 million rows. This is known as a Cartesian join and normally is not recommended, but for this purpose, it is a convenient method of creating a lot of rows. If you have an existing table with data that you want to test, you can skip this step.

After we have a table with sample data, we create a table with six columns, each with a different compression encoding: raw, bytedict, lzo, runlength, text255, and text32k. We populate each column with exactly the same data by executing an INSERT command that selects the data from the first table.

To test compression encodings:

1. (Optional) First, we’ll create a table with a large number of rows. Skip this step if you want to test an existing table.

   ```sql
   create table reallybigvenue as
   select venueid, venuename, venuecity, venuestate, venueseats
   from venue, listing;
   ```

2. Next, create a table with the encodings that you want to compare.

   ```sql
   create table encodingvenue (
   venueraw varchar(100) encode raw,
   ```
3. Insert the same data into all of the columns using an INSERT statement with a SELECT clause.

```sql
insert into encodingvenue
select venuename as venueraw, venuename as venuebytedict, venuename as venuerunlength, venuename as venuetext32k, venuename as venuetext255
from reallybigvenue;
```

4. Verify the number of rows in the new table.

```sql
select count(*) from encodingvenue

<table>
<thead>
<tr>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>38884394</td>
</tr>
</tbody>
</table>
```

5. Query the STV_BLOCKLIST (p. 650) system table to compare the number of 1 MB disk blocks used by each column.

The MAX aggregate function returns the highest block number for each column. The STV_BLOCKLIST table includes details for three system-generated columns. This example uses `col < 6` in the WHERE clause to exclude the system-generated columns.

```sql
select col, max(blocknum)
from stv_blocklist b, stv_tbl_perm p
where (b.tbl=p.id) and name = 'encodingvenue'
and col < 6
group by name, col
order by col;

<table>
<thead>
<tr>
<th>col</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>203</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>22</td>
</tr>
<tr>
<td>3</td>
<td>204</td>
</tr>
<tr>
<td>4</td>
<td>56</td>
</tr>
<tr>
<td>5</td>
<td>72</td>
</tr>
</tbody>
</table>
```

The query returns the following results. The columns are numbered beginning with zero. Depending on how your cluster is configured, your result might have different numbers, but the relative sizes should be similar. You can see that BYTEDICT encoding on the second column produced the best results for this data set, with a compression ratio of better than 20:1. LZO encoding also produced excellent results. Different data sets will produce different results, of course. When a column contains longer text strings, LZO often produces the best compression results.
If you have data in an existing table, you can use the `ANALYZE COMPRESSION` (p. 269) command to view the suggested encodings for the table. For example, the following example shows the recommended encoding for a copy of the VENUE table, REALLYBIGVENUE, that contains 38 million rows. Notice that `ANALYZE COMPRESSION` recommends `BYTEDICT` encoding for the `VENUE_NAME` column, which agrees with the results of our previous test.

```
analyze compression reallybigvenue;
```

<table>
<thead>
<tr>
<th>Column</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>venueid</td>
<td>lzo</td>
</tr>
<tr>
<td>venuename</td>
<td>bytedict</td>
</tr>
<tr>
<td>venuecity</td>
<td>lzo</td>
</tr>
<tr>
<td>venuestate</td>
<td>lzo</td>
</tr>
<tr>
<td>venueseats</td>
<td>lzo</td>
</tr>
</tbody>
</table>
(5 rows)

**Example: Choosing compression encodings for the CUSTOMER table**

The following statement creates a CUSTOMER table that has columns with various data types. This `CREATE TABLE` statement shows one of many possible combinations of compression encodings for these columns.

```
create table customer(
  custkey int encode delta,
  custname varchar(30) encode raw,
  gender varchar(7) encode text255,
  address varchar(200) encode text255,
  city varchar(30) encode text255,
  state char(2) encode raw,
  zipcode char(5) encode bytedict,
  start_date date encode delta32k);
```

The following table shows the column encodings that were chosen for the CUSTOMER table and gives an explanation for the choices:

<table>
<thead>
<tr>
<th>Column</th>
<th>Data Type</th>
<th>Encoding</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSTKEY</td>
<td>int</td>
<td>delta</td>
<td>CUSTKEY consists of unique, consecutive integer values. Since the differences will be one byte, DELTA is a good choice.</td>
</tr>
<tr>
<td>CUSTNAME</td>
<td>varchar(30)</td>
<td>raw</td>
<td>CUSTNAME has a large domain with few repeated values. Any compression encoding would probably be ineffective.</td>
</tr>
</tbody>
</table>
Example: Choosing compression encodings for the CUSTOMER table

<table>
<thead>
<tr>
<th>Column</th>
<th>Data Type</th>
<th>Encoding</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>GENDER</td>
<td>varchar(7)</td>
<td>text255</td>
<td>GENDER is very small domain with many repeated values. Text255 works well with VARCHAR columns in which the same words recur.</td>
</tr>
<tr>
<td>ADDRESS</td>
<td>varchar(200)</td>
<td>text255</td>
<td>ADDRESS is a large domain, but contains many repeated words, such as Street Avenue, North, South, and so on. Text 255 and text 32k are useful for compressing VARCHAR columns in which the same words recur. The column length is short, so text255 is a good choice.</td>
</tr>
<tr>
<td>CITY</td>
<td>varchar(30)</td>
<td>text255</td>
<td>CITY is a large domain, with some repeated values. Certain city names are used much more commonly than others. Text255 is a good choice for the same reasons as ADDRESS.</td>
</tr>
<tr>
<td>STATE</td>
<td>char(2)</td>
<td>raw</td>
<td>In the United States, STATE is a precise domain of 50 two-character values. Bytedict encoding would yield some compression, but because the column size is only two characters, compression might not be worth the overhead of uncompressing the data.</td>
</tr>
<tr>
<td>ZIPCODE</td>
<td>char(5)</td>
<td>bytedict</td>
<td>ZIPCODE is a known domain of fewer than 50,000 unique values. Certain zip codes occur much more commonly than others. Bytedict encoding is very effective when a column contains a limited number of unique values.</td>
</tr>
</tbody>
</table>
Choosing a data distribution style

Topics

- Data distribution concepts (p. 101)
- Distribution styles (p. 102)
- Viewing distribution styles (p. 102)
- Evaluating query patterns (p. 103)
- Designating distribution styles (p. 103)
- Evaluating the query plan (p. 104)
- Query plan example (p. 106)
- Distribution examples (p. 111)

When you load data into a table, Amazon Redshift distributes the rows of the table to each of the compute nodes according to the table’s distribution style. When you execute a query, the query optimizer redistributes the rows to the compute nodes as needed to perform any joins and aggregations. The goal in selecting a table distribution style is to minimize the impact of the redistribution step by locating the data where it needs to be before the query is executed.

This section will introduce you to the principles of data distribution in an Amazon Redshift database and give you a methodology to choose the best distribution style for each of your tables.

Data distribution concepts

Nodes and slices

An Amazon Redshift cluster is a set of nodes. Each node in the cluster has its own operating system, dedicated memory, and dedicated disk storage. One node is the leader node, which manages the distribution of data and query processing tasks to the compute nodes.

The disk storage for a compute node is divided into a number of slices, equal to the number of processor cores on the node. For example, each DW1.XL compute node has two slices, and each DW2.8XL compute node has 32 slices. The nodes all participate in parallel query execution, working on data that is distributed as evenly as possible across the slices.

Data redistribution

When you load data into a table, Amazon Redshift distributes the rows of the table to each of the node slices according to the table’s distribution style. As part of a query plan, the optimizer determines where blocks of data need to be located to best execute the query. The data is then physically moved, or redistributed, during execution. Redistribution might involve either sending specific rows to nodes for joining or broadcasting an entire table to all of the nodes.

Data redistribution can account for a substantial portion of the cost of a query plan, and the network traffic it generates can affect other database operations and slow overall system performance. To the extent that you anticipate where best to locate data initially, you can minimize the impact of data redistribution.
Data distribution goals

When you load data into a table, Amazon Redshift distributes the table’s rows to the compute nodes and slices according to the distribution style that you chose when you created the table. Data distribution has two primary goals:

• To distribute the workload uniformly among the nodes in the cluster. Uneven distribution, or data distribution skew, forces some nodes to do more work than others, which impairs query performance.
• To minimize data movement during query execution. If the rows that participate in joins or aggregates are already collocated on the nodes with their joining rows in other tables, the optimizer does not need to redistribute as much data during query execution.

The distribution strategy that you choose for your database has important consequences for query performance, storage requirements, data loading, and maintenance. By choosing the best distribution style for each table, you can balance your data distribution and significantly improve overall system performance.

Distribution styles

When you create a table, you designate one of three distribution styles; EVEN, KEY, or ALL.

Even distribution

The leader node distributes the rows across the slices in a round-robin fashion, regardless of the values in any particular column. EVEN distribution is appropriate when a table does not participate in joins or when there is not a clear choice between KEY distribution and ALL distribution. EVEN distribution is the default distribution style.

Key distribution

The rows are distributed according to the values in one column. The leader node will attempt to place matching values on the same node slice. If you distribute a pair of tables on the joining keys, the leader node collocates the rows on the slices according to the values in the joining columns so that matching values from the common columns are physically stored together.

ALL distribution

A copy of the entire table is distributed to every node. Where EVEN distribution or KEY distribution place only a portion of a table’s rows on each node, ALL distribution ensures that every row is collocated for every join that the table participates in.

ALL distribution multiplies the storage required by the number of nodes in the cluster, and so it takes much longer to load, update, or insert data into multiple tables. ALL distribution is appropriate only for relatively slow moving tables; that is, tables that are not updated frequently or extensively. Small dimension tables do not benefit significantly from ALL distribution, because the cost of redistribution is low.

Note

After you have specified a distribution style for a column, Amazon Redshift handles data distribution at the cluster level. Amazon Redshift does not require or support the concept of partitioning data within database objects. You do not need to create table spaces or define partitioning schemes for tables.

Viewing distribution styles

To view the distribution style of a table, query the PG_CLASS system catalog table. The RELDISTSTYLE column indicates the distribution style for the table. The following table gives the distribution style for each value in PG_CLASS:
The following example creates three tables using the three distribution styles and then queries PG_CLASS to view the distribution styles.

```
cREATE TABLE alldiststyle (col1 INT) 
DISTSTYLE ALL;

CREATE TABLE evendiststyle (col1 INT) 
DISTSTYLE EVEN;

CREATE TABLE keydiststyle (col1 INT) 
DISTSTYLE KEY DISTKEY (col1);

SELECT relname, reldiststyle FROM pg_class 
WHERE relname like '%diststyle';
```

<table>
<thead>
<tr>
<th>relname</th>
<th>reldiststyle</th>
</tr>
</thead>
<tbody>
<tr>
<td>evendiststyle</td>
<td>0</td>
</tr>
<tr>
<td>keydiststyle</td>
<td>1</td>
</tr>
<tr>
<td>alldiststyle</td>
<td>8</td>
</tr>
</tbody>
</table>

**Evaluating query patterns**

Choosing distribution styles is just one aspect of database design. You should consider distribution styles only within the context of the entire system, balancing distribution with other important factors such as cluster size, compression encoding methods, sort keys, and table constraints.

Test your system with data that is as close to real data as possible.

In order to make good choices for distribution styles, you need to understand the query patterns for your Amazon Redshift application. Identify the most costly queries in your system and base your initial database design on the demands of those queries. Factors that determine the total cost of a query are how long the query takes to execute, how much computing resources it consumes, how often it is executed, and how disruptive it is to other queries and database operations.

Identify the tables that are used by the most costly queries, and evaluate their role in query execution. Consider how the tables are joined and aggregated.

Use the guidelines in this section to choose a distribution style for each table. When you have done so, create the tables, load them with data that is as close as possible to real data, and then test the tables for the types of queries that you expect to use. You can evaluate the query explain plans to identify tuning opportunities. Compare load times, storage space, and query execution times in order to balance your system's overall requirements.

**Designating distribution styles**

The considerations and recommendations for designating distribution styles in this section use a star schema as an example. Your database design might be based on a star schema, some variant of a star...
schema, or an entirely different schema. Amazon Redshift is designed to work effectively with whatever schema design you choose. The principles in this section can be applied to any design schema.

1. **Specify the primary key and foreign keys for all your tables.**

   Amazon Redshift does not enforce primary key and foreign key constraints, but the query optimizer uses them when it generates query plans. If you set primary keys and foreign keys, your application must maintain the validity of the keys.

2. **Distribute the fact table and its largest dimension table on their common columns.**

   Choose the largest dimension based on the size of data set that participates in the most common join, not just the size of the table. If a table is commonly filtered, using a WHERE clause, only a portion of its rows participate in the join. Such a table has less impact on redistribution than a smaller table that contributes more data. Designate both the dimension table's primary key and the fact table's corresponding foreign key as DISTKEY. If multiple tables use the same distribution key, they will also be collocated with the fact table. Your fact table can have only one distribution key. Any tables that join on another key will not be collocated with the fact table.

3. **Designate distribution keys for the other dimension tables.**

   Distribute the tables on their primary keys or their foreign keys, depending on how they most commonly join with other tables.

4. **Evaluate whether to change some of the dimension tables to use ALL distribution.**

   If a dimension table cannot be collocated with the fact table or other important joining tables, you can improve query performance significantly by distributing the entire table to all of the nodes. Using ALL distribution multiplies storage space requirements and increases load times and maintenance operations, so you should weigh all factors before choosing ALL distribution. The following section explains how to identify candidates for ALL distribution by evaluating the EXPLAIN plan.

5. **Use EVEN distribution for the remaining tables.**

   If a table is largely denormalized and does not participate in joins, or if you don't have a clear choice for another distribution style, use EVEN distribution (the default).

You cannot change the distribution style of a table after it is created. To use a different distribution style, you can recreate the table and populate the new table with a deep copy. For more information, see [Performing a deep copy](p. 156).

### Evaluating the query plan

You can use query plans to identify candidates for optimizing the distribution style.

After making your initial design decisions, create your tables, load them with data, and test them. Use a test data set that is as close as possible to the real data. Measure load times to use as a baseline for comparisons.

Evaluate queries that are representative of the most costly queries you expect to execute; specifically, queries that use joins and aggregations. Compare execution times for various design options. When you compare execution times, do not count the first time the query is executed, because the first run time includes the compilation time.

**DS_DISTNONE**

No redistribution is required, because corresponding slices are collocated on the compute nodes. You will typically have only one DS_DISTNONE step, the join between the fact table and one dimension table.

**DS_DISTALLNONE**

No redistribution is required, because the inner join table used DISTSTYLE ALL. The entire table is located on every node.
The inner table is redistributed.

A copy of the entire inner table is broadcast to all the compute nodes.

The entire inner table is redistributed to a single slice because the outer table uses DISTSTYLE ALL.

Both tables are redistributed.

DS_DIST_INNER means that the step will probably have a relatively high cost because the inner table is being redistributed to the nodes. DS_DIST_INNER indicates that the outer table is already properly distributed on the join key. Set the inner table's distribution key to the join key to convert this to DS_DIST_NONE. If distributing the inner table on the join key is not possible because the outer table is not distributed on the join key, evaluate whether to use ALL distribution for the inner table. If the table is relatively slow moving, that is, it is not updated frequently or extensively, and it is large enough to carry a high redistribution cost, change the distribution style to ALL and test again. ALL distribution causes increased load times, so when you retest, include the load time in your evaluation factors.

DS_DIST_ALL_INNER is not good. It means the entire inner table is redistributed to a single slice because the outer table uses DISTSTYLE ALL, so that a copy of the entire outer table is located on each node. This results in inefficient serial execution of the join on a single node instead taking advantage of parallel execution using all of the nodes. DISTSTYLE ALL is meant to be used only for the inner join table. Instead, specify a distribution key or use even distribution for the outer table.

DS_BCAST_INNER and DS_DIST_BOTH are not good. Usually these redistributions occur because the tables are not joined on their distribution keys. If the fact table does not already have a distribution key, specify the joining column as the distribution key for both tables. If the fact table already has a distribution key on another column, you should evaluate whether changing the distribution key to collocate this join will improve overall performance. If changing the distribution key of the outer table is not an optimal choice, you can achieve collocation by specifying DISTSTYLE ALL for the inner table.

The following example shows a portion of a query plan with DS_DIST_NONE and DS_DIST_INNER labels.

```
->  XN Hash Join DS_BCAST_INNER  (cost=112.50..3272334142.59 rows=170771 width=84)
   Hash Cond: ("outer".venueid = "inner".venueid)
   ->  XN Hash Join DS_BCAST_INNER  (cost=109.98..3167290276.71 rows=172456 width=47)
      Hash Cond: ("outer".eventid = "inner".eventid)
      ->  XN Merge Join DS_DIST_NONE  (cost=0.00..6286.47 rows=172456 width=30)
         Merge Cond: ("outer".listid = "inner".listid)
         ->  XN Seq Scan on listing  (cost=0.00..1924.97 rows=192497 width=14)
         ->  XN Seq Scan on sales  (cost=0.00..1724.56 rows=172456)
```

After changing the dimension tables to use DISTSTYLE ALL, the query plan for the same query shows DS_DIST_ALL_NONE in place of DS_BCAST_INNER. Also, there is a dramatic change in the relative cost for the join steps.
Query plan example

This example shows how to evaluate a query plan to find opportunities to optimize the distribution.

Run the following query with an EXPLAIN command to produce a query plan.

```sql
explain
select lastname, catname, venuename, venuecity, venuestate, eventname, month, sum(pricepaid) as buyercost, max(totalprice) as maxtotalprice
from category join event on category.catid = event.catid
join venue on venue.venueid = event.venueid
join sales on sales.eventid = event.eventid
join listing on sales.listid = listing.listid
join date on sales.dateid = date.dateid
join users on users.userid = sales.buyerid
group by lastname, catname, venuename, venuecity, venuestate, eventname, month
having sum(pricepaid)>9999
order by catname, buyercost desc;
```

In the TICKIT database, SALES is a fact table and LISTING is its largest dimension. In order to collocate the tables, SALES is distributed on the LISTID, which is the foreign key for LISTING, and LISTING is distributed on its primary key, LISTID. The following example shows the CREATE TABLE commands for SALES and LISTID.

```sql
create table sales(
salesid integer not null,
listid integer not null distkey,
sellerid integer not null,
buyerid integer not null,
eventid integer not null encode mostly16,
dateid smallint not null,
qtysold smallint not null encode mostly8,
pricepaid decimal(8,2) encode delta32k,
commission decimal(8,2) encode delta32k,
saletime timestamp,
primary key(salesid),
foreign key(listid) references listing(listid),
foreign key(sellerid) references users(userid),
foreign key(buyerid) references users(userid),
foreign key(dateid) references date(dateid))
    sortkey(listid,sellerid);
```
create table listing(
    listid integer not null distkey sortkey,
    sellerid integer not null,
    eventid integer not null encode mostly16,
    dateid smallint not null,
    numtickets smallint not null encode mostly8,
    priceperticket decimal(8,2) encode bytedict,
    totalprice decimal(8,2) encode mostly32,
    listtime timestamp,
    primary key(listid),
    foreign key(sellerid) references users(userid),
    foreign key(eventid) references event(eventid),
    foreign key(dateid) references date(dateid));

In the following query plan, the Merge Join step for the join on SALES and LISTING shows DS_DIST_NONE, which indicates that no redistribution is required for the step. However, moving up the query plan, the other inner joins show DS_BCAST_INNER, which indicates that the inner table is broadcast as part of the query execution. Because only one pair of tables can be collocated using key distribution, five tables need to be rebroadcast.

QUERY PLAN
XN Merge  (cost=1015345167117.54..1015345167544.46 rows=1000 width=103)
  Merge Key: category.catname, sum(sales.pricepaid)
  ->  XN Network  (cost=1015345167117.54..1015345167544.46 rows=170771 width=103)
      Send to leader
      ->  XN Sort  (cost=1015345167117.54..1015345167544.46 rows=170771 width=103)
         Sort Key: category.catname, sum(sales.pricepaid)
         ->  XN HashAggregate  (cost=15345150568.37..15345152276.08 rows=170771 width=103)
            Filter: (sum(pricepaid) > 9999.00)
            ->  XN Hash Join DS_BCAST_INNER
               (cost=742.08..15345146299.10 rows=170771 width=97)
                  Hash Cond: ("outer".catid = "inner".catid)
                  ->  XN Hash Join DS_BCAST_INNER
                      (cost=741.94..15342942456.61 rows=170771 width=90)
                          Hash Cond: ("outer".dateid = "inner".dateid)
                          ->  XN Hash Join DS_BCAST_INNER
                              (cost=737.38..15269938609.81 rows=170766 width=84)
                                  Hash Cond: ("outer".buyerid = "inner".userid)
                                  ->  XN Hash Join DS_BCAST_INNER
                                      (cost=112.50..3272334142.59 rows=170771 width=64)
                                          Hash Cond: ("outer".venueid = "inner".venueid)
                                          ->  XN Hash Join DS_BCAST_INNER
                                              (cost=109.98..3167290276.71 rows=172456 width=47)
                                                  Hash Cond: ("outer".eventid = "inner".eventid)
                                                  ->  XN Hash Join DS_BCAST_INNER
                                                      (cost=0.00..6286.47 rows=172456 width=30)
                                                          Hash Cond: ("outer".listid = "inner".listid)
                                                          ->  XN Merge Join DS_DIST_NONE
                                                              (cost=0.00..1924.97 rows=192497 width=14)
                                                              ->  XN Seq Scan on listing
One solution is to recreate the tables with DISTSTYLE ALL. You cannot change a table’s distribution style after it is created. To recreate tables with a different distribution style, use a deep copy.

First, rename the tables.

```
alter table users rename to userscopy;
alter table venue rename to venuecopy;
alter table category rename to categorycopy;
alter table date rename to datecopy;
alter table event rename to eventcopy;
```

Run the following script to recreate USERS, VENUE, CATEGORY, DATE, EVENT. Don’t make any changes to SALES and LISTING.

```
create table users(
    userid integer not null sortkey,
    username char(8),
    firstname varchar(30),
    lastname varchar(30),
    city varchar(30),
    state char(2),
    email varchar(100),
    phone char(14),
    likesports boolean,
    liketheatre boolean,
    likeconcerts boolean,
    likejazz boolean,
    likeclassical boolean,
    likeopera boolean,
    likerock boolean,
    likevegas boolean,
    likemusicals boolean,
    primary key(userid)) diststyle all;
```
create table venue(
    venueid smallint not null sortkey,
    venuename varchar(100),
    venuecity varchar(30),
    venuestate char(2),
    venueseats integer,
    primary key(venueid)) diststyle all;

create table category(
    catid smallint not null,
    catgroup varchar(10),
    catname varchar(10),
    catdesc varchar(50),
    primary key(catid)) diststyle all;

create table date(
    dateid smallint not null sortkey,
    caldate date not null,
    day character(3) not null,
    week smallint not null,
    month character(5) not null,
    qtr character(5) not null,
    year smallint not null,
    holiday boolean default('N'),
    primary key (dateid)) diststyle all;

create table event(
    eventid integer not null sortkey,
    venueid smallint not null,
    catid smallint not null,
    dateid smallint not null,
    eventname varchar(200),
    starttime timestamp,
    primary key(eventid),
    foreign key(venueid) references venue(venueid),
    foreign key(catid) references category(catid),
    foreign key(dateid) references date(dateid)) diststyle all;

Insert the data back into the tables and run an ANALYZE command to update the statistics.

insert into users select * from userscopy;
insert into venue select * from venuecopy;
insert into category select * from categorycopy;
insert into date select * from datecopy;
insert into event select * from eventcopy;
analyze;

Finally, drop the copies.

drop table userscopy;
drop table venuecopy;
drop table categorycopy;
drop table datecopy;
drop table eventcopy;
Run the same query with EXPLAIN again, and examine the new query plan. The joins now show DS_DIST_ALL_NONE, indicating that no redistribution is required because the data was distributed to every node using DISTSTYLE ALL.

```
QUERY PLAN
XN Merge  (cost=1000000047117.54..1000000047544.46 rows=1000 width=103)
 Merge Key: category.catname, sum(sales.pricepaid)
  ->  XN Network  (cost=1000000047117.54..1000000047544.46 rows=170771 width=103)
          Send to leader
            ->  XN Sort  (cost=1000000047117.54..1000000047544.46 rows=170771 width=103)
                  Sort Key: category.catname, sum(sales.pricepaid)
                    ->  XN HashAggregate  (cost=30568.37..32276.08 rows=170771 width=103)
                        Filter: (sum(pricepaid) > 9999.00)
                          ->  XN Hash Join DS_DIST_ALL_NONE  (cost=742.08..26299.10 rows=170771 width=103)
                              Hash Cond: ("outer".buyerid = "inner".userid)
                                ->  XN Hash Join DS_DIST_ALL_NONE  (cost=117.20..21831.99 rows=170766 width=97)
                                    Hash Cond: ("outer".dateid = "inner".dateid)
                                      ->  XN Hash Join DS_DIST_ALL_NONE  (cost=112.64..17985.08 rows=170771 width=90)
                                          Hash Cond: ("outer".catid = "inner".catid)
                                            ->  XN Hash Join DS_DIST_ALL_NONE  (cost=112.50..14142.59 rows=170771 width=84)
                                                Hash Cond: ("outer".venueid = "inner".venueid)
                                                  ->  XN Hash Join DS_DIST_ALL_NONE  (cost=109.98..10276.71 rows=172456 width=47)
                                                      Hash Cond: ("outer".eventid = "inner".eventid)
                                                        ->  XN Merge Join DS_DIST_NONE  (cost=0.00..6286.47 rows=172456 width=30)
                                                            Merge Cond: ("outer".listid = "inner".listid)
                                                              ->  XN Seq Scan on listing  (cost=0.00..1924.97 rows=192497 width=14)
                                                                ->  XN Seq Scan on sales  (cost=0.00..1724.56 rows=172456 width=24)
                                                                    ->  XN Hash  (cost=87.98..87.98 rows=8798 width=25)
                                                                        ->  XN Seq Scan on event  (cost=0.00..87.98 rows=8798 width=25)
                                                                            ->  XN Hash  (cost=2.02..2.02 rows=202 width=41)
                                                                                ->  XN Seq Scan on venue  (cost=0.00..2.02 rows=202 width=41)
                                                                                    ->  XN Hash  (cost=0.11..0.11 rows=11 width=10)
                                                                                        ->  XN Seq Scan on category  (cost=0.00..0.11 rows=11 width=10)
                                                                                            ->  XN Hash  (cost=3.65..3.65 rows=365 width=11)
                                                                                                ->  XN Seq Scan on date  (cost=0.00..3.65 rows=365 width=11)
```

Distribution examples

The following examples show how data is distributed according to the options that you define in the CREATE TABLE statement.

DISTKEY examples

Look at the schema of the USERS table in the TICKIT database. USERID is defined as the SORTKEY database and the DISTKEY column:

```sql
select "column", type, encoding, distkey, sortkey
from pg_table_def where tablename = 'users';
```

<table>
<thead>
<tr>
<th>column</th>
<th>type</th>
<th>encoding</th>
<th>distkey</th>
<th>sortkey</th>
</tr>
</thead>
<tbody>
<tr>
<td>userid</td>
<td>integer</td>
<td>none</td>
<td>t</td>
<td>1</td>
</tr>
<tr>
<td>username</td>
<td>character(8)</td>
<td>none</td>
<td>f</td>
<td>0</td>
</tr>
<tr>
<td>firstname</td>
<td>character varying(30)</td>
<td>text32k</td>
<td>f</td>
<td>0</td>
</tr>
</tbody>
</table>

USERID is a good choice for the distribution column on this table. If you query the SVV_DISKUSAGE system view, you can see that the table is very evenly distributed:

```sql
select slice, col, num_values, minvalue, maxvalue
from svv_diskusage
where name='users' and col =0
order by slice, col;
```

<table>
<thead>
<tr>
<th>slice</th>
<th>col</th>
<th>num_values</th>
<th>minvalue</th>
<th>maxvalue</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>12496</td>
<td>4</td>
<td>49987</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>12498</td>
<td>1</td>
<td>49988</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>12497</td>
<td>2</td>
<td>49989</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>12499</td>
<td>3</td>
<td>49990</td>
</tr>
</tbody>
</table>

The table contains 49,990 rows. The num_values columns shows the number of rows on each of the four slices. Each slice contains almost the same number of rows.

This example demonstrates distribution on a small test system. The total number of slices is typically much higher.

If you create a new table with the same data as the USERS table, but you set the DISTKEY to the STATE column, the distribution will not be as even. Slice 0 (13,587 rows) holds approximately 30% more rows than slice 3 (10,150 rows). In a much larger table, this amount of distribution skew could have an adverse impact on query processing.
create table userskey distkey(state) as select * from users;

select slice, col, num_values, minvalue, maxvalue from svv_diskusage
where name = 'newusers' and col=0 order by slice, col;

<table>
<thead>
<tr>
<th>slice</th>
<th>col</th>
<th>num_values</th>
<th>minvalue</th>
<th>maxvalue</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>13587</td>
<td>5</td>
<td>49989</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11245</td>
<td>2</td>
<td>49990</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>15008</td>
<td>1</td>
<td>49976</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>10150</td>
<td>4</td>
<td>49986</td>
</tr>
</tbody>
</table>

(4 rows)

**DISTSTYLE EVEN example**

If you create a new table with the same data as the USERS table but set the DISTSTYLE to EVEN, rows are always evenly distributed across slices.

create table userseven diststyle even as select * from users;

select slice, col, num_values, minvalue, maxvalue from svv_diskusage
where name = 'userseven' and col=0 order by slice, col;

<table>
<thead>
<tr>
<th>slice</th>
<th>col</th>
<th>num_values</th>
<th>minvalue</th>
<th>maxvalue</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>12497</td>
<td>4</td>
<td>49990</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>12498</td>
<td>8</td>
<td>49984</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>12498</td>
<td>2</td>
<td>49988</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>12497</td>
<td>1</td>
<td>49989</td>
</tr>
</tbody>
</table>

(4 rows)

However, because distribution is not based on a specific column, query processing can be degraded, especially if the table is joined to other tables. The lack of distribution on a joining column often influences the type of join operation that can be performed efficiently. Joins, aggregations, and grouping operations are optimized when both tables are distributed and sorted on their respective joining columns.

**DISTSTYLE ALL example**

If you create a new table with the same data as the USERS table but set the DISTSTYLE to ALL, all the rows are distributed to each slice.

select slice, col, num_values, minvalue, maxvalue from svv_diskusage
where name = 'usersall' and col=0 order by slice, col;

<table>
<thead>
<tr>
<th>slice</th>
<th>col</th>
<th>num_values</th>
<th>minvalue</th>
<th>maxvalue</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>49990</td>
<td>4</td>
<td>49990</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>49990</td>
<td>2</td>
<td>49990</td>
</tr>
</tbody>
</table>

(4 rows)
Choosing sort keys

Sorting data is a mechanism for optimizing query performance.

When you create a table, you can define one or more of its columns as sort keys. When data is initially loaded into the empty table, the values in the sort key columns are stored on disk in sorted order. Information about sort key columns is passed to the query planner, and the planner uses this information to construct plans that exploit the way that the data is sorted. For example, a merge join, which is often faster than a hash join, is feasible when the data is distributed and presorted on the joining columns. After adding a significant amount of new data and vacuuming to resort the data, run an ANALYZE command to update the statistical metadata for the query planner. For more information, see Analyzing tables (p. 158).

Another optimization that depends on sorted data is the efficient handling of range-restricted predicates. Amazon Redshift stores columnar data in 1 MB disk blocks. The min and max values for each block are stored as part of the metadata. If a range-restricted column is a sort key, the query processor is able to use the min and max values to rapidly skip over large numbers of blocks during table scans. For example, if a table stores five years of data sorted by date and a query specifies a date range of one month, up to 98% of the disk blocks can be eliminated from the scan. If the data is not sorted, more of the disk blocks (possibly all of them) have to be scanned. For more information about these optimizations, see Distribution styles (p. 102).

Sorted column data is also valuable for general query processing (GROUP BY and ORDER BY operations), window functions (PARTITION BY and ORDER BY operations), and as a means of optimizing compression. To understand the impact of the chosen sort key on query performance, use the EXPLAIN command. For more information, see Analyzing the Explain Plan (p. 184).

As new rows are incrementally loaded into tables, these new rows are sorted but they reside temporarily in a separate region on disk. In order to maintain a fully sorted table, you have to run the VACUUM command at regular intervals. For more information, see Vacuuming tables (p. 161).

Defining constraints

Uniqueness, primary key, and foreign key constraints are informational only; they are not enforced by Amazon Redshift. Nonetheless, primary keys and foreign keys are used as planning hints and they should be declared if your ETL process or some other process in your application enforces their integrity.

For example, the query planner uses primary and foreign keys in certain statistical computations, to infer uniqueness and referential relationships that affect subquery decorrelation techniques, to order large numbers of joins, and to eliminate redundant joins.

The planner leverages these key relationships, but it assumes that all keys in Amazon Redshift tables are valid as loaded. If your application allows invalid foreign keys or primary keys, some queries could return incorrect results. For example, a SELECT DISTINCT query might return duplicate rows if the primary key is not unique. Do not define key constraints for your tables if you doubt their validity. On the other hand, you should always declare primary and foreign keys and uniqueness constraints when you know that they are valid.

Amazon Redshift does enforce NOT NULL column constraints.

Analyzing table design

As you have seen in the previous sections, specifying sort keys, distribution keys, and column encodings can significantly improve storage, I/O, and query performance. This section provides a SQL script that you can run to help you identify tables where these options are missing or performing poorly.
Copy and paste the following code to create a SQL script named `table_inspector.sql`, then execute the script in your SQL client application as superuser. The script generates a temporary table named `TEMP_TABLES_REPORT`. The first few statements in the script drop tables that are created by the script, so you should remove those drop table statements or ignore them the first time you run the script.

```sql
DROP TABLE temp_staging_tables_1;
DROP TABLE temp_staging_tables_2;
DROP TABLE temp_tables_report;

CREATE TEMP TABLE temp_staging_tables_1
  (schemaname TEXT, 
   tablename TEXT, 
   tableid BIGINT, 
   size_in_megabytes BIGINT);

INSERT INTO temp_staging_tables_1
SELECT n.nspname, c.relname, c.oid, 
   (SELECT COUNT(*) FROM STV_BLOCKLIST b WHERE b.tbl = c.oid)
FROM pg_namespace n, pg_class c
WHERE n.oid = c.relnamespace 
AND nspname NOT IN ('pg_catalog', 'pg_toast', 'information_schema') 
AND c.relname <> 'temp_staging_tables_1';

CREATE TEMP TABLE temp_staging_tables_2
  (tableid BIGINT, 
   min_blocks_per_slice BIGINT, 
   max_blocks_per_slice BIGINT, 
   slice_count BIGINT);

INSERT INTO temp_staging_tables_2
SELECT tableid, MIN(c), MAX(c), COUNT(DISTINCT slice)
FROM (SELECT t.tableid, slice, COUNT(*) AS c
      FROM temp_staging_tables_1 t, STV_BLOCKLIST b
      WHERE t.tableid = b.tbl
      GROUP BY t.tableid, slice)
GROUP BY tableid;

CREATE TEMP TABLE temp_tables_report
  (schemaname TEXT, 
   tablename TEXT, 
   tableid BIGINT, 
   size_in_mb BIGINT, 
   has_dist_key INT, 
   has_sort_key INT, 
   has_col_encoding INT, 
   pct_skew_across_slices FLOAT, 
   pct_slices_populated FLOAT);

INSERT INTO temp_tables_report
SELECT t1.*,
   CASE WHEN EXISTS (SELECT *
      FROM pg_attribute a
      WHERE t1.tableid = a.attrelid 
      AND a.attnum > 0 
      AND NOT a.attisdropped 
      AND a.attisdistkey = 't')
   THEN 1 ELSE 0 END,
   CASE WHEN EXISTS (SELECT *
```
FROM pg_attribute a 
WHERE t1.tableid = a.attrelid 
AND a.attnum > 0 
AND NOT a.attisdropped 
AND a.attsortkeyord > 0) 

THEN 1 ELSE 0 END,
CASE WHEN EXISTS (SELECT * 
FROM pg_attribute a 
WHERE t1.tableid = a.attrelid 
AND a.attnum > 0 
AND NOT a.attisdropped 
AND a.attencodingtype <> 0) 

THEN 1 ELSE 0 END,
100 * CAST(t2.max_blocks_per_slice - t2.min_blocks_per_slice AS FLOAT) 
/ CASE WHEN (t2.min_blocks_per_slice = 0) 

CAST(100 * t2.slice_count AS FLOAT) / (SELECT COUNT(*) FROM STV_SLICES)
FROM temp_staging_tables_1 t1, temp_staging_tables_2 t2 
WHERE t1.tableid = t2.tableid;

SELECT * FROM temp_tables_report
ORDER BY schemaname, tablename;

The following sample shows the results of running the script with two sample tables, SKEW and SKEW2, that demonstrate the effects of data skew.

<table>
<thead>
<tr>
<th>schemaname</th>
<th>tablename</th>
<th>tableid</th>
<th>size_in_mb</th>
<th>has_dist_key</th>
<th>has_sort_key</th>
<th>has_encoding</th>
<th>pct_skew_across_slices</th>
<th>pct_slices_populated</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>category</td>
<td>100553</td>
<td>28</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>public</td>
<td>date</td>
<td>100555</td>
<td>44</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>public</td>
<td>event</td>
<td>100558</td>
<td>36</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>public</td>
<td>listing</td>
<td>100560</td>
<td>44</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>public</td>
<td>nation</td>
<td>100563</td>
<td>175</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>39.06</td>
</tr>
<tr>
<td>public</td>
<td>region</td>
<td>100566</td>
<td>30</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>7.81</td>
</tr>
<tr>
<td>public</td>
<td>sales</td>
<td>100562</td>
<td>52</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>public</td>
<td>skew</td>
<td>100547</td>
<td>18978</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>15.12</td>
<td>50</td>
</tr>
<tr>
<td>public</td>
<td>skew2</td>
<td>100548</td>
<td>353</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1.56</td>
</tr>
<tr>
<td>public</td>
<td>venue</td>
<td>100551</td>
<td>32</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>public</td>
<td>users</td>
<td>100549</td>
<td>82</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>public</td>
<td>venue</td>
<td>100551</td>
<td>32</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>100</td>
</tr>
</tbody>
</table>

The following list describes the TEMP_TABLES_REPORT columns:

**has_dist_key**
Indicates whether the table has distribution key. 1 indicates a key exists; 0 indicates there is no key. For example, nation does not have a distribution key.

**has_sort_key**
Indicates whether the table has a sort key. 1 indicates a key exists; 0 indicates there is no key. For example, nation does not have a sort key.
has_column_encoding
Indicates whether the table has any compression encodings defined for any of the columns. 1 indicates at least one column has an encoding. 0 indicates there is no encoding. For example, region has no compression encoding.

pct_skew_across_slices
The percentage of data distribution skew. A smaller value is good.

pct_slices_populated
The percentage of slices populated. A larger value is good.

Tables for which there is significant data distribution skew will have either a large value in the pct_skew_across_slices column or a small value in the pct_slices_populated column. This indicates that you have not chosen an appropriate distribution key column. In the example above, the SKEW table has a 15.12% skew across slices, but that's not necessarily a problem. What's more significant is the 1.56% value for the slices populated for the SKEW2 table. The small value is an indication that the SKEW2 table has the wrong distribution key.

Run the table_inspector.sql script whenever you add new tables to your database or whenever you have significantly modified your tables.
Loading Data

Topics
- Using a COPY command to load data (p. 117)
- Updating tables with DML commands (p. 150)
- Updating and inserting new data (p. 150)
- Performing a deep copy (p. 156)
- Analyzing tables (p. 158)
- Vacuuming tables (p. 161)
- Managing concurrent write operations (p. 169)

You can bulk load data into your tables either from flat files that are stored in an Amazon S3 bucket or from an Amazon DynamoDB table.

A COPY command is the most efficient way to load a table. When you load data from Amazon S3, the COPY command is able to read from multiple data files simultaneously. Whether you load from data files on Amazon S3 or from a DynamoDB table, Amazon Redshift distributes the workload to the cluster nodes and performs the load process in parallel.

When you use the COPY command to load data, you can limit the access users have to your load data by using temporary security credentials.

You can also add data to your tables using INSERT commands, though it is much less efficient than using COPY.

After your initial data load, if you add, modify, or delete a significant amount of data, you should follow up by running a VACUUM command to reorganize your data and reclaim space after deletes. You should also run an ANALYZE command to update table statistics.

This section explains how to load data and troubleshoot data loads and presents best practices for loading data.

Using a COPY command to load data

Topics
- Preparing your input data (p. 118)
- Loading data from Amazon S3 (p. 119)
Preparing your input data

The COPY command leverages the Amazon Redshift massively parallel processing (MPP) architecture to read and load data in parallel from files on Amazon S3, from a DynamoDB table, or from text output from one or more remote hosts.

**Note**
We strongly recommend using the COPY command to load large amounts of data. Using individual INSERT statements to populate a table might be prohibitively slow. Alternatively, if your data already exists in other Amazon Redshift database tables, use INSERT INTO ... SELECT or CREATE TABLE AS to improve performance. For information, see INSERT (p. 349) or CREATE TABLE AS (p. 320).

To grant or revoke privilege to load data into a table using a COPY command, grant or revoke the INSERT privilege.

Your data needs to be in the proper format for loading into your Amazon Redshift table. This section presents guidelines for preparing and verifying your data before the load and for validating a COPY statement before you execute it.

To protect the information in your files, you can encrypt the data files before you upload them to your Amazon S3 bucket; COPY will decrypt the data as it performs the load. You can also limit access to your load data by providing temporary security credentials to users. Temporary security credentials provide enhanced security because they have short life spans and cannot be reused after they expire.

You can compress the files using gzip or lzop to save time uploading the files. COPY is then able to speed up the load process by uncompressing the files as they are read.

To help keep your data secure in transit within the AWS cloud, Amazon Redshift uses hardware accelerated SSL to communicate with Amazon S3 or Amazon DynamoDB for COPY, UNLOAD, backup, and restore operations.

When you load your table directly from an Amazon DynamoDB table, you have the option to control the amount of Amazon DynamoDB provisioned throughput you consume.

You can optionally let COPY analyze your input data and automatically apply optimal compression encodings to your table as part of the load process.

**Preparing your input data**

If your input data is not compatible with the table columns that will receive it, the COPY command will fail.

Use the following guidelines to help ensure that your input data is valid:

- Your data can only contain UTF-8 characters up to four bytes long.
• Verify that CHAR and VARCHAR strings are no longer than the lengths of the corresponding columns.
  VARCHAR strings are measured in bytes, not characters, so, for example, a four-character string of
  Chinese characters that occupy four bytes each requires a VARCHAR(16) column.
• Multibyte characters can only be used with VARCHAR columns. Verify that multibyte characters are
  no more than four bytes long.
• Verify that data for CHAR columns only contains single-byte characters.
• Do not include any special characters or syntax to indicate the last field in a record. This field can be
  a delimiter.
• If your data includes null terminators, also referred to as NUL (UTF-8 0000) or binary zero (0x000),
  you can load these characters as NULLS into CHAR or VARCHAR columns by using the NULL AS
  option in the COPY command: null as '\0' or null as '\000'. If you do not use NULL AS, null
  terminators will cause your COPY to fail.
• If your strings contain special characters, such as delimiters and embedded newlines, use the ESCAPE
  option with the COPY (p. 276) command.
• Verify that all single and double quotes are appropriately matched.
• Verify that floating-point strings are in either standard floating-point format, such as 12.123, or an
  exponential format, such as 1.0E4.
• Verify that all timestamp and date strings follow the specifications for DATEFORMAT and TIMEFORMAT
  strings (p. 297). The default timestamp format is YYYY-MM-DD hh:mm:ss, and the default date format
  is YYYY-MM-DD.
• For more information about boundaries and limitations on individual data types, see Data types (p. 214).
  For information about multi-byte character errors, see Multi-byte character load errors (p. 148)

Loading data from Amazon S3

Topics
• Splitting your data into multiple files (p. 120)
• Uploading files to Amazon S3 (p. 120)
• Using the COPY command to load from Amazon S3 (p. 123)

The COPY command leverages the Amazon Redshift massively parallel processing (MPP) architecture
to read and load data in parallel from files in an Amazon S3 bucket. You can take maximum advantage
of parallel processing by splitting your data into multiple files and by setting distribution keys on your
tables. For more information about distribution keys, see Choosing a data distribution style (p. 101).

Data from the files is loaded into the target table, one line per row. The fields in the data file are matched
to table columns in order, left to right. Fields in the data files can be fixed-width or character delimited;
the default delimiter is a pipe (|). By default, all the table columns are loaded, but you can optionally define
a comma-separated list of columns. If a table column is not included in the column list specified in the
COPY command, it is loaded with a default value. For more information, see Loading default column
values (p. 145).

Follow this general process to load data from Amazon S3:
1. Split your data into multiple files.
2. Upload your files to Amazon S3.
3. Run a COPY command to load the table.
4. Verify that the data was loaded correctly.

The rest of this section explains these steps in detail.
Splitting your data into multiple files

You can load table data from a single file, or you can split the data for each table into multiple files. The COPY command can load data from multiple files in parallel. You can load multiple files by specifying a common prefix, or prefix key, for the set, or by explicitly listing the files in a manifest file.

Note
We strongly recommend that you divide your data into multiple files to take advantage of parallel processing.

Split your data into files so that the number of files is a multiple of the number of slices in your cluster. That way Amazon Redshift can divide the data evenly among the slices. The number of slices is equal to the number of processor cores on the node. For example, each DW1.XL compute node has two slices, and each DW2.8XL compute node has 32 slices. The nodes all participate in parallel query execution, working on data that is distributed as evenly as possible across the slices. If you have a cluster with two DW1.XL nodes, you might split your data into four files or some multiple of four. Amazon Redshift does not take file size into account when dividing the workload, so you need to ensure that the files are roughly the same size.

If you intend to use object prefixes to identify the load files, name each file with a common prefix. For example, the venue.txt file might be split into four files, as follows:

<table>
<thead>
<tr>
<th>File Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>venue.txt.1</td>
</tr>
<tr>
<td>venue.txt.2</td>
</tr>
<tr>
<td>venue.txt.3</td>
</tr>
<tr>
<td>venue.txt.4</td>
</tr>
</tbody>
</table>

If you put multiple files in a folder in your bucket, you can specify the folder name as the prefix and COPY will load all of the files in the folder. If you explicitly list the files to be loaded by using a manifest file, the files can reside in different buckets or folders.

Uploading files to Amazon S3

Topics
- Managing data consistency (p. 121)
- Uploading encrypted data to Amazon S3 (p. 122)
- Verifying that the correct files are present in your bucket (p. 123)

After splitting your files, you can upload them to your bucket. You can optionally compress or encrypt the files before you load them.

Create an Amazon S3 bucket to hold your data files, and then upload the data files to the bucket. For information about creating buckets and uploading files, see Working with Amazon S3 Buckets in the Amazon Simple Storage Service Developer Guide.

Amazon S3 provides eventual consistency for some operations, so it is possible that new data will not be available immediately after the upload. For more information see, Managing data consistency (p. 121)

Important
The Amazon S3 bucket that holds the data files must be created in the same region as your cluster unless you use the REGION (p. 285) option to specify the region in which the Amazon S3 bucket is located.

You can create an Amazon S3 bucket in a specific region either by selecting the region when you create the bucket by using the Amazon S3 console, or by specifying an endpoint when you create the bucket using the Amazon S3 API or CLI.
Following the data load, verify that the correct files are present on Amazon S3.

**Managing data consistency**

Amazon S3 provides eventual consistency for some operations, so it is possible that new data will not be available immediately after the upload, which could result in an incomplete data load or loading stale data. All COPY operations from buckets in the US Standard Region are eventually consistent. COPY operations where the cluster and the bucket are in different regions are also eventually consistent. All other regions provide read-after-write consistency for uploads of new objects with unique object keys. For more information about data consistency, see Amazon S3 Data Consistency Model in the Amazon Simple Storage Service Developer Guide.

To ensure that your application loads the correct data, we recommend the following practices:

- **Create new object keys.**

  Amazon S3 provides eventual consistency in all regions for overwrite operations. Creating new file names, or object keys, in Amazon S3 for each data load operation provides strong consistency in all regions except US Standard.

- **Use a manifest file with your COPY operation.**

  The manifest explicitly names the files to be loaded. Using a manifest file enforces strong consistency, so it is especially important for buckets in the US Standard region, but it is a good practice in all regions.

- **Use a named endpoint.**

  If your cluster is in the US East (N. Virginia) Region, you can improve data consistency and reduce latency by using a named endpoint when you create your Amazon S3 bucket.

The rest of this section explains these steps in detail.

**Creating new object keys**

Because of potential data consistency issues, we strongly recommend creating new files with unique Amazon S3 object keys for each data load operation. If you overwrite existing files with new data, and then issue a COPY command immediately following the upload, it is possible for the COPY operation to begin loading from the old files before all of the new data is available. For more information about eventual consistency, see Amazon S3 Data Consistency Model in the Amazon S3 Developer Guide.

**Using a manifest file**

You can explicitly specify which files to load by using a manifest file. When you use a manifest file, COPY enforces strong consistency by searching secondary servers if it does not find a listed file on the primary server. The manifest file can be configured with an optional mandatory flag. If mandatory is true and the file is not found, COPY returns an error.

For more information about using a manifest file, see the MANIFEST (p. 278) option for the COPY command and Using a manifest to specify data files (p. 299) in the COPY examples.

Using a manifest file addresses eventual consistency when loading from new data files in the US Standard Region. However, because Amazon S3 provides eventual consistency for overwrites in all regions, it is possible to load stale data if you overwrite existing objects with new data. As a best practice, never overwrite existing files with new data.

**Using a named endpoint**

If you created your cluster in the US East (N. Virginia) Region and you intend to use an object prefix to identify your load files, do not specify the US Standard Region when you create your bucket. Amazon S3 provides eventual consistency for all operations in the US Standard Region. If your cluster is in the US
East (N. Virginia) Region, you can improve data consistency by using a named endpoint when you create your Amazon S3 bucket. Amazon S3 automatically routes requests that specify the US Standard Region to facilities in Northern Virginia or the Pacific Northwest, so if you create a cluster in the US East Region and specify the US Standard Region when you create your Amazon S3 bucket, it is possible that your data will not be located near your cluster. If you use a named endpoint to specify the US East Region when you create your Amazon S3 bucket, AWS will route your data to Northern Virginia whenever possible. Using a named endpoint can also reduce data latency.

If you use a manifest to specify your load files, the COPY command enforces consistency for the listed files. For more information, see Using a manifest file (p. 121).

You do not need to use a named endpoint for any other regions.

To use a named endpoint for the US East region, replace s3.amazonaws.com with s3-external-1.amazonaws.com in the hostname substring that you are using to access Amazon S3. For example, replace the following string:

http://mybucket.s3.amazonaws.com/somekey.ext

with this:

http://mybucket.s3-external-1.amazonaws.com/somekey.ext

You can use named endpoints only with the Amazon S3 API or CLI. You cannot use named endpoints with the Amazon S3 console.

For more information about Amazon S3 regions, see Buckets and Regions in the Amazon Simple Storage Service Developer Guide.

**Uploading encrypted data to Amazon S3**

Amazon S3 supports both server-side encryption and client-side encryption. This topic discusses the differences between the server-side and client-side encryption and describes the steps to use client-side encryption with Amazon Redshift. Server-side encryption is transparent to Amazon Redshift.

**Server-side encryption**

Server-side encryption is data encryption at rest, that is, Amazon S3 encrypts your data as it uploads it and decrypts it for you when you access it. When you load tables using a COPY command, there is no difference in the way you load from server-side encrypted or unencrypted objects on Amazon S3. For more information about server-side encryption, see Using Server-Side Encryption in the Amazon Simple Storage Service Developer Guide.

**Client-side encryption**

In client-side encryption, your client application manages encryption of your data, the encryption keys, and related tools. You can upload data to an Amazon S3 bucket using client-side encryption, and then load the data using the COPY command with the ENCRYPTED option and a private encryption key to provide greater security.

You encrypt your data using envelope encryption. With envelope encryption, your application handles all encryption exclusively. Your private encryption keys and your unencrypted data are never sent to AWS, so it's very important that you safely manage your encryption keys. If you lose your encryption keys, you won't be able to unencrypt your data, and you can't recover your encryption keys from AWS. Envelope encryption combines the performance of fast symmetric encryption while maintaining the greater security that key management with asymmetric keys provides. A one-time-use symmetric key (the envelope symmetric key) is generated by your Amazon S3 encryption client to encrypt your data, then that key is...
encrypted by your master key and stored alongside your data in Amazon S3. When Amazon Redshift accesses your data during a load, the encrypted symmetric key is retrieved and decrypted with your real key, then the data is decrypted.

To work with Amazon S3 client-side encrypted data in Amazon Redshift, follow the steps outlined in Using Client-Side Encryption in the Amazon S3 Developer Guide with the additional requirements that you use:

- **Symmetric encryption** — The AWS SDK for Java AmazonS3EncryptionClient class uses a process called *envelope encryption* which is based on symmetric key encryption. Use this class to create an Amazon S3 client to upload client-side encrypted data.

- **256-bit AES master symmetric key** — A master key encrypts the envelope key. You pass the master key to your instance of the AmazonS3EncryptionClient class. The master key must use the 256-AES encryption standard. Save this key because you will need it to copy data into Amazon Redshift.

- **Object metadata to store encrypted envelope key** — By default, AmazonS3 stores the envelope key as object metadata for the AmazonS3EncryptionClient class. The encrypted envelope key that is stored as object metadata is used during the decryption process.

For information about loading client-side encrypted files into your Amazon Redshift tables using the COPY command, see Loading encrypted data files from Amazon S3 (p. 127).

**Verifying that the correct files are present in your bucket**

After you upload your files to your Amazon S3 bucket, we recommend listing the contents of the bucket to verify that all of the correct files are present and that no unwanted files are present. For example, if the bucket `mybucket` holds a file named `venue.txt.back`, that file will be loaded, perhaps unintentionally, by the following command:

```sql
COPY venue FROM 's3://mybucket/venue' ... ;
```

If you want to control specifically which files are loaded, you can use a manifest file to explicitly list the data files. For more information about using a manifest file, see the MANIFEST (p. 278) option for the COPY command and Using a manifest to specify data files (p. 299) in the COPY examples.

For more information about listing the contents of the bucket, see Listing Object Keys in the Amazon S3 Developer Guide.

**Using the COPY command to load from Amazon S3**

**Topics**

- Using a manifest to specify data files (p. 125)
- Loading gzip or lzop compressed data files from Amazon S3 (p. 126)
- Loading fixed-width data from Amazon S3 (p. 126)
- Loading multibyte data from Amazon S3 (p. 127)
- Loading encrypted data files from Amazon S3 (p. 127)

Use the `COPY` command to load a table in parallel from data files on Amazon S3. You can specify the files to be loaded by using an Amazon S3 object prefix or by using a manifest file.

The syntax to specify the files to be loaded by using a prefix is as follows:

```sql
COPY <table_name> FROM 's3://<bucket_name>/<object_prefix>'
```
credentials 'aws_access_key_id=<access-key-id>;aws_secret_access_key=<secret-access-key>'; 

The manifest file is a JSON-formatted file that lists the data files to be loaded. The syntax to specify the files to be loaded by using a manifest file is as follows:

```sql
copy <table_name> from 's3://<bucket_name>/<manifest_file>'
credentials 'aws_access_key_id=<access-key-id>;aws_secret_access_key=<secret-access-key>'
manifest;
```

**Note**
Do not include line breaks or spaces in your `aws_access_credentials` string.

The table to be loaded must already exist in the database. For information about creating a table, see `CREATE TABLE` (p. 310) in the SQL Reference.

The values for `<access-key-id>` and `<secret-access-key>` are the AWS credentials needed to access the Amazon S3 objects. The credentials must correspond to an AWS IAM user that has permission to GET and LIST the Amazon S3 objects that are being loaded. For more information about Amazon S3 IAM users, see Access Control in Amazon Simple Storage Service Developer Guide.

You can limit the access users have to your load data by using temporary security credentials. Temporary security credentials provide enhanced security because they have short life spans and cannot be reused after they expire. A user who has these temporary security credentials can access your resources only until the credentials expire. For more information, see Temporary security credentials (p. 290). To load data using temporary access credentials, use the following syntax:

```sql
copy <table_name> from 's3://<bucket_name>/<object_prefix>'
credentials 'aws_access_key_id=<temporary-access-key-id>;aws_secret_access_key=<temporary-secret-access-key>;token=<temporary-token>';
```

**Important**
The temporary security credentials must be valid for the entire duration of the COPY statement. If the temporary security credentials expire during the load process, the COPY will fail and the transaction will be rolled back. For example, if temporary security credentials expire after 15 minutes and the COPY requires one hour, the COPY will fail before it completes.

If you want to validate your data without actually loading the table, use the NOLOAD option with the COPY (p. 276) command.

The following example shows the first few rows of a pipe-delimited data in a file named `venue.txt`.

```
1|Toyota Park|Bridgeview|IL|0  
2|Columbus Crew Stadium|Columbus|OH|0  
3|RFK Stadium|Washington|DC|0
```

Before uploading the file to Amazon S3, split the file into multiple files so that the COPY command can load it using parallel processing. For more information, see Splitting your data into multiple files (p. 120).

For example, the `venue.txt` file might be split into four files, as follows:
The following COPY command loads the VENUE table using the pipe-delimited data in the data files with the prefix 'venue' in the Amazon S3 bucket mybucket.

**Note**
The Amazon S3 bucket mybucket in the following examples does not exist. For sample COPY commands that use real data in an existing Amazon S3 bucket, see Step 4: Load sample data (p. 18).

```sql
copy venue from 's3://mybucket/venue'
credentials 'aws_access_key_id=<access-key-id>;aws_secret_access_key=<secret-access-key>'
delimiter '|';
```

If no Amazon S3 objects with the key prefix 'venue' exist, the load fails.

**Using a manifest to specify data files**

You can use a manifest to ensure that the COPY command loads all of the required files, and only the required files, for a data load. Instead of supplying an object path for the COPY command, you supply the name of a JSON-formatted text file that explicitly lists the files to be loaded. The URL in the manifest must specify the bucket name and full object path for the file, not just a prefix. You can use a manifest to load files from different buckets or files that do not share the same prefix. The following example shows the JSON to load files from different buckets and with file names that begin with date stamps.

```json
{
   "entries": [
      {"url":"s3://mybucket-alpha/2013-10-04-custdata", "mandatory":true},
      {"url":"s3://mybucket-alpha/2013-10-05-custdata", "mandatory":true},
      {"url":"s3://mybucket-beta/2013-10-04-custdata", "mandatory":true},
      {"url":"s3://mybucket-beta/2013-10-05-custdata", "mandatory":true}
   ]
}
```

The optional `mandatory` flag specifies whether COPY should return an error if the file is not found. The default of `mandatory` is `false`.

The following example runs the COPY command with the manifest in the previous example, which is named `cust.manifest`.

```sql
copy customer
from 's3://mybucket/cust.manifest'
credentials 'aws_access_key_id=<access-key-id>;aws_secret_access_key=<secret-access-key>'
manifest;
```

For more information, see Using a manifest to specify data files (p. 299).

If your Amazon S3 bucket is in the US Standard Region, you can avoid eventual consistency issues by using a manifest. For more information, see Managing data consistency (p. 121).
Loading gzip or lzop compressed data files from Amazon S3

To load data files that are compressed using gzip, include the GZIP option. To load data files that are compressed using lzop, include the LZOP option.

```sql
COPY customer FROM 's3://mybucket/customer.lzo'
CREDENTIALS 'aws_access_key_id=<access-key-id>;aws_secret_access_key=<secret-access-key>'
DELIMITER '|' LZOP;
```

If you use the GZIP option or LZOP option, ensure that all data files being loaded are gzip or lzop compressed. COPY does not support files compressed using the lzop --filter option.

Loading fixed-width data from Amazon S3

Fixed-width data files have uniform lengths for each column of data. Each field in a fixed-width data file has exactly the same length and position. For character data (CHAR and VARCHAR) in a fixed-width data file, you must include leading or trailing spaces as placeholders in order to keep the width uniform. For integers, you must use leading zeros as placeholders. A fixed-width data file has no delimiter to separate columns.

To load a fixed-width data file into an existing table, USE the FIXEDWIDTH parameter in the COPY command. Your table specifications must match the value of fixedwidth_spec in order for the data to load correctly.

To load fixed-width data from a file to a table, issue the following command:

```sql
COPY table_name FROM 's3://mybucket/prefix'
CREDENTIALS 'aws_access_key_id=<access-key-id>;aws_secret_access_key=<secret-access-key>'
FIXEDWIDTH 'fixedwidth_spec';
```

The `fixedwidth_spec` parameter is a string that contains an identifier for each column and the width of each column, separated by a colon. The `column:width` pairs are delimited by commas. The identifier can be anything that you choose: numbers, letters, or a combination of the two. The identifier has no relation to the table itself, so the specification must contain the columns in the same order as the table.

The following two examples show the same specification, with the first using numeric identifiers and the second using string identifiers:

'0:3,1:25,2:12,3:2,4:6'

'venueid:3,venuename:25,venuecity:12,venuestate:2,venueseats:6'

The following example shows fixed-width sample data that could be loaded into the VENUE table using the above specifications:

<table>
<thead>
<tr>
<th></th>
<th>Toyota Park</th>
<th>Bridgeview</th>
<th>IL0</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Columbus Crew Stadium</td>
<td>Columbus</td>
<td>OH0</td>
</tr>
<tr>
<td>3</td>
<td>RFK Stadium</td>
<td>Washington</td>
<td>DC0</td>
</tr>
<tr>
<td>4</td>
<td>CommunityAmerica Ballpark</td>
<td>Kansas City</td>
<td>KS0</td>
</tr>
<tr>
<td>5</td>
<td>Gillette Stadium</td>
<td>Foxborough</td>
<td>MA68756</td>
</tr>
</tbody>
</table>
The following COPY command loads this data set into the VENUE table:

```
COPY venue
FROM 's3://mybucket/data/venue_fw.txt'
CREDENTIALS 'aws_access_key_id=<access-key-id>;aws_secret_access_key=<secret-access-key>'
FIXEDWIDTH 'venueid:3,venuename:25,venuecity:12,venuestate:2,venueseats:6';
```

**Loading multibyte data from Amazon S3**

If your data includes non-ASCII multibyte characters (such as Chinese or Cyrillic characters), you must load the data to VARCHAR columns. The VARCHAR data type supports four-byte UTF-8 characters, but the CHAR data type only accepts single-byte ASCII characters. You cannot load five-byte or longer characters into Amazon Redshift tables. For more information about CHAR and VARCHAR, see Data types (p. 214).

To check which encoding an input file uses, use the Linux `file` command:

```
$ file ordersdata.txt
ordersdata.txt: ASCII English text
$ file uni_ordersdata.dat
uni_ordersdata.dat: UTF-8 Unicode text
```

**Loading encrypted data files from Amazon S3**

You can use the COPY command to load data files that were uploaded to Amazon S3 using client-side encryption.

The files are encrypted using a base64 encoded AES 256 bit key. You will need to provide that key value when loading the encrypted files. See Uploading encrypted data to Amazon S3 (p. 122).

To load encrypted data files, add the master key value to the credentials string and include the ENCRYPTED option.

```
copy customer from 's3://mybucket/encrypted/customer'
cREDENTIALS 'aws_access_key_id=<access-key-id>;aws_secret_access_key=<secret-access-key>;master_symmetric_key=<master_key>'
delimiter '|' ENCRYPTED;
```

To load encrypted data files that are gzip compressed, include the GZIP option along with the master key value and the ENCRYPTED option.

```
copy customer from 's3://mybucket/encrypted/customer'
cREDENTIALS 'aws_access_key_id=<access-key-id>;aws_secret_access_key=<secret-access-key>;master_symmetric_key=<master_key>'
delimiter '|' ENCRYPTED GZIP;
```

**Loading data from Amazon EMR**

You can use the COPY command to load data in parallel from an Amazon EMR cluster configured to write text files to the cluster’s Hadoop Distributed File System (HDFS) in the form of fixed-width files, character-delimited files, CSV files, or JSON-formatted files.
Amazon EMR provides a bootstrap action for output to Amazon Redshift that performs much of the preparation work for you. The bootstrap action must be specified when the Amazon EMR cluster is created. The Amazon Redshift bootstrap action is not available for clusters created using the following AMI versions: 2.1.4, 2.2.4, 2.3.6.

You will follow different procedures to load data from an Amazon EMR cluster, depending on whether or not you choose to use the Amazon Redshift bootstrap action. Follow the steps in one of the following sections.

- Loading data from Amazon EMR using the Amazon Redshift bootstrap action (p. 128)
- Loading data from Amazon EMR without the Amazon Redshift bootstrap action (p. 130)

## Loading data from Amazon EMR using the Amazon Redshift bootstrap action

When you create an Amazon EMR cluster, you can avoid several steps by using the Amazon Redshift bootstrap action. The bootstrap action configures the Amazon EC2 host instances to accept commands from the Amazon Redshift cluster nodes. The Amazon Redshift bootstrap action is not available for Amazon EMR clusters created using the following AMI versions: 2.1.4, 2.2.4, 2.3.6. If you choose not to use the bootstrap action, follow the steps in Loading data from Amazon EMR without the Amazon Redshift bootstrap action (p. 130).

### Loading data process

This section walks you through the process of loading data from an Amazon EMR cluster. The following sections provide the details you need to accomplish each step.

- **Step 1: Configure IAM permissions (p. 128)**
  
  The users that create the Amazon EMR cluster and run the Amazon Redshift COPY command must have the necessary permissions.

- **Step 2: Create an Amazon EMR cluster (p. 129)**

  When you create the Amazon EMR cluster, specify the Amazon Redshift bootstrap action to avoid configuring the Amazon EMR host instances to allow ingress from Amazon Redshift. Configure the cluster to output text files to the Hadoop Distributed File System (HDFS).

- **Step 3: Run the COPY command to load the data (p. 130)**

  From an Amazon Redshift database, run the COPY command to load the data into an Amazon Redshift table.

### Step 1: Configure IAM permissions

The users that create the Amazon EMR cluster and run the Amazon Redshift COPY command must have the necessary permissions.

#### To configure IAM permissions

1. Add the following permissions for the IAM user that will create the Amazon EMR cluster.

   ```
   ec2:DescribeSecurityGroups
   ec2:RevokeSecurityGroupIngress
   ec2:AuthorizeSecurityGroupIngress
   redshift:DescribeClusters
   ```
2. Add the following permission for the IAM user that will execute the COPY command.

```
elasticmapreduce:ListInstances
```

Step 2: Create an Amazon EMR cluster

The COPY command loads data from files on the Amazon EMR Hadoop Distributed File System (HDFS). When you create the Amazon EMR cluster, configure the cluster to output data files to the cluster's HDFS.

To create an Amazon EMR cluster

1. Create an Amazon EMR cluster in the same AWS region as the Amazon Redshift cluster.

   If the Amazon Redshift cluster is in a VPC, the Amazon EMR cluster must be in the same VPC group. If the Amazon Redshift cluster uses EC2-Classic mode (that is, it is not in a VPC), the Amazon EMR cluster must also use EC2-Classic mode. For more information, see Managing Clusters in Virtual Private Cloud (VPC) in the Amazon Redshift Management Guide.

2. Configure the cluster to output data files to the cluster's HDFS.

   **Important**
   
   The HDFS file names must not include asterisks (*) or question marks (?).

3. Add the Amazon Redshift bootstrap action.

   The Amazon Redshift bootstrap action adds security ingress rules to the Amazon EMR host instances to permit the Amazon Redshift cluster to access the instances. The rules are added to the `elasticmapreduce-master` and `elasticmapreduce-slave` security groups. To view details about the added security rules, see stderr in the bootstrap action log files. For more information, see View Log Files.

   The bootstrap action does not add security rules if they exist already as a result of a previous bootstrap action. Amazon EMR does not clean up rules automatically. You will need to remove them manually, if necessary. For more information, see Deleting Rules from a Security Group.

   Amazon EC2 enforces a limit of rules to 50 rules per security group for VPC or 100 rules per security group for EC2-Classic. If the security rules added by the bootstrap action exceed the limit, the cluster will fail. You can either remove rules as needed or request a limit increase. For more information, see AWS Service Limits.

   To add the Amazon Redshift bootstrap action:

   1. Follow the steps in either of the following topics:
      
      - Running Custom Bootstrap Actions from the CLI.
      - Running Custom Bootstrap Actions from the Amazon EMR Console.

   2. For the Amazon Redshift bootstrap action, specify the following Amazon S3 location.

      ```
s3://beta.elasticmapreduce/bootstrap-actions/redshift/setup-redshift-authorization
```

   3. Specify your Amazon Redshift cluster name in the arguments.

   For more information, see Create Bootstrap Actions to Install Additional Software.
4. Specify **No** for the **Auto-terminate** option in the Amazon EMR cluster configuration so that the cluster remains available while the COPY command executes.

**Important**
If any of the data files are changed or deleted before the COPY completes, you might have unexpected results, or the COPY operation might fail.

5. Note the Amazon EMR cluster ID. You will use the cluster ID in the COPY command.

**Step 3: Run the COPY command to load the data**

Run a COPY (p. 276) command to connect to the Amazon EMR cluster and load the data into an Amazon Redshift table. The Amazon EMR cluster must continue running until the COPY command completes. For example, do not configure the cluster to auto-terminate.

In the COPY command, specify the Amazon EMR cluster ID and the HDFS file path and file name.

```
copy sales
from 'emr:// j-1H7OUO3B52HI5/myoutput/part*' credentials
'aws_access_key_id=<access-key-id>;aws_secret_access_key=<secret-access-key>'';
```

You can use the wildcard characters asterisk (*) and question mark (?) as part of the file name argument. For example, `part*` loads the files `part-0000, part-0001`, and so on. If you specify only a folder name, COPY attempts to load all files in the folder.

**Important**
If you use wildcard characters or use only the folder name, verify that no unwanted files will be loaded or the COPY command will fail. For example, some processes might write a log file to the output folder.

**Loading data from Amazon EMR without the Amazon Redshift bootstrap action**

The Amazon Redshift bootstrap action is not available for Amazon EMR clusters created using AMI versions 2.1.4, 2.2.4, and 2.3.6. If you choose to create an Amazon EMR cluster that does not use the Amazon Redshift bootstrap action, follow the steps in this section.

This section presents the steps to load a table from Amazon EMR without using the Amazon Redshift bootstrap action.

**Loading data process**

This section walks you through the process of loading data from an Amazon EMR cluster. The following sections provide the details you need to accomplish each step.

- **Step 1: Configure IAM permissions (p. 131)**
  
The users that create the Amazon EMR cluster and run the Amazon Redshift COPY command must have the necessary permissions.

- **Step 2: Create an Amazon EMR cluster (p. 131)**
  
Configure the cluster to output text files to the Hadoop Distributed File System (HDFS). You will need the Amazon EMR cluster ID and the cluster’s master public DNS (the endpoint for the Amazon EC2 instance that hosts the cluster).

- **Step 3: Retrieve the Amazon Redshift cluster public key and cluster node IP addresses (p. 132)**
The public key enables the Amazon Redshift cluster nodes to establish SSH connections to the hosts. You will use the IP address for each cluster node to configure the host security groups to permit access from your Amazon Redshift cluster using these IP addresses.

- **Step 4: Add the Amazon Redshift cluster public key to each Amazon EC2 host's authorized keys file (p. 133)**
  
  You add the Amazon Redshift cluster public key to the host's authorized keys file so that the host will recognize the Amazon Redshift cluster and accept the SSH connection.

- **Step 5: Configure the hosts to accept all of the Amazon Redshift cluster's IP addresses (p. 134)**
  
  Modify the Amazon EMR instance's security groups to add ingress rules to accept the Amazon Redshift IP addresses.

- **Step 6: Run the COPY command to load the data (p. 134)**
  
  From an Amazon Redshift database, run the COPY command to load the data into an Amazon Redshift table.

### Step 1: Configure IAM permissions

The users that create the Amazon EMR cluster and run the Amazon Redshift COPY command must have the necessary permissions.

#### To configure IAM permissions

1. Add the following permissions for the IAM user that will create the Amazon EMR cluster.

   ```
   ec2:DescribeSecurityGroups
   ec2:RevokeSecurityGroupIngress
   ec2:AuthorizeSecurityGroupIngress
   redshift:DescribeClusters
   ```

2. Add the following permission for the IAM user that will execute the COPY command.

   ```
   elasticmapreduce:ListInstances
   ```

### Step 2: Create an Amazon EMR cluster

The COPY command loads data from files on the Amazon EMR Hadoop Distributed File System (HDFS). When you create the Amazon EMR cluster, configure the cluster to output data files to the cluster's HDFS.

#### To create an Amazon EMR cluster

1. Create an Amazon EMR cluster in the same AWS region as the Amazon Redshift cluster.

   If the Amazon Redshift cluster is in a VPC, the Amazon EMR cluster must be in the same VPC group. If the Amazon Redshift cluster uses EC2-Classic mode (that is, it is not in a VPC), the Amazon EMR cluster must also use EC2-Classic mode. For more information, see Managing Clusters in Virtual Private Cloud (VPC) in the Amazon Redshift Management Guide.

2. Configure the cluster to output data files to the cluster's HDFS. The HDFS file names must not include asterisks (*) or question marks (?).

   **Important**
   
   The file names must not include asterisks ( * ) or question marks ( ? ).
3. Specify No for the Auto-terminate option in the Amazon EMR cluster configuration so that the cluster remains available while the COPY command executes.

   Important
   If any of the data files are changed or deleted before the COPY completes, you might have unexpected results, or the COPY operation might fail.

4. Note the cluster ID and the master public DNS (the endpoint for the Amazon EC2 instance that hosts the cluster). You will use that information in later steps.

Step 3: Retrieve the Amazon Redshift cluster public key and cluster node IP addresses

To retrieve the Amazon Redshift cluster public key and cluster node IP addresses for your cluster using the console

1. Access the Amazon Redshift Management Console.
2. Click the Clusters link in the left navigation pane.
3. Select your cluster from the list.
4. Locate the SSH Ingestion Settings group.

   Note the Cluster Public Key and Node IP addresses. You will use them in later steps.

You will use the Private IP addresses in Step 3 to configure the Amazon EC2 host to accept the connection from Amazon Redshift.

To retrieve the cluster public key and cluster node IP addresses for your cluster using the Amazon Redshift CLI, execute the describe-clusters command. For example:

```
aws redshift describe-clusters --cluster-identifier <cluster-identifier>
```

The response will include a ClusterPublicKey value and the list of private and public IP addresses, similar to the following:
{
  "Clusters": [
    {
      "VpcSecurityGroups": [],
      "ClusterStatus": "available",
      "ClusterNodes": [
        {
          "PrivateIPAddress": "10.nnn.nnn.nnn",
          "NodeRole": "LEADER",
          "PublicIPAddress": "10.nnn.nnn.nnn"
        },
        {
          "PrivateIPAddress": "10.nnn.nnn.nnn",
          "NodeRole": "COMPUTE-0",
          "PublicIPAddress": "10.nnn.nnn.nnn"
        },
        {
          "PrivateIPAddress": "10.nnn.nnn.nnn",
          "NodeRole": "COMPUTE-1",
          "PublicIPAddress": "10.nnn.nnn.nnn"
        }
      ],
      "AutomatedSnapshotRetentionPeriod": 1,
      "PreferredMaintenanceWindow": "wed:05:30-wed:06:00",
      "AvailabilityZone": "us-east-1a",
      "NodeType": "dw.hs1.xlarge",
      "ClusterPublicKey": "ssh-rsa AAAABexamplepublickey...Y3TA1 Amazon-Redshift",
      ...
    }
  ]
}

To retrieve the cluster public key and cluster node IP addresses for your cluster using the Amazon Redshift API, use the DescribeClusters action. For more information, see describe-clusters in the Amazon Redshift CLI Guide or DescribeClusters in the Amazon Redshift API Guide.

**Step 4: Add the Amazon Redshift cluster public key to each Amazon EC2 host’s authorized keys file**

You add the cluster public key to each host's authorized keys file for all of the Amazon EMR cluster nodes so that the hosts will recognize Amazon Redshift and accept the SSH connection.

**To add the Amazon Redshift cluster public key to the host’s authorized keys file**

1. Access the host using an SSH connection.

   For information about connecting to an instance using SSH, see Connect to Your Instance in the Amazon EC2 User Guide.

2. Copy the Amazon Redshift public key from the console or from the CLI response text.

3. Copy and paste the contents of the public key into the /
/home/<ssh_username>/.ssh/authorized_keys file on the host. Include the complete string, including the prefix "ssh-rsa" and suffix "Amazon-Redshift". For example:

   ```
   ssh-rsa AAAAC5T3isxGzW0IWpbVvRC0zYdVifMrh... uA70BnMHCaMiRdmsDOedZ0edZ Amazon-Redshift
   ```
Step 5: Configure the hosts to accept all of the Amazon Redshift cluster's IP addresses

Add an ingress rule to the security group to allow inbound traffic to the instance from each of your cluster node IP addresses. Use the Amazon Redshift Cluster Node Private IP addresses you retrieved in Step 3: Retrieve the Amazon Redshift cluster public key and cluster node IP addresses (p. 132). For information about adding rules to an Amazon EC2 security group, see Authorizing Inbound Traffic for Your Instances in the Amazon EC2 User Guide.

Step 6: Run the COPY command to load the data

Run a COPY (p. 276) command to connect to the Amazon EMR cluster and load the data into an Amazon Redshift table. The Amazon EMR cluster must continue running until the COPY command completes. For example, do not configure the cluster to auto-terminate.

Important
If any of the data files are changed or deleted before the COPY completes, you might have unexpected results, or the COPY operation might fail.

In the COPY command, specify the Amazon EMR cluster ID and the HDFS file path and file name.

```
copy sales
from 'emr:// j-1H7OU03B52HI5/myoutput/part*' credentials
'aws_access_key_id=<access-key-id>;aws_secret_access_key=<secret-access-key>';  
```

You can use the wildcard characters asterisk (*) and question mark (?) as part of the file name argument. For example, `part*` loads the files `part-0000`, `part-0001`, and so on. If you specify only a folder name, COPY attempts to load all files in the folder.

Important
If you use wildcard characters or use only the folder name, verify that no unwanted files will be loaded or the COPY command will fail. For example, some processes might write a log file to the output folder.

Loading data from remote hosts

You can use the COPY command to load data in parallel from one or more remote hosts, such as Amazon EC2 instances or other computers. COPY connects to the remote hosts using SSH and executes commands on the remote hosts to generate text output.

The remote host can be an Amazon EC2 Linux instance or another Unix or Linux computer configured to accept SSH connections. This guide assumes your remote host is an Amazon EC2 instance. Where the procedure is different for another computer, the guide will point out the difference.

Amazon Redshift can connect to multiple hosts, and can open multiple SSH connections to each host. Amazon Redshift sends a unique command through each connection to generate text output to the host's standard output, which Amazon Redshift then reads as it would a text file.

Before you begin

Before you begin, you should have the following in place:

- One or more host machines, such as Amazon EC2 instances, that you can connect to using SSH.
- Data sources on the hosts.

You will provide commands that the Amazon Redshift cluster will run on the hosts to generate the text output. After the cluster connects to a host, the COPY command runs the commands, reads the text
from the hosts' standard output, and loads the data in parallel into an Amazon Redshift table. The text output must be in a form that the COPY command can ingest. For more information, see Preparing your input data (p. 118)

- Access to the hosts from your computer.

For an Amazon EC2 instance, you will use an SSH connection to access the host. You will need to access the host to add the Amazon Redshift cluster's public key to the host's authorized keys file.

- A running Amazon Redshift cluster.

For information about how to launch a cluster, see Getting Started with Amazon Redshift in the Amazon Redshift Getting Started Guide.

Loading data process

This section walks you through the process of loading data from remote hosts. The following sections provide the details you need to accomplish each step.

1. **Step 1: Retrieve the cluster public key and cluster node IP addresses (p. 136)**

   The public key enables the Amazon Redshift cluster nodes to establish SSH connections to the remote hosts. You will use the IP address for each cluster node to configure the host security groups or firewall to permit access from your Amazon Redshift cluster using these IP addresses.

2. **Step 2: Add the Amazon Redshift cluster public key to the host’s authorized keys file (p. 137)**

   You add the Amazon Redshift cluster public key to the host's authorized keys file so that the host will recognize the Amazon Redshift cluster and accept the SSH connection.

3. **Step 3: Configure the host to accept all of the Amazon Redshift cluster’s IP addresses (p. 138)**

   For Amazon EC2, modify the instance’s security groups to add ingress rules to accept the Amazon Redshift IP addresses. For other hosts, modify the firewall so that your Amazon Redshift nodes are able to establish SSH connections to the remote host.

4. **Step 4: Get the public key for the host (p. 138)**

   You can optionally specify that Amazon Redshift should use the public key to identify the host. You will need to locate the public key and copy the text into your manifest file.

5. **Step 5: Create a manifest file (p. 138)**

   The manifest is a JSON-formatted text file with the details Amazon Redshift needs to connect to the hosts and fetch the data.

6. **Step 6: Upload the manifest file to an Amazon S3 bucket (p. 140)**

   Amazon Redshift reads the manifest and uses that information to connect to the remote host. The bucket must be located in the same region as your Amazon Redshift cluster.

7. **Step 7: Run the COPY command to load the data (p. 140)**

   From an Amazon Redshift database, run the COPY command to load the data into an Amazon Redshift table.
Step 1: Retrieve the cluster public key and cluster node IP addresses

To retrieve the cluster public key and cluster node IP addresses for your cluster using the console

1. Access the Amazon Redshift Management Console.
2. Click the Clusters link in the left navigation pane.
3. Select your cluster from the list.
4. Locate the Copy from SSH Setting group.

Note the Cluster Public Key and Node IP addresses. You will use them in later steps.

You will use the IP addresses in Step 3 to configure the host to accept the connection from Amazon Redshift. Depending on what type of host you connect to and whether it is in a VPC, you will use either the public IP addresses or the private IP addresses.

To retrieve the cluster public key and cluster node IP addresses for your cluster using the Amazon Redshift CLI, execute the describe-clusters command.

For example:

```
aws redshift describe-clusters --cluster-identifier <cluster-identifier>
```

The response will include the ClusterPublicKey and the list of Private and Public IP addresses, similar to the following:

```
{
  "Clusters": [
    {
      "VpcSecurityGroups": [],
```
To retrieve the cluster public key and cluster node IP addresses for your cluster using the Amazon Redshift API, use the DescribeClusters action. For more information, see describe-clusters in the Amazon Redshift CLI Guide or DescribeClusters in the Amazon Redshift API Guide.

Step 2: Add the Amazon Redshift cluster public key to the host’s authorized keys file

You add the cluster public key to each host’s authorized keys file so that the host will recognize Amazon Redshift and accept the SSH connection.

To add the Amazon Redshift cluster public key to the host’s authorized keys file

1. Access the host using an SSH connection.
   You will add the user name you log in with to the "username" field in the manifest file you create in Step 5. For information about connecting to an instance using SSH, see Connect To Your Instance in the Amazon EC2 User Guide.
2. Copy the Amazon Redshift public key from the console or from the CLI response text.
3. Copy and paste the contents of the public key into the 
/home/<ssh_username>/.ssh/authorized_keys file on the remote host. The <ssh_username> must match the value for the "username" field in the manifest file. Include the complete string, including the prefix "ssh-rsa " and suffix "Amazon-Redshift". For example:

```text
ssh-rsa AAAACTP3ixgGzVWoIwpbVvROzYdVifMrh... uA70BnMHCaMiRdmvsDOedZD0edZ Amazon-Redshift
```
Step 3: Configure the host to accept all of the Amazon Redshift cluster's IP addresses

If you are working with an Amazon EC2 instance or an Amazon EMR cluster, add an ingress rule to the security group to allow inbound traffic to the instance from each of your cluster node IP addresses. Use the Amazon Redshift Cluster Node IP addresses you retrieved in Step 1: Retrieve the cluster public key and cluster node IP addresses (p. 136). For information about adding rules to an Amazon EC2 security group, see Authorizing Inbound Traffic for Your Instances in the Amazon EC2 User Guide.

Use the Private IP addresses when:

• You have an Amazon Redshift cluster that is not in a Virtual Private Cloud (VPC), and an Amazon EC2-Classic instance, both of which are in the same AWS region.
• You have an Amazon Redshift cluster that is in a VPC, and an Amazon EC2-VPC instance, both of which are in the same AWS region and in the same VPC.

Otherwise, use the Public IP addresses.

For more information about using Amazon Redshift in a VPC, see Managing Clusters in Virtual Private Cloud (VPC) in the Amazon Redshift Management Guide.

Step 4: Get the public key for the host

You can optionally provide the host's public key in the manifest file so that Amazon Redshift can identify the host. The COPY command does not require the host public key but, for security reasons, we strongly recommend using a public key to help prevent 'man-in-the-middle' attacks.

You can find the host's public key in the following location, where <ssh_host_rsa_key_name> is the unique name for the host's public key:

```
/etc/ssh/<ssh_host_rsa_key_name>.pub
```

**Note**
Amazon Redshift only supports RSA keys. We do not support DSA keys.

When you create your manifest file in Step 5, you will paste the text of the public key into the "Public Key" field in the manifest file entry.

Step 5: Create a manifest file

The COPY command can connect to multiple hosts using SSH, and can create multiple SSH connections to each host. COPY executes a command through each host connection, and then loads the output from the commands in parallel into the table. The manifest file is a text file in JSON format that Amazon Redshift uses to connect to the host. The manifest file specifies the SSH host endpoints and the commands that will be executed on the hosts to return data to Amazon Redshift. Optionally, you can include the host public key, the login user name, and a mandatory flag for each entry.

The manifest file is in the following format:

```json
{
    "entries": [
        {
            "endpoint": "<ssh_endpoint_or_IP>",
            "command": "<remote_command>",
            "mandatory": true,
        }
    ]
}
```
The manifest file contains one "entries" construct for each SSH connection. Each entry represents a single SSH connection. You can have multiple connections to a single host or multiple connections to multiple hosts. The double quotes are required as shown, both for the field names and the values. The only value that does not need double quotes is the Boolean value true or false for the mandatory field.

The following table describes the fields in the manifest file.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>endpoint</td>
<td>The URL address or IP address of the host. For example, &quot;ec2-111-222-333.compute-1.amazonaws.com&quot; or &quot;22.33.44.56&quot;</td>
</tr>
<tr>
<td>command</td>
<td>The command that will be executed by the host to generate text or binary (gzip or lzop) output. The command can be any command that the user &quot;host_user_name&quot; has permission to run. The command can be as simple as printing a file, or it could query a database or launch a script. The output (text file, gzip binary file, or lzop binary file) must be in a form the Amazon Redshift COPY command can ingest. For more information, see Preparing your input data (p. 118).</td>
</tr>
<tr>
<td>publickey</td>
<td>(Optional) The public key of the host. If provided, Amazon Redshift will use the public key to identify the host. If the public key is not provided, Amazon Redshift will not attempt host identification. For example, if the remote host's public key is: ssh-rsa AbcCbaxxx...xxxDHKJ <a href="mailto:root@amazon.com">root@amazon.com</a> enter the following text in the publickey field: AbcCbaxxx...xxxDHKJ.</td>
</tr>
<tr>
<td>mandatory</td>
<td>(Optional) Indicates whether the COPY command should fail if the connection fails. The default is false. If Amazon Redshift does not successfully make at least one connection, the COPY command fails.</td>
</tr>
<tr>
<td>username</td>
<td>(Optional) The username that will be used to log in to the host system and execute the remote command. The user login name must be the same as the login that was used to add the public key to the host's authorized keys file in Step 2. The default username is &quot;redshift&quot;.</td>
</tr>
</tbody>
</table>

The following example shows a completed manifest to open four connections to the same host and execute a different command through each connection:

```
{
  "entries": [
    {
      "endpoint": "ec2-184-72-204-112.compute-1.amazonaws.com",
      "command": "cat loaddata1.txt",
      "mandatory":true,
      "publickey": "ec2publickeyportionoftheec2keypair",
      "username": "ec2-user"},
    {
      "endpoint": "ec2-184-72-204-112.compute-1.amazonaws.com",
      "command": "cat loaddata2.txt",
      "mandatory":true,
      "publickey": "ec2publickeyportionoftheec2keypair",
      "username": "ec2-user"},
    ...
  ]
}
```
Step 6: Upload the manifest file to an Amazon S3 bucket

Upload the manifest file to an Amazon S3 bucket that is located in the same region as your Amazon Redshift cluster. For information about creating an Amazon S3 bucket and uploading a file, see Get Started With Amazon Simple Storage Service in the Amazon S3 Getting Starting Guide.

Step 7: Run the COPY command to load the data

Run a COPY (p. 276) command to connect to the host and load the data into an Amazon Redshift table. In the COPY command, specify the explicit Amazon S3 object path for the manifest file and include the SSH option. For example,

```
copy sales
from 's3://mybucket/ssh_manifest' credentials
'aws_access_key_id=<access-key-id>;aws_secret_access_key=<secret-access-key>'
ssh;
```

Note
If you use automatic compression, the COPY command performs two data reads, which means it will execute the remote command twice. The first read is to provide a sample for compression analysis, then the second read actually loads the data. If executing the remote command twice might cause a problem because of potential side effects, you should disable automatic compression. To disable automatic compression, run the COPY command with the COMPUPDATE option set to OFF. For more information, see Loading tables with automatic compression.

Loading data from an Amazon DynamoDB table

You can use the COPY command to load a table with data from a single Amazon DynamoDB table. Important
The Amazon DynamoDB table that provides the data must be created in the same region as your cluster unless you use the REGION (p. 285) option to specify the region in which the Amazon DynamoDB table is located.

The COPY command leverages the Amazon Redshift massively parallel processing (MPP) architecture to read and load data in parallel from an Amazon DynamoDB table. You can take maximum advantage of parallel processing by setting distribution styles on your Amazon Redshift tables. For more information, see Choosing a data distribution style (p. 101).

Important
When the COPY command reads data from the Amazon DynamoDB table, the resulting data transfer is part of that table's provisioned throughput.
To avoid consuming excessive amounts of provisioned read throughput, we recommend that you not load data from Amazon DynamoDB tables that are in production environments. If you do load data from production tables, we recommend that you set the READRATIO option much lower than the average percentage of unused provisioned throughput. A low READRATIO setting will help minimize throttling issues. To use the entire provisioned throughput of an Amazon DynamoDB table, set READRATIO to 100.

The COPY command matches attribute names in the items retrieved from the Amazon DynamoDB table to column names in an existing Amazon Redshift table by using the following rules:

- Amazon Redshift table columns are case-insensitively matched to Amazon DynamoDB item attributes. If an item in the DynamoDB table contains multiple attributes that differ only in case, such as Price and PRICE, the COPY command will fail.
- Amazon Redshift table columns that do not match an attribute in the Amazon DynamoDB table are loaded as either NULL or empty, depending on the value specified with the EMPTYASNULL option in the COPY (p. 276) command.
- Amazon DynamoDB attributes that do not match a column in the Amazon Redshift table are discarded. Attributes are read before they are matched, and so even discarded attributes consume part of that table's provisioned throughput.
- Only Amazon DynamoDB attributes with scalar STRING and NUMBER data types are supported. The Amazon DynamoDB BINARY and SET data types are not supported. If a COPY command tries to load an attribute with an unsupported data type, the command will fail. If the attribute does not match an Amazon Redshift table column, COPY does not attempt to load it, and it does not raise an error.

The COPY command uses the following syntax to load data from an Amazon DynamoDB table:

```
copy <redshift_tablename> from 'dynamodb://<dynamodb_table_name>'
credentials 'aws_access_key_id=<access-key-id>;aws_secret_access_key=<secret-access-key>'
readratio '<integer>';
```

The values for <your-access-key-id> and <your-secret-access-key> are the AWS credentials needed to access the Amazon DynamoDB table. If these credentials correspond to an IAM user, that IAM user must have permission to SCAN and DESCRIBE the Amazon DynamoDB table that is being loaded.

You can limit the access users have to your data by using temporary security credentials. Temporary security credentials provide enhanced security because they have short life spans and cannot be reused after they expire. A user who has these temporary security credentials can access your resources only until the credentials expire. For more information, see Temporary security credentials (p. 290). To load data using temporary access credentials, use the following syntax:

```
copy <redshift_tablename> from 'dynamodb://<dynamodb_table_name>'
credentials 'aws_access_key_id=<temporary-access-key-id>;aws_secret_access_key=<temporary-access-key>;token=<temporary-token>;z
```

Important
The temporary security credentials must be valid for the entire duration of the COPY statement. If the temporary security credentials expire during the load process, the COPY will fail and the transaction will be rolled back. For example, if temporary security credentials expire after 15 minutes and the COPY requires one hour, the COPY will fail before it completes.

If you want to validate your data without actually loading the table, use the NOLOAD option with the COPY (p. 276) command.
The following example loads the FAVORITEMOVIES table with data from the Amazon DynamoDB table my-favorite-movies-table. The read activity can consume up to 50% of the provisioned throughput.

```
copy favoritemovies from 'dynamodb://my-favorite-movies-table'
credentials 'aws_access_key_id=<access-key-id>;aws_secret_access_key=<secret-access-key>'
readratio 50;
```

To maximize throughput, the COPY command loads data from an Amazon DynamoDB table in parallel across the compute nodes in the cluster.

### Provisioned throughput with automatic compression

By default, the COPY command applies automatic compression whenever you specify an empty target table with no compression encoding. The automatic compression analysis initially samples a large number of rows from the Amazon DynamoDB table. The sample size is based on the value of the COMPROWS parameter. The default is 100,000 rows per slice.

After sampling, the sample rows are discarded and the entire table is loaded. As a result, many rows are read twice. For more information about how automatic compression works, see [Loading tables with automatic compression](p. 143).

**Important**

When the COPY command reads data from the Amazon DynamoDB table, including the rows used for sampling, the resulting data transfer is part of that table’s provisioned throughput.

### Loading multibyte data from Amazon DynamoDB

If your data includes non-ASCII multibyte characters (such as Chinese or Cyrillic characters), you must load the data to VARCHAR columns. The VARCHAR data type supports four-byte UTF-8 characters, but the CHAR data type only accepts single-byte ASCII characters. You cannot load five-byte or longer characters into Amazon Redshift tables. For more information about CHAR and VARCHAR, see [Data types](p. 214).

### Verifying that the data was loaded correctly

After the load operation is complete, query the `STL_LOAD_COMMITS` system table to verify that the expected files were loaded. You should execute the COPY command and load verification within the same transaction so that if there is problem with the load you can roll back the entire transaction.

**Important**

The US Standard Region, which includes the US East Region, provides eventual consistency for all Amazon S3 requests, so there might be lag between the time you upload your files to Amazon S3 and the time they are available to be loaded into Amazon Redshift. For more information about data consistency, see [Managing data consistency](p. 121).

The following query returns entries for loading the tables in the TICKIT database:

```
select query, trim(filename) as filename, curtime, status
from stl_load_commits
where filename like '%tickit%' order by query;
```

```
query |           btrim           |          curtime           | status
--- | ---------------------------|----------------------------|--------
22475 | tickit/allusers_pipe.txt   | 2013-02-08 20:58:23.274186 |      1
22478 | tickit/venue_pipe.txt      | 2013-02-08 20:58:25.070604 |      1
```
Validating input data

To validate the data in the Amazon S3 input files or Amazon DynamoDB table before you actually load the data, use the NOLOAD option with the COPY (p. 276) command. Use NOLOAD with the same COPY commands and options you would use to actually load the data. NOLOAD checks the integrity of all of the data without loading it into the database. The NOLOAD option displays any errors that would occur if you had attempted to load the data.

For example, if you specified the incorrect Amazon S3 path for the input file, Amazon Redshift would display the following error:

```
ERROR:  No such file or directory
DETAIL:
-----------------------------------------------
Amazon Redshift error:  The specified key does not exist
code:      2
color:     S3 key being read :
location:  step_scan.cpp:1883
process:   xenmaster [pid=22199]
-----------------------------------------------
```

To troubleshoot error messages, see the Load error reference (p. 149).

Loading tables with automatic compression

Topics

- How automatic compression works (p. 143)
- Automatic compression example (p. 144)

You can apply compression encodings to columns in tables manually, based on your own evaluation of the data, or you can use the COPY command to analyze and apply compression automatically. We strongly recommend using the COPY command to apply automatic compression.

You can use automatic compression when you create and load a brand new table. The COPY command will perform a compression analysis. You can also perform a compression analysis without loading data or changing the compression on a table by running the ANALYZE COMPRESSION (p. 269) command against an already populated table. For example, you can run the ANALYZE COMPRESSION command when you want to analyze compression on a table for future use, while preserving the existing DDL.

How automatic compression works

By default, the COPY command applies automatic compression whenever you run the COPY command with an empty target table and all of the table columns either have RAW encoding or no encoding.

To apply automatic compression to an empty table, regardless of its current compression encodings, run the COPY command with the COMPUPDATE option set to ON. To disable automatic compression, run the COPY command with the COMPUPDATE option set to OFF.
You cannot apply automatic compression to a table that already contains data.

**Note**
Automatic compression analysis requires enough rows in the load data (at least 100,000 rows per slice) to allow sampling to take place.

Automatic compression performs these operations in the background as part of the load transaction:

1. An initial sample of rows is loaded from the input file. Sample size is based on the value of the COMPROWS parameter. The default is 100,000.
2. Compression options are chosen for each column.
3. The sample rows are removed from the table.
4. If enough data is being loaded to provide a meaningful sample, the table is re-created with the chosen compression encodings.
5. The entire input file is loaded and compressed using the new encodings.

After you run the COPY command, the table is fully loaded, compressed, and ready for use. If you load more data later, appended rows are compressed according to the existing encoding.

If you only want to perform a compression analysis, run ANALYZE COMPRESSION, which is more efficient than running a full COPY. Then you can evaluate the results to decide whether to use automatic compression or recreate the table manually.

Automatic compression is supported only for the COPY command. Alternatively, you can manually apply compression encoding when you create the table. For information about manual compression encoding, see Choosing a column compression type (p. 90).

**Automatic compression example**

In this example, assume that the TICKIT database contains a copy of the LISTING table called BIGLIST, and you want to apply automatic compression to this table when it is loaded with approximately 3 million rows.

**To load and automatically compress the table**

1. Ensure that the table is empty. You can apply automatic compression only to an empty table:

   ```sql
   truncate biglist;
   ```

2. Load the table with a single COPY command. Although the table is empty, some earlier encoding might have been specified. To ensure that Amazon Redshift performs a compression analysis, set the COMPUPDATE parameter to ON.

   ```sql
   copy biglist from 's3://mybucket/biglist.txt'
   credentials 'aws_access_key_id=<access-key-id>;aws_secret_access_key=<secret-access-key>'
   delimiter '|' COMPUPDATE ON;
   ```

   Because no COMPROWS option is specified, the default and recommended sample size of 100,000 rows per slice is used.

3. Look at the new schema for the BIGLIST table in order to review the automatically chosen encoding schemes.
select "column", type, encoding
from pg_table_def where tablename = 'biglist';

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>listid</td>
<td>integer</td>
<td>delta</td>
</tr>
<tr>
<td>sellerid</td>
<td>integer</td>
<td>delta32k</td>
</tr>
<tr>
<td>eventid</td>
<td>integer</td>
<td>delta32k</td>
</tr>
<tr>
<td>dateid</td>
<td>smallint</td>
<td>delta</td>
</tr>
<tr>
<td>+numtickets</td>
<td>smallint</td>
<td>delta</td>
</tr>
<tr>
<td>priceticket</td>
<td>numeric(8,2)</td>
<td>delta32k</td>
</tr>
<tr>
<td>totalprice</td>
<td>numeric(8,2)</td>
<td>mostly32</td>
</tr>
<tr>
<td>listtime</td>
<td>timestamp without time zone</td>
<td>none</td>
</tr>
</tbody>
</table>

4. Verify that the expected number of rows were loaded:

```
select count(*) from biglist;
```

<table>
<thead>
<tr>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>3079952</td>
</tr>
</tbody>
</table>

When rows are later appended to this table using COPY or INSERT statements, the same compression encodings will be applied.

**Optimizing storage for narrow tables**

If you have a table with very few columns but a very large number of rows, the three hidden metadata identity columns (INSERT_XID, DELETE_XID, ROW_ID) will consume a disproportionate amount of the disk space for the table.

In order to optimize compression of the hidden columns, load the table in a single COPY transaction where possible. If you load the table with multiple separate COPY commands, the INSERT_XID column will not compress well. You will need to perform a vacuum operation if you use multiple COPY commands, but it will not improve compression of INSERT_XID.

**Loading default column values**

You can optionally define a column list in your COPY command. If a column in the table is omitted from the column list, COPY will load the column with either the value supplied by the DEFAULT option that was specified in the CREATE TABLE command, or with NULL if the DEFAULT option was not specified.

If COPY attempts to assign NULL to a column that is defined as NOT NULL, the COPY command fails. For information about assigning the DEFAULT option, see CREATE TABLE (p. 310).

When loading from data files on Amazon S3, the columns in the column list must be in the same order as the fields in the data file. If a field in the data file does not have a corresponding column in the column list, the COPY command fails.

When loading from Amazon DynamoDB table, order does not matter. Any fields in the Amazon DynamoDB attributes that do not match a column in the Amazon Redshift table are discarded.
The following restrictions apply when using the COPY command to load DEFAULT values into a table:

- If an IDENTITY (p. 312) column is included in the column list, the EXPLICIT_IDS option must also be specified in the COPY (p. 276) command, or the COPY command will fail. Similarly, if an IDENTITY column is omitted from the column list, and the EXPLICIT_IDS option is specified, the COPY operation will fail.
- Because the evaluated DEFAULT expression for a given column is the same for all loaded rows, a DEFAULT expression that uses a RANDOM() function will assign to same value to all the rows.
- DEFAULT expressions that contain CURRENT_DATE or SYSDATE are set to the timestamp of the current transaction.

For an example, see "Load data from a file with default values" in COPY examples (p. 299).

**Troubleshooting data loads**

**Topics**
- S3ServiceException errors (p. 146)
- System tables for troubleshooting data loads (p. 147)
- Multi-byte character load errors (p. 148)
- Load error reference (p. 149)

This section provides information about identifying and resolving data loading errors.

**S3ServiceException errors**

The most common s3ServiceException errors are caused by an improperly formatted credentials sting, having your cluster and your bucket in different regions, and insufficient Amazon S3 privileges.

The section provides troubleshooting information for each type of error.

**Invalid credentials string**

If your credentials string was improperly formatted, you will receive the following error message:

```
ERROR: Invalid credentials. Must be of the format: credentials
    'aws_access_key_id=<access-key-id>;aws_secret_access_key=<secret-access-key>
    [;token=<temporary-session-token>]' 
```

Verify that the credentials string does not contain any spaces or line breaks, and is enclosed in single quotes.

**Bucket is in a different region**

The Amazon S3 bucket specified in the COPY command must be in the same region as the cluster. If your Amazon S3 bucket and your cluster are in different regions, you will receive an error similar to the following:

```
ERROR: S3ServiceException:The bucket you are attempting to access must be addressed using the specified endpoint. 
```

You can create an Amazon S3 bucket in a specific region either by selecting the region when you create the bucket by using the Amazon S3 Management Console, or by specifying an endpoint when you create
the bucket using the Amazon S3 API or CLI. For more information, see Uploading files to Amazon S3 (p. 120).

For more information about Amazon S3 regions, see Buckets and Regions in the Amazon Simple Storage Service Developer Guide.

Alternatively, you can specify the region using the REGION (p. 285) option with the COPY command.

**Access denied**

The user account identified by the credentials must have LIST and GET access to the Amazon S3 bucket. If the user does not have sufficient privileges, you will receive the following error message:

```
ERROR: S3ServiceException:Access Denied,Status 403,Error AccessDenied
```

For information about managing user access to buckets, see Access Control in the Amazon S3 Developer Guide.

### System tables for troubleshooting data loads

The following Amazon Redshift system tables can be helpful in troubleshooting data load issues:

- Query STL_LOAD_ERRORS (p. 608) to discover the errors that occurred during specific loads.
- Query STL_FILE_SCAN (p. 599) to view load times for specific files or to see if a specific file was even read.

#### To find and diagnose load errors

1. Create a view or define a query that returns details about load errors. The following example joins the STL_LOAD_ERRORS table to the STV_TBL_PERM table to match table IDs with actual table names.

   ```sql
   create view loadview as
   (select distinct tbl, trim(name) as table_name, query, starttime,
   trim(filename) as input, line_number, colname, err_code,
   trim(err_reason) as reason
   from stl_load_errors sl, stv_tbl_perm sp
   where sl.tbl = sp.id);
   ```

2. Set the MAXERRORS option in your COPY command to a large enough value to enable COPY to return useful information about your data. If the COPY encounters errors, an error message directs you to consult the STL_LOAD_ERRORS table for details.

3. Query the LOADVIEW view to see error details. For example:

   ```sql
   select * from loadview where table_name='venue';
   ```

<table>
<thead>
<tr>
<th>tbl</th>
<th>table_name</th>
<th>query</th>
<th>starttime</th>
</tr>
</thead>
<tbody>
<tr>
<td>100551</td>
<td>venue</td>
<td>20974</td>
<td>2013-01-29 19:05:58.365391</td>
</tr>
</tbody>
</table>
   |       | input      | line_number | colname | err_code | reason
   +---------------------------------+-----------------+-----------------+-----------------+-----------------+-----------------
4. Fix the problem in the input file or the load script, based on the information that the view returns. Some typical load errors to watch for include:

- Mismatch between data types in table and values in input data fields.
- Mismatch between number of columns in table and number of fields in input data.
- Mismatched quotes. Amazon Redshift supports both single and double quotes; however, these quotes must be balanced appropriately.
- Incorrect format for date/time data in input files.
- Out-of-range values in input files (for numeric columns).
- Number of distinct values for a column exceeds the limitation for its compression encoding.

**Multi-byte character load errors**

Columns with a CHAR data type only accept single-byte UTF-8 characters, up to byte value 127, or 7F hex, which is also the ASCII character set. VARCHAR columns accept multi-byte UTF-8 characters, to a maximum of four bytes. For more information, see Character types (p. 223).

If a line in your load data contains a character that is invalid for the column data type, COPY returns an error and logs a row in the STL_LOAD_ERRORS system log table with error number 1220. The ERR_REASON field includes the byte sequence, in hex, for the invalid character.

An alternative to fixing invalid characters in your load data is to replace the invalid characters during the load process. To replace invalid UTF-8 characters, specify the ACCEPTINVCHARS option with the COPY command. For more information, see ACCEPTINVCHARS (p. 286).

The following example shows the error reason when COPY attempts to load UTF-8 character e0 a1 c7a4 into a CHAR column:

Multibyte character not supported for CHAR
(Hint: Try using VARCHAR). Invalid char: e0 a1 c7a4

If the error is related to a VARCHAR datatype, the error reason includes an error code as well as the invalid UTF-8 hex sequence. The following example shows the error reason when COPY attempts to load UTF-8 a4 into a VARCHAR field:

String contains invalid or unsupported UTF-8 codepoints.
Bad UTF-8 hex sequence: a4 (error 3)

The following table lists the descriptions and suggested workarounds for VARCHAR load errors. If one of these errors occurs, replace the character with a valid UTF-8 code sequence or remove the character.

<table>
<thead>
<tr>
<th>Error code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>The UTF-8 byte sequence exceeds the four-byte maximum supported by VARCHAR.</td>
</tr>
<tr>
<td>2</td>
<td>The UTF-8 byte sequence is incomplete. COPY did not find the expected number of continuation bytes for a multi-byte character before the end of the string.</td>
</tr>
</tbody>
</table>
### Load error codes

<table>
<thead>
<tr>
<th>Error code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1200</td>
<td>Unknown parse error. Contact support.</td>
</tr>
<tr>
<td>1201</td>
<td>Field delimiter was not found in the input file.</td>
</tr>
<tr>
<td>1202</td>
<td>Input data had more columns than were defined in the DDL.</td>
</tr>
<tr>
<td>1203</td>
<td>Input data had fewer columns than were defined in the DDL.</td>
</tr>
<tr>
<td>1204</td>
<td>Input data exceeded the acceptable range for the data type.</td>
</tr>
<tr>
<td>1205</td>
<td>Date format is invalid. See <a href="#">DATEFORMAT and TIMEFORMAT strings</a> for valid formats.</td>
</tr>
<tr>
<td>1206</td>
<td>Timestamp format is invalid. See <a href="#">DATEFORMAT and TIMEFORMAT strings</a> for valid formats.</td>
</tr>
<tr>
<td>1207</td>
<td>Data contained a value outside of the expected range of 0-9.</td>
</tr>
<tr>
<td>1208</td>
<td>FLOAT data type format error.</td>
</tr>
<tr>
<td>1209</td>
<td>DECIMAL data type format error.</td>
</tr>
<tr>
<td>1210</td>
<td>BOOLEAN data type format error.</td>
</tr>
<tr>
<td>1211</td>
<td>Input line contained no data.</td>
</tr>
<tr>
<td>1212</td>
<td>Load file was not found.</td>
</tr>
<tr>
<td>1213</td>
<td>A field specified as NOT NULL contained no data.</td>
</tr>
<tr>
<td>1214</td>
<td>VARCHAR field error.</td>
</tr>
</tbody>
</table>
### Error code Description

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1215</td>
<td>CHAR field error.</td>
</tr>
<tr>
<td>1216</td>
<td>Invalid input line.</td>
</tr>
<tr>
<td>1217</td>
<td>Invalid identity column value.</td>
</tr>
<tr>
<td>1218</td>
<td>When using NULL AS <code>\0</code>, a field containing a null terminator (NUL, or UTF-8 0000) contained more than one byte.</td>
</tr>
<tr>
<td>1219</td>
<td>UTF-8 hexadecimal contains an invalid digit.</td>
</tr>
<tr>
<td>1220</td>
<td>String contains invalid or unsupported UTF-8 codepoints.</td>
</tr>
</tbody>
</table>

### Updating tables with DML commands

Amazon Redshift supports standard Data Manipulation Language (DML) commands (INSERT, UPDATE, and DELETE) that you can use to modify rows in tables. You can also use the TRUNCATE command to do fast bulk deletes.

**Note**
We strongly encourage you to use the COPY (p. 276) command to load large amounts of data. Using individual INSERT statements to populate a table might be prohibitively slow. Alternatively, if your data already exists in other Amazon Redshift database tables, use SELECT INTO ... INSERT or CREATE TABLE AS to improve performance. For information, see INSERT (p. 349) or CREATE TABLE AS (p. 320).

If you insert, update, or delete a significant number of rows in a table, relative to the number of rows before the changes, run the ANALYZE and VACUUM commands against the table when you are done. If a number of small changes accumulate over time in your application, you might want to schedule the ANALYZE and VACUUM commands to run at regular intervals. For more information, see Analyzing tables (p. 158) and Vacuuming tables (p. 161).

### Updating and inserting new data

You can efficiently add new data to an existing table by using a combination of updates and inserts from a staging table. While Amazon Redshift does not support a single merge, or upsert, command to update a table from a single data source, you can perform a merge operation by creating a staging table and then using one of the methods described in this section to update the target table from the staging table.

**Topics**
- Merge Method 1: Replacing existing rows (p. 151)
- Merge Method 2: Specifying a column list (p. 151)
- Creating a temporary staging table (p. 151)
- Performing a merge operation by replacing existing rows (p. 152)
- Performing a merge operation by specifying a column list (p. 152)
- Merge examples (p. 154)

**Note**
You should run the entire merge operation, including creating and dropping the temporary staging table, in a single transaction so that the transaction will roll back if any step fails. Using a single transaction also reduces the number of commits, which saves time and resources.
Merge Method 1: Replacing existing rows

If you are overwriting all of the columns in the target table, the fastest method for performing a merge is by replacing the existing rows because it scans the target table only once, by using an inner join to delete rows that will be updated. After the rows are deleted, they are replaced along with new rows by a single insert operation from the staging table.

Use this method if all of the following are true:

- Your target table and your staging table contain the same columns.
- You intend to replace all of the data in the target table columns with all of the staging table columns.
- You will use all of the rows in the staging table in the merge.

If any of these criteria do not apply, use Merge Method 2: Specifying a column list, described in the following section.

If you will not use all of the rows in the staging table, you can filter the DELETE and INSERT statements by using a WHERE clause to leave out rows that are not actually changing. However, if most of the rows in the staging table will not participate in the merge, we recommend performing an UPDATE and an INSERT in separate steps, as described later in this section.

Merge Method 2: Specifying a column list

Use this method to update specific columns in the target table instead of overwriting entire rows. This method takes longer than the previous method because it requires an extra update step. Use this method if any of the following are true:

- Not all of the columns in the target table are to be updated.
- Most rows in the staging table will not be used in the updates.

Creating a temporary staging table

The staging table is a temporary table that holds all of the data that will be used to make changes to the target table, including both updates and inserts.

A merge operation requires a join between the staging table and the target table. To collocate the joining rows, set the staging table’s distribution key to the same column as the target table’s distribution key. For example, if the target table uses a foreign key column as its distribution key, use the same column for the staging table’s distribution key. If you create the staging table by using a CREATE TABLE LIKE (p. 313) statement or a CREATE TABLE AS (p. 320) statement, the staging table will inherit the distribution key from the parent table. For more information, see Choosing a data distribution style (p. 101)

If the distribution key is not the same as the primary key and the distribution key is not updated as part of the merge operation, add a redundant join predicate on the distribution key columns to enable a collocated join. For example:

```
where target.primarykey = stage.primarykey
and target.distkey = stage.distkey
```

To verify that the query will use a collocated join, run the query with EXPLAIN (p. 340) and check for DS_DIST_NONE on all of the joins. For more information, see Evaluating the query plan (p. 104)
Performing a merge operation by replacing existing rows

To perform a merge operation by replacing existing rows

1. Put the entire operation in a single transaction block so that if there is a problem, everything will be rolled back.

   ```sql
   begin transaction;
   ...
   end transaction;
   ```

2. Create a staging table, and then populate it with data to be merged.

   ```sql
   create temp table stage like target;
   insert into stage
   select * from source
   where source.filter = 'filter_expression';
   ```

3. Use an inner join with the staging table to delete the rows from the target table that are being updated.

   ```sql
   delete from target
   using stage
   where target.primarykey = stage.primarykey;
   ```

4. Insert all of the rows from the staging table.

   ```sql
   insert into target
   select * from stage;
   ```

5. Drop the staging table.

   ```sql
   drop table stage;
   ```

Performing a merge operation by specifying a column list

To perform a merge operation by specifying a column list

1. Put the entire operation in a single transaction block so that if there is a problem, everything will be rolled back.

   ```sql
   begin transaction;
   ...
   end transaction;
   ```
2. Create a staging table, and then populate it with data to be merged.

```sql
create temp table stage like target;
insert into stage
select * from source
where source.filter = 'filter_expression';
```

3. Update the target table by using an inner join with the staging table.

   - In the UPDATE clause, explicitly list the columns to be updated.
   - Perform an inner join with the staging table.
   - If the distribution key is different from the primary key and the distribution key is not being updated, add a redundant join on the distribution key. To verify that the query will use a collocated join, run the query with EXPLAIN (p. 340) and check for DS_DIST_NONE on all of the joins. For more information, see Evaluating the query plan (p. 104).
   - If your target table is sorted by time stamp, add a predicate to take advantage of range-restricted scans on the target table. For more information, see Specify redundant predicates on the sort column (p. 34).
   - If you will not use all of the rows in the merge, add a clause to filter the rows that need to be changed. For example, add an inequality filter on one or more columns to exclude rows that have not changed.

For example:

```sql
update target
    set target.col1 = staging.col1,
    target.col2 = staging.col2,
    target.col3 = 'expression'
from stage
 where target.key = stage.key
   and target.distkey = stage.distkey
   and target.timestamp > 'last_update_time'
   and (target.col1 != staging.col1
        or target.col2 != staging.col2
        or target.key = 'filter_expression');
```

4. Delete unneeded rows from the staging table by using an inner join with the target table. Some rows in the target table already match the corresponding rows in the staging table, and others were updated in the previous step. In either case, they are not needed for the insert.

```sql
delete from stage
using target
where stage.primarykey = target.primarykey;
```

5. Insert the remaining rows from the staging table. Use the same column list in the VALUES clause that you used in the UPDATE statement in step two.

```sql
insert into target
    select target.col1 = stage.col1,
    target.col2 = stage.col2,
    target.col3 = 'expression'
from stage;
```
6. Drop the staging table.

```sql
drop table stage;
```

## Merge examples

The following examples perform a merge to update the SALES table. The first example uses the simpler method of deleting from the target table and then inserting all of the rows from the staging table. The second example requires updating on select columns in the target table, so it includes an extra update step.

### Sample merge data source

The examples in this section need a sample data source that includes both updates and inserts. For the examples, we will create a sample table named SALES_UPDATE that uses data from the SALES table. We'll populate the new table with random data that represents new sales activity for December. We will use the SALES_UPDATE sample table to create the staging table in the examples that follow.

```sql
-- Create a sample table as a copy of the SALES table
create table sales_update as
select * from sales;

-- Change every fifth row so we have updates
update sales_update
set qtysold = qtysold*2,
pricepaid = pricepaid*0.8,
commission = commission*1.1
where saletime > '2008-11-30'
and mod(sellerid, 5) = 0;

-- Add some new rows so we have insert examples
-- This example creates a duplicate of every fourth row
insert into sales_update
select (salesid + 172456) as salesid, listid, sellerid, buyerid, eventid, dateid,
qtysold, pricepaid, commission, getdate() as saletime
from sales_update
where saletime > '2008-11-30'
and mod(sellerid, 4) = 0;
```

### Example of a merge that replaces existing rows

The following script uses the SALES_UPDATE table to perform a merge operation on the SALES table with new data for December sales activity. This example deletes rows in the SALES table that have updates so they can be replaced with the updated rows in the staging table. The staging table should contain only rows that will participate in the merge, so the CREATE TABLE statement includes a filter to exclude rows that have not changed.

```sql
-- Start a new transaction
begin transaction;

-- Create a staging table and populate it with updated rows from SALES_UPDATE
```
create temp table stagesales as
select * from sales_update
where sales_update.saletime > '2008-11-30'
and sales_update.salesid = (select sales.salesid from sales
where sales.salesid = sales_update.salesid
and sales.listid = sales_update.listid
and (sales_update.qtysold != sales.qtysold
or sales_update.pricepaid != sales.pricepaid));

-- Delete any rows from SALES that exist in STAGESALES, because they are updates
-- The join includes a redundant predicate to collocate on the distribution key
-- A filter on saletime enables a range-restricted scan on SALES

delete from sales
using stagesales
where sales.salesid = stagesales.salesid
and sales.listid = stagesales.listid
and sales.saletime > '2008-11-30';

-- Insert all the rows from the staging table into the target table
insert into sales
select * from stagesales;

-- Drop the staging table
drop table stagesales;

-- End transaction and commit
end transaction;

Example of a merge that specifies a column list

The following example performs a merge operation to update SALES with new data for December sales activity. We need sample data that includes both updates and inserts, along with rows that have not changed. For this example, we want to update the QTYSOLD and PRICEPAID columns but leave COMMISSION and SALETIME unchanged. The following script uses the SALES_UPDATE table to perform a merge operation on the SALES table.

-- Start a new transaction
begin transaction;

-- Create a staging table and populate it with rows from SALES_UPDATE for Dec
create temp table stagesales as select * from sales_update
where saletime > '2008-11-30';

-- Update the target table using an inner join with the staging table
-- The join includes a redundant predicate to collocate on the distribution key
-- A filter on saletime enables a range-restricted scan on SALES
update sales
set qtysold = stagesales.qtysold,
pricepaid = stagesales.pricepaid
from stagesales
where sales.salesid = stagesales.salesid
and sales.listid = stagesales.listid
and stagesales.saletime > '2008-11-30'
and (sales.qtysold != stagesales.qtysold
or sales.pricepaid != stagesales.pricepaid);
Performing a deep copy

A deep copy recreates and repopulates a table by using a bulk insert, which automatically sorts the table. If a table has a large unsorted region, a deep copy is much faster than a vacuum. The trade off is that you cannot make concurrent updates during a deep copy operation, which you can do during a vacuum.

You can choose one of four methods to create a copy of the original table:

- Use the original table DDL.
  
  If the CREATE TABLE DDL is available, this is the best method.
- Use CREATE TABLE AS (CTAS).
  
  If the original DDL is not available, you can use CREATE TABLE AS to create a copy of current table, then rename the copy; however, the new table will not inherit the encoding, distkey, sortkey, notnull, primary key, and foreign key attributes of the parent table. If the original DDL is not available, and you do not need to retain the table attributes, this is the fastest method.
- Use CREATE TABLE LIKE.
  
  If the original DDL is not available, you can use CREATE TABLE LIKE to recreate the original table; however, the new table will not inherit the primary key and foreign key attributes of the parent table. The new table does inherit the encoding, distkey, sortkey, and notnull attributes of the parent table. This method is faster than the following method, creating a temporary table and truncating the original table, because it only uses one insert statement.
- Create a temporary table and truncate the original table.
  
  If you need to retain the primary key and foreign key attributes of the parent table, you can use CTAS to create a temporary table, then truncate the original table and populate it from the temporary table. This method is slower than CREATE TABLE LIKE because it requires two insert statements.

To perform a deep copy using the original table DDL

1. Create a copy of the table using the original CREATE TABLE DDL.
2. Use an INSERT INTO … SELECT statement to populate the copy with data from the original table.
3. Drop the original table.
4. Use an ALTER TABLE statement to rename the copy to the original table name.

The following example performs a deep copy on the SALES table using a duplicate of SALES named SALES_COPY.

```sql
create table salescopy ( ... );
insert into salescopy (select * from sales);
drop table sales;
alter table salescopy rename to sales;
```

To perform a deep copy using CREATE TABLE AS (CTAS)

1. Create a copy of the original table by using CREATE TABLE AS to select the rows from the original table.
2. Drop the original table.
3. Use an ALTER TABLE statement to rename the new table to the original table name.

The following example performs a deep copy on the SALES table using CREATE TABLE AS.

```sql
create table salesas as (select * from sales);
drop table sales;
alter table salesas rename to sales;
```

To perform a deep copy using CREATE TABLE LIKE

1. Create a new table using CREATE TABLE LIKE.
2. Use an INSERT INTO … SELECT statement to copy the rows from the current table to the new table.
3. Drop the current table.
4. Use an ALTER TABLE statement to rename the new table to the original table name.

The following example performs a deep copy on the SALES table using CREATE TABLE LIKE.

```sql
create table likesales (like sales);
insert into likesales (select * from sales);
drop table sales;
alter table likesales rename to sales;
```

To perform a deep copy by creating a temporary table and truncating the original table

1. Use CREATE TABLE AS to create a temporary table with the rows from the original table.
2. Truncate the current table.
3. Use an INSERT INTO … SELECT statement to copy the rows from the temporary table to the original table.
4. Drop the temporary table.

The following example performs a deep copy on the SALES table by creating a temporary table and truncating the original table:
create temp table salestemp as select * from sales;
truncate sales;
insert into sales (select * from salestemp);
drop table tempsales;

Analyzing tables

Topics

- ANALYZE command history (p. 159)
- Automatic analysis of new tables (p. 160)

You should, at regular intervals, update the statistical metadata that the query planner uses to build and choose optimal plans. To do so, you analyze your tables.

You can analyze a table explicitly by running the ANALYZE (p. 268) command. When you load data with the COPY command, you can perform an analysis automatically by setting the STATUPDATE option to ON. By default, the COPY command performs an analysis after it loads data into an empty table. You can force an analysis regardless of whether a table is empty by setting STATUPDATE ON. If you specify STATUPDATE OFF, no analysis is performed.

Only the table owner or a superuser can run the ANALYZE command or run the COPY command with STATUPDATE set to ON.

If you run a query against a new table that was not analyzed after its data was initially loaded, a warning message is displayed; however, no warning occurs when you query a table after a subsequent update or load. The same behavior occurs when you run the EXPLAIN command on a query that contains tables that have not been analyzed.

Whenever adding data to a nonempty table significantly changes the size of the table, we recommend that you update statistics either by running an ANALYZE command or by using the STATUPDATE ON option with the COPY command.

If performance degradation occurs that might be the result of inefficient data storage or a significant change in the statistical profile of the data, run the analysis to see if the updated statistics solve the problem.

To build or update statistics, run the ANALYZE (p. 268) command against:

- The entire current database
- A single table
- One or more specific columns in a single table

The ANALYZE command obtains a sample of rows from the table, does some calculations, and saves resulting column statistics. By default, Amazon Redshift runs a sample pass for the DISTKEY column and another sample pass for all of the other columns in the table. If you want to generate statistics for a subset of columns, you can specify a comma-separated column list.

ANALYZE operations are resource intensive, so run them only on tables and columns that actually require statistics updates. You do not need to analyze all columns in all tables regularly or on the same schedule. If the data changes substantially, analyze the columns that are frequently used in the following:

- Sorting and grouping operations
- Joins
Columns that are less likely to require frequent analysis are those that represent facts and measures and any related attributes that are never actually queried, such as large VARCHAR columns. For example, consider the LISTING table in the TICKIT database:

```
select "column", type, encoding, distkey, sortkey
from pg_table_def where tablename = 'listing';
```

<table>
<thead>
<tr>
<th>column</th>
<th>type</th>
<th>encoding</th>
<th>distkey</th>
<th>sortkey</th>
</tr>
</thead>
<tbody>
<tr>
<td>listid</td>
<td>integer</td>
<td>none</td>
<td>t</td>
<td>1</td>
</tr>
<tr>
<td>sellerid</td>
<td>integer</td>
<td>none</td>
<td>f</td>
<td>0</td>
</tr>
<tr>
<td>eventid</td>
<td>integer</td>
<td>mostly16</td>
<td>f</td>
<td>0</td>
</tr>
<tr>
<td>dateid</td>
<td>smallint</td>
<td>none</td>
<td>f</td>
<td>0</td>
</tr>
<tr>
<td>numtickets</td>
<td>smallint</td>
<td>mostly8</td>
<td>f</td>
<td>0</td>
</tr>
<tr>
<td>priceperticket</td>
<td>numeric(8,2)</td>
<td>bytedict</td>
<td>f</td>
<td>0</td>
</tr>
<tr>
<td>totalprice</td>
<td>numeric(8,2)</td>
<td>mostly32</td>
<td>f</td>
<td>0</td>
</tr>
<tr>
<td>listtime</td>
<td>timestamp with...</td>
<td>none</td>
<td>f</td>
<td>0</td>
</tr>
</tbody>
</table>

If this table is loaded every day with a large number of new records, the LISTID column, which is frequently used in queries as a join key, would need to be analyzed regularly. If TOTALPRICE and LISTTIME are the frequently used constraints in queries, you could analyze those columns and the distribution key on every weekday:

```
analyze listing(listid, totalprice, listtime);
```

If the sellers and events in the application are much more static, and the date IDs refer to a fixed set of days covering only two or three years, the unique values for these columns will not change significantly, although the number of instances of each unique value will increase steadily. In addition, if the NUMTICKETS and PRICEPERTICKET measures are queried infrequently compared to the TOTALPRICE column, you could run the ANALYZE command on the whole table once every weekend to update statistics for the five columns that are not analyzed daily:

```
analyze listing;
```

To maintain current statistics for tables, do the following:

- Run the ANALYZE command before running queries.
- Run the ANALYZE command against the database routinely at the end of every regular load or update cycle.
- Run the ANALYZE command against any new tables that you create and any existing tables or columns that undergo significant change.
- Consider running ANALYZE operations on different schedules for different types of tables and columns, depending on their use in queries and their propensity to change.

## ANALYZE command history

It is useful to know when the last ANALYZE command was run on a table or database. When an ANALYZE command is run, Amazon Redshift executes multiple queries that look like this:
To find out when ANALYZE commands were run, you can query system tables and views such as `STL_QUERY` and `SVL_STATEMENTTEXT` and include a restriction on `padb_fetch_sample`. For example, to find out when the `SALES` table was last analyzed, run this query:

```sql
select query, rtrim(querytxt), starttime
from stl_query
where querytxt like 'padb_fetch_sample%' and querytxt like '%sales%'
order by query desc;
```

<table>
<thead>
<tr>
<th>query</th>
<th>rtrim</th>
<th>starttime</th>
</tr>
</thead>
<tbody>
<tr>
<td>padb_fetch_sample: select * from sales</td>
<td></td>
<td>2012-04-18 12:...</td>
</tr>
<tr>
<td>padb_fetch_sample: select * from sales</td>
<td></td>
<td>2012-04-18 12:...</td>
</tr>
<tr>
<td>padb_fetch_sample: select count(*) from sales</td>
<td></td>
<td>2012-04-18 12:...</td>
</tr>
</tbody>
</table>

(3 rows)

Alternatively, you can run a more complex query that returns all the statements that ran in every completed transaction that included an ANALYZE command:

```sql
select xid, to_char(starttime, 'HH24:MM:SS.MS') as starttime,
date_diff('sec',starttime,endtime ) as secs, substring(text, 1, 40)
from svl_statementtext
where sequence = 0
and xid in (select xid from svl_statementtext s where s.text like 'padb_fetch_sample%')
order by xid desc, starttime;
```

<table>
<thead>
<tr>
<th>xid</th>
<th>starttime</th>
<th>secs</th>
<th>substring</th>
</tr>
</thead>
<tbody>
<tr>
<td>1338</td>
<td>12:04:28.511</td>
<td>4</td>
<td>Analyze date</td>
</tr>
<tr>
<td>1338</td>
<td>12:04:28.511</td>
<td>1</td>
<td>padb_fetch_sample: select count(*) from sales</td>
</tr>
<tr>
<td>1338</td>
<td>12:04:29.443</td>
<td>2</td>
<td>padb_fetch_sample: select * from date</td>
</tr>
<tr>
<td>1338</td>
<td>12:04:31.456</td>
<td>1</td>
<td>padb_fetch_sample: select * from date</td>
</tr>
<tr>
<td>1337</td>
<td>12:04:24.388</td>
<td>1</td>
<td>padb_fetch_sample: select count(*) from sales</td>
</tr>
<tr>
<td>1337</td>
<td>12:04:24.388</td>
<td>4</td>
<td>Analyze sales</td>
</tr>
<tr>
<td>1337</td>
<td>12:04:25.322</td>
<td>2</td>
<td>padb_fetch_sample: select * from sales</td>
</tr>
<tr>
<td>1337</td>
<td>12:04:27.363</td>
<td>1</td>
<td>padb_fetch_sample: select * from sales</td>
</tr>
</tbody>
</table>

(3 rows)

### Automatic analysis of new tables

Amazon Redshift automatically analyzes tables that you create with the following commands:

- CREATE TABLE AS (CTAS)
- CREATE TEMP TABLE AS
- SELECT INTO

You do not need to run the ANALYZE command on these tables when they are first created. If you modify them, you should analyze them in the same way as other tables.
Vacuuming tables

Topics
- VACUUM frequency (p. 161)
- Sort stage and merge stage (p. 161)
- Vacuum types (p. 162)
- Managing vacuum times (p. 162)
- Vacuum column limit exceeded error (p. 168)

To clean up tables after a bulk delete, a load, or a series of incremental updates, you need to run the VACUUM (p. 411) command, either against the entire database or against individual tables.

Note
Only the table owner or a superuser can effectively vacuum a table. A VACUUM for a single table will fail if you do not have owner or superuser privileges for the table. If you run a VACUUM of the entire database, without specifying a table name, the operation completes successfully but has no effect on tables for which you do not have owner or superuser privileges. For this reason, and because vacuuming the entire database is potentially an expensive operation, we recommend vacuuming individual tables as needed.

Amazon Redshift does not automatically reclaim and reuse space that is freed when you delete rows and update rows. To perform an update, Amazon Redshift deletes the original row and appends the updated row, so every update is effectively a delete followed by an insert. When you perform a delete, the rows are marked for deletion, but not removed. The query processor needs to scan the deleted rows as well as undeleted rows, so too many deleted rows can cost unnecessary processing. You should vacuum following a significant number of deletes or updates to reclaim space and improve query performance.

For tables with a sort key, the VACUUM command ensures that new data in tables is fully sorted on disk. When data is initially loaded into a table that has a sort key, the data is sorted according to the SORTKEY specification in the CREATE TABLE (p. 310) statement. However, when you update the table, using COPY, INSERT, or UPDATE statements, new rows are stored in a separate unsorted region on disk, then sorted on demand for queries as required. If large numbers of rows remain unsorted on disk, query performance might be degraded for operations that rely on sorted data, such as range-restricted scans or merge joins. The VACUUM command merges new rows with existing sorted rows, so range-restricted scans are more efficient and the execution engine doesn't need to sort rows on demand during query execution.

**VACUUM frequency**

You should vacuum as often as you need to in order to maintain consistent query performance. Consider these factors when determining how often to run your VACUUM command.

- Run VACUUM during maintenance windows or time periods when you expect minimal activity on the cluster.
- A large unsorted region results in longer vacuum times. If you delay vacuuming, the vacuum will take longer because more data has to be reorganized.
- VACUUM is an I/O intensive operation, so the longer it takes for your vacuum to complete, the more impact it will have on concurrent queries and other database operations running on your cluster.

**Sort stage and merge stage**

Amazon Redshift performs a vacuum operation in two stages: first, it sorts the rows in the unsorted region, then, if necessary, it merges the newly sorted rows at the end of the table with the existing rows. When vacuuming a large table, the vacuum operation proceeds in a series of steps consisting of incremental
sorts followed by merges. If the operation fails or if Amazon Redshift goes off line during the vacuum, the partially vacuumed table or database will be in a consistent state, but you will need to manually restart the vacuum operation. Incremental sorts are lost, but merged rows that were committed before the failure do not need to be vacuumed again. If the unsorted region is large, the lost time might be significant. For more information about the sort and merge stages, see Managing the volume of merged rows (p. 163).

Users can access tables while they are being vacuumed. You can perform queries and write operations while a table is being vacuumed, but when DML and a vacuum run concurrently, both might take longer. If you execute UPDATE and DELETE statements during a vacuum, system performance might be reduced. Incremental merges temporarily block concurrent UPDATE and DELETE operations, and UPDATE and DELETE operations in turn temporarily block incremental merge steps on the affected tables. DDL operations, such as ALTER TABLE, are blocked until the vacuum operation finishes with the table.

**Vacuum types**

You can run a full vacuum, a delete only vacuum, or sort only vacuum.

- **Full vacuum**
  
  We recommend a full vacuum for most applications where reclaiming space and resorting rows are equally important. It is more efficient to run a full vacuum than to run back-to-back DELETE ONLY and SORT ONLY vacuum operations. Full vacuum is the default vacuum operation.

- **DELETE ONLY**
  
  DELETE ONLY is the same as a full vacuum except that it skips the sort. A DELETE ONLY vacuum saves time when reclaiming disk space is important but resorting new rows is not. For example, you might perform a DELETE ONLY vacuum operation if you don't need to resort rows to optimize query performance, and compacts the table to eliminate to free up the consumed space.

- **SORT ONLY**
  
  A SORT ONLY vacuum saves some time by not reclaiming disk space, but in most cases there is little benefit compared to a full vacuum.

**Managing vacuum times**

Depending on the nature of your data, we recommend following the practices in this section to minimize vacuum times.

**Topics**

- Managing the size of the unsorted region (p. 162)
- Managing the volume of merged rows (p. 163)
- Loading your data in sort key order (p. 166)
- Using time series tables (p. 167)

**Managing the size of the unsorted region**

The unsorted region grows when you load large amounts of new data into tables that already contain data or when you do not vacuum tables as part of your routine maintenance operations. To avoid long-running vacuum operations, use the following practices:

- Run vacuum operations on a regular schedule.
If you load your tables in small increments (such as daily updates that represent a small percentage of the total number of rows in the table), running VACUUM regularly will help ensure that individual vacuum operations go quickly.

- Run the largest load first.

If you need to load a new table with multiple COPY operations, run the largest load first. When you run an initial load into a new or truncated table, all of the data is loaded directly into the sorted region, so no vacuum is required.

- Truncate a table instead of deleting all of the rows.

Deleting rows from a table does not reclaim the space that the rows occupied until you perform a vacuum operation; however, truncating a table empties the table and reclaims the disk space, so no vacuum is required. Alternatively, drop the table and re-create it.

- Truncate or drop test tables.

If you are loading a small number of rows into a table for test purposes, don’t delete the rows when you are done. Instead, truncate the table and reload those rows as part of the subsequent production load operation.

- Perform a deep copy.

A deep copy recreates and repopulates a table by using a bulk insert, which automatically resorts the table. If a table has a large unsorted region, a deep copy is much faster than a vacuum. The trade off is that you cannot make concurrent updates during a deep copy operation, which you can do during a vacuum. For more information, see Avoid long vacuums by using a deep copy (p. 33).

**Managing the volume of merged rows**

If a vacuum operation needs to merge new rows into a table's sorted region, the time required for a vacuum will increase as the table grows larger. You can improve vacuum performance by reducing the number of rows that must be merged.

Prior to a vacuum, a table consists of a sorted region at the head of the table, followed by an unsorted region, which grows whenever rows are added or updated. When a set of rows is added by a COPY operation, the new set of rows is sorted on the sort key as it is added to the unsorted region at the end of the table. The new rows are ordered within their own set, but not within the unsorted region.

The following diagram illustrates the unsorted region after two successive COPY operations, where the sort key is CUSTID.
A vacuum restores the table's sort order in two stages:

1. Sort the unsorted region into a newly-sorted region.

   The first stage is relatively cheap, because only the unsorted region is rewritten. If the range of sort key values of the newly-sorted region is higher than the existing range, only the new rows need to be rewritten, and the vacuum is complete. For example, if the sorted region contains ID values 1 to 500 and subsequent copy operations add key values greater than 500, then only the unsorted region only needs to be rewritten.

2. Merge the newly-sorted region with the previously-sorted region.

   If the keys in the newly sorted region overlap the keys in the sorted region, then VACUUM needs to merge the rows. Starting at the beginning of the newly-sorted region (at the lowest sort key), the vacuum writes the merged rows from the previously sorted region and the newly sorted region into a new set of blocks.

The extent to which the new sort key range overlaps the existing sort keys determines the extent to which the previously-sorted region will need to be rewritten. If the unsorted keys are scattered throughout the existing sort range, a vacuum might need to rewrite existing portions of the table.

The following diagram shows how a vacuum would sort and merge rows that are added to a table where CUSTID is the sort key. Because each copy operation adds a new set of rows with key values that overlap the existing keys, almost the entire table needs to be rewritten. The diagram shows single sort and merge, but in practice, a large vacuum consists of a series of incremental sort and merge steps.
If the range of sort keys in a set of new rows overlaps the range of existing keys, the cost of the merge stage continues to grow in proportion to the table size as the table grows while the cost of the sort stage remains proportional to the size of the unsorted region. In such a case, the cost of the merge stage overshadows the cost of the sort stage, as the following diagram shows.
To determine what proportion of a table was remerged, query `SVV_VACUUM_SUMMARY` after the vacuum operation completes. The following query shows the effect of six successive vacuums as `CUSTSALES` grew larger over time.

```sql
select * from svv_vacuum_summary
where table_name = 'custsales';
```

<table>
<thead>
<tr>
<th>table_name</th>
<th>xid</th>
<th>sort_</th>
<th>merge_</th>
<th>elapsed_</th>
<th>row_</th>
<th>sortedrow_</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>block_</td>
<td>max_merge_</td>
<td>partitions</td>
<td>increments</td>
<td>time</td>
<td>delta</td>
</tr>
<tr>
<td></td>
<td>delta</td>
<td>partitions</td>
<td>+-----------------+-----------------+-----+-----</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>custsales</td>
<td>7072</td>
<td>3</td>
<td>2</td>
<td>143918314</td>
<td>0</td>
<td>88297472</td>
</tr>
<tr>
<td></td>
<td>1524</td>
<td>47</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>custsales</td>
<td>7122</td>
<td>3</td>
<td>3</td>
<td>164157882</td>
<td>0</td>
<td>88297472</td>
</tr>
<tr>
<td></td>
<td>772</td>
<td>47</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>custsales</td>
<td>7212</td>
<td>3</td>
<td>4</td>
<td>187433171</td>
<td>0</td>
<td>88297472</td>
</tr>
<tr>
<td></td>
<td>767</td>
<td>47</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>custsales</td>
<td>7289</td>
<td>3</td>
<td>4</td>
<td>255482945</td>
<td>0</td>
<td>88297472</td>
</tr>
<tr>
<td></td>
<td>770</td>
<td>47</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>custsales</td>
<td>7420</td>
<td>3</td>
<td>5</td>
<td>316583833</td>
<td>0</td>
<td>88297472</td>
</tr>
<tr>
<td></td>
<td>769</td>
<td>47</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>custsales</td>
<td>9007</td>
<td>3</td>
<td>6</td>
<td>306685472</td>
<td>0</td>
<td>88297472</td>
</tr>
<tr>
<td></td>
<td>772</td>
<td>47</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(6 rows)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

The `merge_increments` column gives an indication of the amount of data that was merged for each vacuum operation. If the number of merge increments over consecutive vacuums increases in proportion to the growth in table size, that is an indication that each vacuum operation is remerging an increasing number of rows in the table because the existing and newly sorted regions overlap.

### Loading your data in sort key order

If you load your data in sort key order, you might reduce or even eliminate the need to vacuum. For example, suppose you have a table that records customer events using a customer ID and time. If you sort on customer ID, it's likely that the sort key range of new rows added by incremental loads will overlap the existing range, as shown in the previous example, leading to an expensive vacuum operation.

If you set your sort key to a timestamp column, your new rows will be appended in sort order at the end of the table, as the following diagram shows, reducing or even eliminating the need to vacuum.
Using time series tables

If you maintain data for a rolling time period, use a series of tables, as the following diagram illustrates.

Create a new table each time you add a set of data, then delete the oldest table in the series. You gain a double benefit:

- You avoid the added cost of deleting rows, because a DROP TABLE operation is much more efficient than a mass DELETE.
- If the tables are sorted by timestamp, no vacuum is needed. If each table contains data for one month, a vacuum will at most have to rewrite one month’s worth of data, even if the tables are not sorted by timestamp.

You can create a UNION ALL view for use by reporting queries that hides the fact that the data is stored in multiple tables. If a query filters on the sort key, the query planner can efficiently skip all the tables that
aren’t used. A UNION ALL can be less efficient for other types of queries, so you should evaluate query performance in the context of all queries that use the tables.

Vacuum column limit exceeded error

If your vacuum fails with the message **ERROR: 1036 DETAIL: Vacuum column limit exceeded**, your table has more columns than VACUUM can process with the available memory. The vacuum column limit is less than the maximum number of columns for a table, which is 1600. The actual column limit for a vacuum varies depending on your cluster’s configuration. You can increase the vacuum column limit by increasing the value of `wlm_query_slot_count` (p. 697), which increases the amount of memory available for the query. The maximum value for `wlm_query_slot_count` is limited to the concurrency value for the queue. For more information, see Increase the available memory (p. 33).

If increasing the value of `wlm_query_slot_count` is not an option, or if it doesn’t solve the problem, you can avoid needing to vacuum by performing a deep copy. To perform a deep copy, you create a copy of the table, insert the rows from the original table into the copy, drop the original table, and then rename the copy. A deep copy is often much faster than a vacuum. For more information, see Performing a deep copy (p. 156).

For example, suppose the table `calendardays` has 365 columns. After a load operation, you perform a vacuum and the vacuum fails, as the following example shows.

```
vacuum calendardays;
```

An error occurred when executing the SQL command:
```
vacuum calendardays;
```

**ERROR: 1036**
**DETAIL:** Vacuum column limit exceeded for table calendardays
**HINT:** Increase the value of `wlm_query_slot_count` or perform a deep copy instead of a vacuum.

The following example sets `wlm_query_slot_count` to 10, performs a vacuum, and then resets `wlm_query_slot_count` to 1. With the higher slot count, the vacuum succeeds.

```
set wlm_query_slot_count to 10;
vacuum calendardays;
set wlm_query_slot_count to 1;
vacuum executed successfully
```

You can perform a deep copy instead of a vacuum. The following example uses CREATE TABLE LIKE to perform a deep copy.

```
cREATE TABLE LIKE calendardays (LIKE calendardays);
INSERT INTO calendardays (SELECT * FROM calendardays);
DROP TABLE calendardays;
ALTER TABLE calendardays RENAME TO calendardays;
```

Performing a deep copy using CREATE TABLE AS (CTAS) is faster than using CREATE TABLE LIKE, but CTAS does not preserve the sort key, encoding, distkey, and notnull attributes of the parent table. For a comparison of different deep copy methods, see Performing a deep copy (p. 156).
Managing concurrent write operations

Topics
- Serializable isolation (p. 169)
- Write and read-write operations (p. 170)
- Concurrent write examples (p. 171)

Amazon Redshift allows tables to be read while they are being incrementally loaded or modified.

In some traditional data warehousing and business intelligence applications, the database is available to users only when the nightly load is complete. In such cases, no updates are allowed during regular work hours, when analytic queries are run and reports are generated; however, an increasing number of applications remain live for long periods of the day or even all day, making the notion of a load window obsolete.

Amazon Redshift supports these types of applications by allowing tables to be read while they are being incrementally loaded or modified. Queries simply see the latest committed version, or snapshot, of the data, rather than waiting for the next version to be committed. If you want a particular query to wait for a commit from another write operation, you have to schedule it accordingly.

The following topics describe some of the key concepts and use cases that involve transactions, database snapshots, updates, and concurrent behavior.

Serializable isolation

Some applications require not only concurrent querying and loading, but also the ability to write to multiple tables or the same table concurrently. In this context, concurrently means overlapping, not scheduled to run at precisely the same time. Two transactions are considered to be concurrent if the second one starts before the first commits. Concurrent operations can originate from different sessions that are controlled either by the same user or by different users.

Note
Amazon Redshift supports a default automatic commit behavior in which each separately-executed SQL command commits individually. If you enclose a set of commands in a transaction block (defined by BEGIN (p. 270) and END (p. 339) statements), the block commits as one transaction, so you can roll it back if necessary. An exception to this behavior is the TRUNCATE command, which automatically commits all outstanding changes made in the current transaction without requiring an END statement.

Concurrent write operations are supported in Amazon Redshift in a protective way, using write locks on tables and the principle of serializable isolation. Serializable isolation preserves the illusion that a transaction running against a table is the only transaction that is running against that table. For example, two concurrently running transactions, T1 and T2, must produce the same results as at least one of the following:

- T1 and T2 run serially in that order
- T2 and T1 run serially in that order

Concurrent transactions are invisible to each other; they cannot detect each other's changes. Each concurrent transaction will create a snapshot of the database at the beginning of the transaction. A database snapshot is created within a transaction on the first occurrence of most SELECT statements, DML commands such as COPY, DELETE, INSERT, UPDATE, and TRUNCATE, and the following DDL commands:

- ALTER TABLE (to add or drop columns)
CREATE TABLE
DROP TABLE
TRUNCATE TABLE

If any serial execution of the concurrent transactions would produce the same results as their concurrent execution, those transactions are deemed “serializable” and can be run safely. If no serial execution of those transactions would produce the same results, the transaction that executes a statement that would break serializability is aborted and rolled back.

System catalog tables (PG) and other Amazon Redshift system tables (STL and STV) are not locked in a transaction; therefore, changes to database objects that arise from DDL and TRUNCATE operations are visible on commit to any concurrent transactions.

For example, suppose that table A exists in the database when two concurrent transactions, T1 and T2, start. If T2 returns a list of tables by selecting from the PG_TABLES catalog table, and then T1 drops table A and commits, and then T2 lists the tables again, table A is no longer listed. If T2 tries to query the dropped table, Amazon Redshift returns a “relation does not exist” error. The catalog query that returns the list of tables to T2 or checks that table A exists is not subject to the same isolation rules as operations against user tables.

Transactions for updates to these tables run in a read committed isolation mode. PG-prefix catalog tables do not support snapshot isolation.

Serializable isolation for system tables and catalog tables

A database snapshot is also created in a transaction for any SELECT query that references a user-created table or Amazon Redshift system table (STL or STV). SELECT queries that do not reference any table will not create a new transaction database snapshot, nor will any INSERT, DELETE, or UPDATE statements that operate solely on system catalog tables (PG).

Write and read-write operations

You can manage the specific behavior of concurrent write operations by deciding when and how to run different types of commands. The following commands are relevant to this discussion:

- COPY commands, which perform loads (initial or incremental)
- INSERT commands that append one or more rows at a time
- UPDATE commands, which modify existing rows
- DELETE commands, which remove rows

COPY and INSERT operations are pure write operations, but DELETE and UPDATE operations are read-write operations. (In order for rows to be deleted or updated, they have to be read first.) The results of concurrent write operations depend on the specific commands that are being run concurrently. COPY and INSERT operations against the same table are held in a wait state until the lock is released, then they proceed as normal.

UPDATE and DELETE operations behave differently because they rely on an initial table read before they do any writes. Given that concurrent transactions are invisible to each other, both UPDATEs and DELETEs have to read a snapshot of the data from the last commit. When the first UPDATE or DELETE releases its lock, the second UPDATE or DELETE needs to determine whether the data that it is going to work with is potentially stale. It will not be stale, because the second transaction does not obtain its snapshot of data until after the first transaction has released its lock.
Potential deadlock situation for concurrent write transactions

Whenever transactions involve updates of more than one table, there is always the possibility of concurrently-running transactions becoming deadlocked when they both try to write to the same set of tables. A transaction releases all of its table locks at once when it either commits or rolls back; it does not relinquish locks one at a time.

For example, suppose that transactions T1 and T2 start at roughly the same time. If T1 starts writing to table A and T2 starts writing to table B, both transactions can proceed without conflict; however, if T1 finishes writing to table A and needs to start writing to table B, it will not be able to proceed because T2 still holds the lock on B. Conversely, if T2 finishes writing to table B and needs to start writing to table A, it will not be able to proceed either because T1 still holds the lock on A. Because neither transaction can release its locks until all its write operations are committed, neither transaction can proceed.

In order to avoid this kind of deadlock, you need to schedule concurrent write operations carefully. For example, you should always update tables in the same order in transactions and, if specifying locks, lock tables in the same order before you perform any DML operations.

Concurrent write examples

The following examples demonstrate how transactions either proceed or abort and roll back when they are run concurrently.

Concurrent COPY operations into the same table

Transaction 1 copies rows into the LISTING table:

```
begin;
  copy listing from ...
end;
```

Transaction 2 starts concurrently in a separate session and attempts to copy more rows into the LISTING table. Transaction 2 must wait until transaction 1 releases the write lock on the LISTING table, then it can proceed.

```
begin;
  [waits]
  copy listing from
end;
```

The same behavior would occur if one or both transactions contained an INSERT command instead of a COPY command.

Concurrent DELETE operations from the same table

Transaction 1 deletes rows from a table:

```
begin;
  delete from listing where ...
end;
```

Transaction 2 starts concurrently and attempts to delete rows from the same table. It will succeed because it waits for transaction 1 to complete before attempting to delete rows.
The same behavior would occur if one or both transactions contained an UPDATE command to the same table instead of a DELETE command.

**Concurrent transactions with a mixture of read and write operations**

In this example, transaction 1 deletes rows from the USERS table, reloads the table, runs a COUNT(*) query, and then ANALYZE, before committing:

```sql
begin
[waits]
delete from listing where ;
end;
```

Meanwhile, transaction 2 starts. This transaction attempts to copy additional rows into the USERS table, analyze the table, and then run the same COUNT(*) query as the first transaction:

```sql
begin;
delete one row from USERS table;
copy ;
select count(*) from users;
analyze ;
end;
```

The second transaction will succeed because it must wait for the first to complete. Its COUNT query will return the count based on the load it has completed.
Unloading Data

Topics

• Unloading data to Amazon S3 (p. 173)
• Unloading encrypted data files (p. 176)
• Unloading data in delimited or fixed-width format (p. 177)
• Reloading unloaded data (p. 178)

To unload data from database tables to a set of files in an Amazon S3 bucket, you can use the UNLOAD (p. 395) command with a SELECT statement. You can unload text data in either delimited format or fixed-width format, regardless of the data format that was used to load it. You can also specify whether to create compressed GZIP files.

You can limit the access users have to your Amazon S3 bucket by using temporary security credentials.

Important

The Amazon S3 bucket where Amazon Redshift will write the output files must be created in the same region as your cluster.

Unloading data to Amazon S3

Amazon Redshift splits the results of a select statement across a set of files, one or more files per node slice, to simplify parallel reloading of the data. Alternatively, you can specify that UNLOAD should write the results serially to one or more files by adding the PARALLEL OFF option.

You can use any select statement in the UNLOAD command that Amazon Redshift supports, except for a select that uses a LIMIT clause in the outer select. For example, you can use a select statement that includes specific columns or that uses a where clause to join multiple tables. If your query contains quotes (enclosing literal values, for example), you need to escape them in the query text (\'\'). For more information, see the SELECT (p. 359) command reference. For more information about using a LIMIT clause, see the Usage notes (p. 398) for the UNLOAD command.

For example, the following UNLOAD command sends the contents of the VENUE table to the Amazon S3 bucket s3://mybucket/ticket/unload/.
If you include a prefix in the Amazon S3 path string, UNLOAD will use that prefix for the file names.

```
unload ('select * from venue')
to 's3://mybucket/ticket/unload/' credentials
    'aws_access_key_id=<access-key-id>;aws_secret_access_key=<secret-access-key>'
```

The file names created by the previous example include the prefix 'venue_'.

```
venue_0000_part_00
venue_0001_part_00
venue_0002_part_00
venue_0003_part_00
```

By default, UNLOAD writes data in parallel to multiple files, according to the number of slices in the cluster. To write data to a single file, specify PARALLEL OFF. UNLOAD writes the data serially, sorted absolutely according to the ORDER BY clause, if one is used. The maximum size for a data file is 6.2 GB. If the data size is greater than the maximum, UNLOAD creates additional files, up to 6.2 GB each.

The following example writes the contents VENUE to a single file. Only one file is required because the file size is less than 6.2 GB.

```
unload ('select * from venue')
to 's3://mybucket/ticket/unload/venue_' credentials
    'aws_access_key_id=<access-key-id>;aws_secret_access_key=<secret-access-key>'
    parallel off;
```

**Note**
The UNLOAD command is designed to use parallel processing. We recommend leaving PARALLEL enabled for most cases, especially if the files will be used to load tables using a COPY command.

You can limit the access users have to your data by using temporary security credentials. Temporary security credentials provide enhanced security because they have short life spans and cannot be reused after they expire. A user who has these temporary security credentials can access your resources only until the credentials expire. For more information, see Temporary security credentials (p. 290). To unload data using temporary access credentials, use the following syntax:

```
unload ('select * from venue')
to 's3://mybucket/ticket/unload/' credentials
    'aws_access_key_id=<temporary-access-key-id>;aws_secret_access_key=<temporary-secret-access-key>;token=<temporary-token>'
```

**Important**
The temporary security credentials must be valid for the entire duration of the COPY statement. If the temporary security credentials expire during the load process, the COPY will fail and the transaction will be rolled back. For example, if temporary security credentials expire after 15 minutes and the COPY requires one hour, the COPY will fail before it completes.
You can create a manifest file that lists the unload files by specifying the MANIFEST option in the UNLOAD command. The manifest is a text file in JSON format that explicitly lists the URL of each file that was written to Amazon S3.

The following example includes the manifest option.

```sql
unload ('select * from venue)
to 's3://mybucket/ticket/venue_' credentials
'aws_access_key_id=<access-key-id>;aws_secret_access_key=<secret-access-key>'
manifest;
```

The following example shows a manifest for four unload files.

```json
{
  "entries": [
    {"url":"s3://mybucket/ticket/venue_0000_part_00"},
    {"url":"s3://mybucket/ticket/venue_0001_part_00"},
    {"url":"s3://mybucket/ticket/venue_0002_part_00"},
    {"url":"s3://mybucket/ticket/venue_0003_part_00"}
  ]
}
```

The manifest file can be used to load the same files by using a COPY with the MANIFEST option. For more information, see Using a manifest to specify data files (p. 125).

After you complete an UNLOAD operation, confirm that the data was unloaded correctly by navigating to the Amazon S3 bucket where UNLOAD wrote the files. You will see one or more numbered files per slice, starting with the number zero. If you specified the MANIFEST option, you will also see a file ending with 'manifest'. For example:

```sql
mybucket/ticket/venue_0000_part_00
mybucket/ticket/venue_0001_part_00
mybucket/ticket/venue_0002_part_00
mybucket/ticket/venue_0003_part_00
mybucket/ticket/venue_manifest
```

You can programmatically get a list of the files that were written to Amazon S3 by calling an Amazon S3 list operation after the UNLOAD completes; however, depending on how quickly you issue the call, the list might be incomplete because an Amazon S3 list operation is eventually consistent. To get a complete, authoritative list immediately, query STL_UNLOAD_LOG.

The following query returns the pathname for files that were created by an UNLOAD. The `PG_LAST_QUERY_ID()` (p. 576) function returns the most recent query.

```sql
select query, substring(path,0,40) as path
from stl_unload_log
where query=2320
order by path;
```

<table>
<thead>
<tr>
<th>query</th>
<th>path</th>
</tr>
</thead>
<tbody>
<tr>
<td>2320</td>
<td>s3://my-bucket/venue0000_part_00</td>
</tr>
<tr>
<td>2320</td>
<td>s3://my-bucket/venue0001_part_00</td>
</tr>
<tr>
<td>2320</td>
<td>s3://my-bucket/venue0002_part_00</td>
</tr>
</tbody>
</table>
If the amount of data is very large, Amazon Redshift might split the files into multiple parts per slice. For example:

venue_0000_part_00
venue_0000_part_01
venue_0000_part_02
venue_0000_part_00
venue_0000_part_01
venue_0000_part_02
...

The following UNLOAD command includes a quoted string in the select statement, so the quotes are escaped (=\"'OH\'\" ).

```
unload ('select venuename, venuecity from venue where venuestate=\'OH\'')
to 's3://mybucket/tickit/venue/ ' credentials
'aws_access_key_id=<access-key-id>;aws_secret_access_key=<secret-access-key>'
```

By default, UNLOAD will fail rather than overwrite existing files in the destination bucket. To overwrite the existing files, including the manifest file, specify the ALLOWOVERWRITE option.

```
unload ('select * from venue')
to 's3://mybucket/venue_pipe_' credentials
'aws_access_key_id=<access-key-id>;<aws_secret_access_key=<secret-access-key>'
manifest
allowoverwrite;
```

Unloading encrypted data files

You can create encrypted data files in Amazon S3 by using the UNLOAD command with the ENCRYPTED option. UNLOAD uses the same envelope encryption process that Amazon S3 client-side encryption uses. You can then use the COPY command with the ENCRYPTED option to load the encrypted files.

The process works like this:

1. You create a base64 encoded 256-bit AES key that you will use as your private encryption key, or master symmetric key.
2. You issue an UNLOAD command that includes your master symmetric key and the ENCRYPTED option.
3. UNLOAD generates a one-time-use symmetric key (called the envelope symmetric key) and an initialization vector (IV), which it uses to encrypt your data.
4. UNLOAD encrypts the envelope symmetric key using your master symmetric key.
5. UNLOAD then stores the encrypted data files in Amazon S3 and stores the encrypted envelope key and IV as object metadata with each file. The encrypted envelope key is stored as object metadata x-amz-meta-x-amz-key and the IV is stored as object metadata x-amz-meta-x-amz-iv.

For more information about the envelope encryption process, see the Client-Side Data Encryption with the AWS SDK for Java and Amazon S3 article.
To unload encrypted data files, add the master key value to the credentials string and include the ENCRYPTED option.

**Note**
If you use the MANIFEST option, the manifest file is not encrypted.

```sql
unload ('select venuename, venuecity from venue')
to 's3://mybucket/encrypted/venue_
credentials 'aws_access_key_id=<access-key-id>;aws_secret_access_key=<secret-access-key>;master_symmetric_key=<master_key>'
manifest
encrypted;
```

To unload encrypted data files that are GZIP compressed, include the GZIP option along with the master key value and the ENCRYPTED option.

```sql
unload ('select venuename, venuecity from venue')
to 's3://mybucket/encrypted/venue_
credentials 'aws_access_key_id=<access-key-id>;aws_secret_access_key=<secret-access-key>;master_symmetric_key=<master_key>'
encrypted gzip;
```

To load the encrypted data files, add the same master key value to the credentials string and include the ENCRYPTED option.

```sql
copy venue from 's3://mybucket/encrypted/venue_
credentials 'aws_access_key_id=<access-key-id>;aws_secret_access_key=<secret-access-key>;master_symmetric_key=<master_key>
encrypted;
```

---

**Unloading data in delimited or fixed-width format**

You can unload data in delimited format or fixed-width format. The default output is pipe-delimited (using the '|' character).

The following example specifies a comma as the delimiter:

```sql
unload ('select * from venue')
to 's3://mybucket/ticket/venue/comma'
credentials 'aws_access_key_id=<access-key-id>;aws_secret_access_key=<secret-access-key>'
delimiter ',';
```

The resulting output files look like this:

```
20,Air Canada Centre,Toronto,ON,0
60,Rexall Place,Edmonton,AB,0
100,U.S. Cellular Field,Chicago,IL,40615
200,Al Hirschfeld Theatre,New York City,NY,0
240,San Jose Repertory Theatre,San Jose,CA,0
300,Kennedy Center Opera House,Washington,DC,0
...```
To unload the same result set to a tab-delimited file, issue the following command:

```
unload ('select * from venue')
to 's3://mybucket/ticket/venue/tab'
credentials 'aws_access_key_id=<access-key-id>;aws_secret_access_key=<secret-access-key>'
delimiter as '\t';
```

Alternatively, you can use a FIXEDWIDTH specification. This specification consists of an identifier for each table column and the width of the column (number of characters). The UNLOAD command will fail rather than truncate data, so specify a width that is at least as long as the longest entry for that column. Unloading fixed-width data works similarly to unloading delimited data, except that the resulting output contains no delimiting characters. For example:

```
unload ('select * from venue')
to 's3://mybucket/ticket/venue/fw'
credentials 'aws_access_key_id=<access-key-id>;aws_secret_access_key=<secret-access-key>'
fixedwidth '0:3,1:100,2:30,3:2,4:6';
```

The fixed-width output looks like this:

```
20  Air Canada Centre       Toronto      ON0
60  Rexall Place           Edmonton     AB0
100U.S. Cellular Field     Chicago      IL40615
2001  Hirschfeld Theatre  New York CityNY0
240San Jose Repertory TheatreSan Jose     CA0
300Kennedy Center Opera HouseWashington DC0
```

For more details about FIXEDWIDTH specifications, see the `COPY (p. 276)` command.

## Reloading unloaded data

To reload the results of an unload operation, you can use a COPY command.

The following example shows a simple case in which the VENUE table is unloaded using a manifest file, truncated, and reloaded.

```
unload ('select * from venue order by venueid')
to 's3://mybucket/ticket/venue/reload_' credentials
'aws_access_key_id=<access-key-id>;aws_secret_access_key=<secret-access-key>'
manifest
delimiter '|');
truncate venue;

copy venue
from 's3://mybucket/ticket/venue/reload_manifest' credentials
'aws_access_key_id=<access-key-id>;aws_secret_access_key=<secret-access-key>'
manifest
delimiter '|');
```
After it is reloaded, the VENUE table looks like this:

```sql
select * from venue order by venueid limit 5;

<table>
<thead>
<tr>
<th>venueid</th>
<th>venuename</th>
<th>venuecity</th>
<th>venuestate</th>
<th>venueseats</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Toyota Park</td>
<td>Bridgeview</td>
<td>IL</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>Columbus Crew Stadium</td>
<td>Columbus</td>
<td>OH</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>RFK Stadium</td>
<td>Washington</td>
<td>DC</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>CommunityAmerica Ballpark</td>
<td>Kansas City</td>
<td>KS</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>Gillette Stadium</td>
<td>Foxborough</td>
<td>MA</td>
<td>68756</td>
</tr>
</tbody>
</table>
```

(5 rows)
Tuning Query Performance

Topics

- Troubleshooting Queries (p. 181)
- Analyzing the Explain Plan (p. 184)
- Managing how queries use memory (p. 190)
- Monitoring disk space (p. 194)
- Benchmarking with compiled code (p. 195)
- Setting the JDBC fetch size parameter (p. 196)
- Implementing workload management (p. 196)

Retrieving useful information from an Amazon Redshift data warehouse involves executing very complex queries against extremely large amounts of data. As a result, many queries take a very long time to run. Amazon Redshift uses massively parallel processing along with a sophisticated query planner to enable much faster execution of analytic queries. If you designed and built your database according to the principles in the Designing Tables (p. 90) section of this guide, your queries will take full advantage of parallel processing and should run efficiently without significant tuning. If some of your queries are taking too long to run, there are steps you can take to optimize query performance.

To identify the source of the problems with slow queries, you can use the EXPLAIN command to generate the query execution plan, or explain plan, for a query, and then analyze the plan to locate the operations that use the most resources. With that information, you can begin to look for ways to improve performance.

If a very complex query does not have enough available memory, it might need to write some intermediate data to disk temporarily, which takes extra time. You can often improve performance by identifying whether a query is writing to disk, and then taking steps to improve how it uses memory, or to increase the amount of available memory.

After you have isolated performance problems, you can revisit your table design to consider whether changing the sort keys or distribution keys might improve slow queries.

Sometimes a query that should execute quickly is forced to wait until other, longer-running queries finish. In that case, you might be able to improve overall system performance by creating query queues and assigning different types of queries to the appropriate queues.

This section provides information and examples to help you to optimize the performance of your queries by examining a query’s execution plan, by monitoring how a query uses disk storage, and by using queues to manage query workload.
Troubleshooting Queries

This section provides a quick reference for identifying and addressing some of the most common and most serious issues you are likely to encounter with Amazon Redshift queries.

Topics
- Connection Fails (p. 181)
- Query Hangs (p. 181)
- Query Takes Too Long (p. 182)
- Load Fails (p. 183)
- Load Takes Too Long (p. 183)
- Load Data Is Incorrect (p. 184)

These suggestions give you a starting point for troubleshooting. You can also refer to the following resources for more detailed information.
- Accessing Amazon Redshift Clusters and Databases
- Designing Tables (p. 90)
- Loading Data (p. 117)
- Tutorial: Tuning Table Design (p. 35)
- Tutorial: Loading Data from Amazon S3 (p. 62)

Connection Fails

Your query connection can fail for the following reasons; we suggest the following troubleshooting approaches.

Client cannot connect to server

If you are using SSL or server certificates, first remove this complexity while you troubleshoot the connection issue. Then add SSL or server certificates back when you have found a solution. For more information, go to Configure Security Options for Connections in the Amazon Redshift Cluster Management Guide.

Connection is refused

Generally, when you receive an error message indicating that there is a failure to establish a connection, it means that there is an issue with the permission to access the cluster. For more information, go to The connection is refused or fails in the Amazon Redshift Cluster Management Guide.

Query Hangs

Your query can hang, or stop responding, for the following reasons; we suggest the following troubleshooting approaches.

Connection to the database is dropped
Reduce the size of maximum transmission unit (MTU). The MTU size determines the maximum size, in bytes, of a packet that can be transferred in one Ethernet frame over your network connection. For more information, go to The connection to the database is dropped in the Amazon Redshift Cluster Management Guide.

**Connection to the database times out**

Generally, when you receive an error message indicating that there is a failure to establish a connection, it means that there is an issue with permission to access the cluster. For more information, go to The connection is refused or fails in the Amazon Redshift Cluster Management Guide.

**Client-side out-of-memory error occurs with ODBC**

If your client application uses an ODBC connection and your query creates a result set that is too large to fit in memory, you can stream the result set to your client application by using a cursor. For more information, see DECLARE (p. 328) and Performance considerations when using cursors (p. 330).

**Client-side out-of-memory error occurs with JDBC**

When you attempt to retrieve large result sets over a JDBC connection, you might encounter client-side out-of-memory errors. For more information, see Setting the JDBC fetch size parameter.

**There is a potential deadlock**

If there is a potential deadlock, try the following:

- View the STV_LOCKS (p. 657) and STL_TR_CONFLICT (p. 635) system tables to find conflicts involving updates to more than one table.
- Use the PG_CANCEL_BACKEND (p. 566) function to cancel one or more conflicting queries.
- Use the PG_TERMINATE_BACKEND (p. 566) function to terminate a session, which forces any currently running transactions in the terminated session to release all locks and roll back the transaction.
- Schedule concurrent write operations carefully. For more information, see Managing concurrent write operations (p. 169).

**Query Takes Too Long**

Your query can take too long for the following reasons; we suggest the following troubleshooting approaches.

**Tables are not optimized**

Set the sort key, distribution style, and compression encoding of the tables to take full advantage of parallel processing. For more information, see Designing Tables (p. 90) and Tutorial: Tuning Table Design (p. 35).

**Query is writing to disk**

Your queries might be writing to disk for at least part of the query execution. For more information, see Managing how queries use memory (p. 190).

**Query must wait for other queries to finish**

You might be able to improve overall system performance by creating query queues and assigning different types of queries to the appropriate queues. For more information, see Implementing workload management (p. 196).

**Queries are not optimized**
Analyze the explain plan to find opportunities for rewriting queries or optimizing the database. For more information, see Analyzing the Explain Plan (p. 184).

Query needs more memory to run

If a specific query needs more memory, you can increase the available memory by increasing the `wlm_query_slot_count` (p. 697).

Database requires a VACUUM command to be run

Run the VACUUM command whenever you add, delete, or modify a large number of rows, unless you load your data in sort key order. The VACUUM command reorganizes your data to maintain the sort order and restore performance. For more information, see Vacuuming tables (p. 161).

Load Fails

Your data load can fail for the following reasons; we suggest the following troubleshooting approaches.

Data source is in a different region

By default, the Amazon S3 bucket or Amazon DynamoDB table specified in the COPY command must be in the same region as the cluster. If your data and your cluster are in different regions, you will receive an error similar to the following:

`The bucket you are attempting to access must be addressed using the specified endpoint.`

If at all possible, make sure your cluster and your data source are the same region. You can specify a different region by using the `REGION` (p. 285) option with the COPY command.

Note

If your cluster and your data source are in different AWS regions, you will incur data transfer costs. You will also have higher latency and more issues with eventual consistency.

COPY command fails

Query `STL_LOAD_ERRORS` to discover the errors that occurred during specific loads. For more information, see `STL_LOAD_ERRORS` (p. 608).

Load Takes Too Long

Your load operation can take too long for the following reasons; we suggest the following troubleshooting approaches.

COPY loads data from a single file

Split your load data into multiple files. When you load all the data from a single large file, Amazon Redshift is forced to perform a serialized load, which is much slower. For more information, see Split your load data into multiple files (p. 30).

Load operation uses multiple COPY commands

If you use multiple concurrent COPY commands to load one table from multiple files, Amazon Redshift is forced to perform a serialized load, which is much slower. In this case, use a single COPY command.
Load Data Is Incorrect

Your COPY operation can load incorrect data in the following ways; we suggest the following troubleshooting approaches.

Not all files are loaded

Eventual consistency can cause a discrepancy in some cases between the files listed using an Amazon S3 ListBuckets action and the files available to the COPY command. For more information, see Verifying that the data was loaded correctly (p. 142).

Wrong files are loaded

Using an object prefix to specify data files can cause unwanted files to be read. Instead, use a manifest file to specify exactly which files to load. For more information, see the MANIFEST (p. 278) option for the COPY command and Example: COPY from Amazon S3 using a manifest (p. 299) in the COPY examples.

Analyzing the Explain Plan

Topics

• Simple EXPLAIN example (p. 186)
• EXPLAIN operators (p. 187)
• Join examples (p. 188)
• Mapping the query plan to system views (p. 190)

If you have a query that takes an unusually long time to process, you might be able to discover opportunities to improve its performance by examining the explain plan. This section describes how to view the explain plan and how to use the explain plan to find opportunities to optimize Amazon Redshift queries.

To create an explain plan, run the EXPLAIN (p. 340) command followed by the actual query text. For example:

```
explain select avg(datediff(day, listtime, saletime)) as avgwait
from sales, listing where sales.listid = listing.listid;
```

```
QUERY PLAN
XN Aggregate  (cost=6350.30..6350.31 rows=1 width=16)
  ->  XN Hash Join DS_DIST_NONE  (cost=47.08..6340.89 rows=3766 width=16)
      Hash Cond: ("outer".listid = "inner".listid)
      ->  XN Seq Scan on listing  (cost=0.00..1924.97 rows=192497 width=12)
      ->  XN Hash  (cost=37.66..37.66 rows=3766 width=12)
          ->  XN Seq Scan on sales  (cost=0.00..37.66 rows=3766 width=12)
```

The explain plan gives you the following information:

• What steps the execution engine will perform, reading from bottom to top.
• What type of operation each step performs. In this example, the operations are, reading from the bottom, Seq Scan, Hash, Seq Scan, Hash Join, Aggregate. The operations are explained later in this section.
• Which tables and columns are used in each step.
• How much data is processed in each step (number of rows and data width, in bytes)
• The relative cost of the operation.
The first thing to consider is the **cost** of each step. The cost is a measure that compares the relative execution times of the steps within a plan. It does not provide any precise information about actual execution times or memory consumption, nor does it provide a meaningful comparison between execution plans, but it does give you an indication of which steps in a query are consuming the most resources. Identifying the steps with the highest cost gives you a starting point to begin looking for opportunities to reduce the cost.

The **EXPLAIN** command does not actually run the query; the output contains only the plan that Amazon Redshift will execute if the query is run under current operating conditions. If you change the schema of a table in some way or if you change the data in the table and run **ANALYZE** (p. 268) again to update the statistical metadata, the explain plan might be different.

**Note**

You can only use **EXPLAIN** with data manipulation language (DML). If you use **EXPLAIN** for other SQL commands, such as data definition language (DDL) or database operations, the EXPLAIN operation will fail.

You can use **EXPLAIN** only for the following commands:

- SELECT
- SELECT INTO
- CREATE TABLE AS (CTAS)
- INSERT
- UPDATE
- DELETE

The **EXPLAIN** output gives limited information about data distribution and other aspects of parallel query execution. Use the system tables and views, especially some of the STL tables and the **SVL_QUERY_SUMMARY** (p. 681) view, to do the following:

- Return actual execution statistics
- Isolate the behavior of steps within the query plan.
- Monitor query activity
- Detect data distribution skew

For more information about using the system views with the explain plan, see **Mapping the query plan to system views** (p. 190).

**EXPLAIN** will fail if you use it for other SQL commands, such as data definition language (DDL) or database operations.

You can examine **EXPLAIN** output for queries in either of two ways:

- Use the **EXPLAIN** command explicitly for a single query:

```sql
explain select avg(datediff(day, listtime, saletime)) as avgwait
from sales, listing where sales.listid = listing.listid;
```

```
QUERY PLAN
XN Aggregate  (cost=6350.30..6350.31 rows=1 width=16)
  ->  XN Hash Join DS_DIST_NONE  (cost=47.08..6340.89 rows=3766 width=16)
       Hash Cond: ("outer".listid = "inner".listid)
         ->  XN Seq Scan on listing  (cost=0.00..1924.97 rows=192497 width=12)
         ->  XN Hash  (cost=37.66..37.66 rows=3766 width=12)
             ->  XN Seq Scan on sales  (cost=0.00..37.66 rows=3766 width=12)
```

**API Version 2012-12-01**

185
• Query the **STL_EXPLAIN (p. 597)** table.

Suppose you run the previous SELECT query and its query ID is 10. You can query the STL_EXPLAIN table to see the same kind of information that the EXPLAIN command returns:

```sql
select query, nodeid, parentid, substring(plannode from 1 for 30),
substring(info from 1 for 20) from stl_explain
where query=10 order by 1,2;
```

<table>
<thead>
<tr>
<th>query</th>
<th>nodeid</th>
<th>parentid</th>
<th>substring</th>
<th>substring</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1</td>
<td>0</td>
<td>XN Aggregate (cost=6350.30...)</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>2</td>
<td>1</td>
<td>-&gt; XN Merge Join DS_DIST_NO Merge Cond: (&quot;outer&quot;</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>3</td>
<td>2</td>
<td>-&gt; XN Seq Scan on lis</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>4</td>
<td>2</td>
<td>-&gt; XN Seq Scan on sal</td>
<td></td>
</tr>
</tbody>
</table>

(4 rows)

**Simple EXPLAIN example**

The following example shows the EXPLAIN output for a simple GROUP BY query against the EVENT table:

```sql
explain select eventname, count(*) from event group by eventname;
```

**QUERY PLAN**

```
XN HashAggregate (cost=131.97..133.41 rows=576 width=17)
  -> XN Seq Scan on event (cost=0.00..87.98 rows=8798 width=17)
(2 rows)
```

The order of execution is bottom to top. The initial work involves scanning the EVENT table. After the scan, the HashAggregate operator applies the GROUP BY request. The plan shown in the EXPLAIN output is a simplified, high-level view of query execution; it does not illustrate the details of parallel query processing. For example, Amazon Redshift runs parallel HashAggregate operations on each data slice on the compute nodes, and then runs the HashAggregate operation a second time on the intermediate results. To see this finer level of detail, you need to run the query itself, and then query the SVL_QUERY_SUMMARY view.

The following table describes the elements of the explain plan in the previous example:

<table>
<thead>
<tr>
<th>cost</th>
<th>Cost is a relative value that is useful for comparing operations within a plan. The costs in the query plan are cumulative as you read up the plan, so the HashAggregate cost number in this example (133.41) consists mostly of the Seq Scan cost below it (87.98). The cost entries answer two questions:</th>
</tr>
</thead>
<tbody>
<tr>
<td>rows</td>
<td>Expected number of rows to return. In this example, the scan is expected to return 8798 rows. The HashAggregate operator is expected to return 576 rows (when the duplicate event names are discarded).</td>
</tr>
</tbody>
</table>
Note
The rows estimate is based on the available statistics generated by the ANALYZE (p. 268) command. If ANALYZE has not been run recently, the estimate will be less reliable.

width
Estimated width of the average row, in bytes. In this example, the average row is estimated to be 17 bytes wide.

EXPLAIN operators
The following list briefly describes the operators that you see most often in the EXPLAIN output. For a complete list of operators, see EXPLAIN (p. 340) in the SQL command reference.

Scan operator
The Sequential Scan operator (Seq Scan) is the relation scan, or table scan, operator. Seq Scan scans the whole table sequentially from beginning to end and evaluates query constraints (in the WHERE clause) for every row. Seq Scan scans the table by columns.

Join operators
Amazon Redshift queries use different join operators depending on the physical design of the tables being joined, the location of the data required for the join, and the specific requirements of the query itself. The following operators are available:
• Nested Loop — The least optimal join, Nested Loop is used mainly for cross-joins (Cartesian products) and some inequality joins.
• Hash Join and Hash — Typically faster than a nested loop join, Hash Join and Hash are used for inner joins and left and right outer joins. This operator will generally be used when you are joining tables if your joining columns for both tables are not both distribution keys and sort keys. The Hash operator creates the hash table for the inner table in the join; the Hash Join operator reads the outer table, hashes the joining column, and finds matches in the inner hash table.
• Merge Join — Also typically faster than a nested loop join, this operator is used for inner and outer joins. It will generally be used when the joining columns of both tables are both distribution keys and sort keys. The operator reads two sorted tables in order and finds the matching rows.

Aggregate query operators
The following operators are used in queries that involve aggregate functions and GROUP BY operations.
• Aggregate — Used for scalar aggregate functions.
• HashAggregate — Used for unsorted grouped aggregate functions.
• GroupAggregate — Used for sorted grouped aggregate functions.

Sort operators
The following operators are used when queries have to sort or merge result sets.
• Sort — Evaluates the ORDER BY clause and other sort operations, such as sorts required by UNION queries and joins, SELECT DISTINCT queries, and window functions.
• Merge — Produces final sorted results according to intermediate sorted results that derive from parallel operations.

UNION, INTERSECT, and EXCEPT operators
The following operators are used for queries that involve set operations with UNION, INTERSECT, and EXCEPT.
• Subquery — Scan and Append Used to run UNION queries.
• Hash Intersect Distinct and Hash Intersect All — Used for INTERSECT and INTERSECT ALL queries.
• SetOp Except — Used to run EXCEPT (or MINUS) queries.
Other operators
The following operators are difficult to categorize but they appear frequently in EXPLAIN output for routine queries.
- **Unique** — Eliminates duplicates for SELECT DISTINCT queries and UNION queries.
- **Limit** — Evaluates the LIMIT clause.
- **Window** — Runs window functions.
- **Result** — Runs scalar functions that do not involve any table access.
- **Subplan** — Used for certain subqueries.
- **Network** — Sends intermediate results to the leader node for further processing.
- **Materialize** — Saves rows for input to nested loop joins and some merge joins.

Join examples
The EXPLAIN output exposes references to inner and outer table joins. The inner table is scanned first. This is the table that is probed for matches. It is usually held in memory, is usually the source table for hashing, and if possible, is the smaller table of the two being joined. The outer table is the source of rows to match against the inner table. It is usually read from disk on the fly. The order of tables in the FROM clause of a query does not determine which table is inner and which is outer.

The EXPLAIN output for joins also specifies a method for redistributing data (how data will be moved around the cluster to facilitate the join). This data movement can be either a broadcast or a redistribution. In a broadcast, the data values from one side of a join are copied from each compute node to every other compute node, so that every compute node ends up with a complete copy of the data. In a redistribution, participating data values are sent from their current slice to a new slice (possibly on a different node). Data is typically redistributed to match the distribution key of the other table participating in the join if that distribution key is one of the joining columns. If neither of the tables has distribution keys on one of the joining columns, either both tables are distributed or the inner table is broadcast to every node.

You will see the following attributes in the EXPLAIN output for joins:

- **DS_BCAST_INNER**
  Broadcast a copy of the entire inner table to all compute nodes.

- **DS_DIST_ALL_NONE**
  No redistribution is required because the inner table was distributed to every node using DISTSTYLE ALL.

- **DS_DIST_NONE**
  No tables are redistributed: collocated joins are possible because corresponding slices are joined without moving data between nodes.

- **DS_DIST_INNER**
  The inner table is redistributed.

- **DS_DIST_ALL_INNER**
  The entire inner table is redistributed to a single slice because the outer table uses DISTSTYLE ALL.

- **DS_DIST_BOTH**
  Both tables are redistributed.

Join examples
These examples show the different join algorithms chosen by the query planner. In these particular cases, the choices in the query plan depend on the physical design of the tables.
Hash join two tables

The following query joins EVENT and CATEGORY on CATID. The CATID column is the distribution and sort key for CATEGORY but not for EVENT. A hash join is performed with EVENT as the outer table and CATEGORY as the inner table. Because CATEGORY is the smaller table, the planner broadcasts a copy of it to the compute nodes during query processing (DS_BCAST_INNER). The join cost in this example accounts for most of the cumulative cost of the plan.

*explain select * from category, event where category.catid=event.catid;*

**QUERY PLAN**

```
XN Hash Join DS_BCAST_INNER  (cost=0.14..6600286.07 rows=8798 width=84)
  Hash Cond: ("outer".catid = "inner".catid)
  
  ->  XN Seq Scan on event  (cost=0.00..87.98 rows=8798 width=35)
  
  ->  XN Hash  (cost=0.11..0.11 rows=11 width=49)
  
  ->  XN Seq Scan on category  (cost=0.00..0.11 rows=11 width=49)

(5 rows)
```

**Note**

Aligned indents for operators in the EXPLAIN output sometimes indicate that those operations do not depend on each other and can start in parallel. In this case, although the scan on the EVENT table and the Hash operation are aligned, the EVENT scan must wait until the Hash operation has fully completed.

Merge join two tables

The following query has the same structure as the previous example, but it joins SALES and LISTING on LISTID. This column is the distribution and sort key for both tables. A merge join is chosen, and no redistribution of data is required for the join (DS_DIST_NONE).

*explain select * from sales, listing where sales.listid = listing.listid;*

**QUERY PLAN**

```
XN Merge Join DS_DIST_NONE  (cost=0.00..127.65 rows=3766 width=97)
  Merge Cond: ("outer".listid = "inner".listid)
  
  ->  XN Seq Scan on sales  (cost=0.00..37.66 rows=3766 width=53)
  
  ->  XN Seq Scan on listing  (cost=0.00..1924.97 rows=192497 width=44)

(4 rows)
```

This example demonstrates the different types of joins within the same query. As in the previous example, SALES and LISTING are merge-joined, but the third table, EVENT, must be hash-joined with the results of the merge join. Again, the hash join incurs a broadcast cost.

*explain select * from sales, listing, event
where sales.listid = listing.listid and sales.eventid = event.eventid;*

**QUERY PLAN**

```
XN Hash Join DS_DIST_OUTER  (cost=2.50..414400186.40 rows=740 width=440)
  Outer Dist Key: "inner".eventid
  Hash Cond: ("outer".eventid = "inner".eventid)
  
  ->  XN Merge Join DS_DISTNONE  (cost=0.00..26.65 rows=740 width=104)
  
  ->  XN Seq Scan on listing  (cost=0.00..8.00 rows=800 width=48)
  
  ->  XN Seq Scan on sales  (cost=0.00..7.40 rows=740 width=56)

(5 rows)
```
Join, aggregate, and sort example

The following query executes a hash join of the SALES and EVENT tables, followed by aggregation and sort operations to account for the grouped SUM function and the ORDER BY clause. The initial Sort operator runs in parallel on the compute nodes, and then the Network operator sends the results to the leader node where the Merge operator produces the final sorted results.

```
explain select eventname, sum(pricepaid) from sales, event
where sales.eventid=event.eventid group by eventname
order by 2 desc;
```

Mapping the query plan to system views

The explain plan alone does not have all of the details that you need. The actual execution steps and statistics for each query are logged in the system views `SVL_QUERY_SUMMARY` and `SVL_QUERY_REPORT`. These views capture query activity at a finer level of granularity than the EXPLAIN output, and they contain metrics that you can use to monitor query activity. To study the full execution profile of a query, first run the query, then run the EXPLAIN command for the query, and then map the EXPLAIN output to the system views. For example, you can use the system views to detect distribution skew, which might indicate that you need to reevaluate your choices for distribution keys.

Managing how queries use memory

Topics

- Determining whether a query is writing to disk (p. 191)
- Determining which steps are writing to disk (p. 192)

Amazon Redshift supports the ability for queries to run entirely in memory, or if necessary, to swap intermediate query results to disk.
If your Amazon Redshift query performance results are slower than expected, your queries might be writing to disk for at least part of the query execution. A query that runs entirely in memory will be faster than a query that is frequently transferring data to and from disk.

Amazon Redshift can write intermediate results to disk for DELETE statements and for sorts, hashes, and aggregates in SELECT statements. Writing intermediate results to disk ensures that a query continues to run even if it reaches the limits of system memory, but the additional disk I/O can degrade performance.

The first step is to determine whether a query writes intermediate results to disk instead of keeping them in memory. Then you can use the explain plan and the system tables to identify which steps are writing to disk.

### Determining whether a query is writing to disk

To determine whether any query steps wrote intermediate results to disk for a particular query, use the following set of system table queries. Use this method for all types of queries that can write to disk, including both SELECT and DELETE statements.

1. Issue the following query to determine the query ID for the query being investigated:

   ```sql
   select query, elapsed, substring
   from svl_qlog
   order by query desc limit 5;
   ```

   This query displays the query ID, execution time, and truncated query text for the last five queries to run against the database tables, as shown in the following sample output:

<table>
<thead>
<tr>
<th>query</th>
<th>elapsed</th>
<th>substring</th>
</tr>
</thead>
<tbody>
<tr>
<td>1026</td>
<td>9574270</td>
<td>select s.seg, s.maxtime, s.label, s.is_diskbased from query_</td>
</tr>
<tr>
<td>1025</td>
<td>18672594</td>
<td>select t1.c1 x1, t2.c2 x2 from tbig t1, tbig t2 where t1.c1</td>
</tr>
<tr>
<td>1024</td>
<td>84266</td>
<td>select count(<em>) as underrepped from ( select count(</em>) as a from tbig t1, tbig t2 where t1.c1 )</td>
</tr>
<tr>
<td>1023</td>
<td>83217</td>
<td>select system_status from stv_gui_status</td>
</tr>
<tr>
<td>1022</td>
<td>39236</td>
<td>select * from stv_sessions</td>
</tr>
</tbody>
</table>

   (5 rows)

2. Examine the output to determine the query ID that matches the query that you are investigating.

3. Using the query ID, issue the following query to determine whether any steps for this query wrote to the disk. The following example uses query ID 1025:

   ```sql
   select query, step, rows, workmem, label, is_diskbased
   from svl_query_summary
   where query = 1025 order by workmem desc;
   ```

   This query returns the following sample output:

<table>
<thead>
<tr>
<th>query</th>
<th>step</th>
<th>rows</th>
<th>workmem</th>
<th>label</th>
<th>is_diskbased</th>
</tr>
</thead>
<tbody>
<tr>
<td>1025</td>
<td>0</td>
<td>16000000</td>
<td>43205240</td>
<td>scan tbl=9</td>
<td>f</td>
</tr>
<tr>
<td>1025</td>
<td>2</td>
<td>16000000</td>
<td>43205240</td>
<td>hash tbl=142</td>
<td>t</td>
</tr>
</tbody>
</table>
4. Look through the output. If IS_DISKBASED is true ("t") for any step, then that step wrote data to disk. In the previous example, the hash step intermediate results were written to disk.

If you find that steps are writing to disk and affecting performance, the easiest solution is to increase the memory available to a query by increasing the slot count for the query. Workload management (WLM) reserves slots in a query queue according to the concurrency level set for the queue (for example, if concurrency level is set to 5, then the query has five slots). WLM allocates the available memory for a query equally to each slot. Slot count is set in the wlm_query_slot_count (p. 697) parameter. Increasing the slot count increases the amount of memory available for the query.

**Determining which steps are writing to disk**

To see which step in the query is writing to disk, you will use the EXPLAIN command, along with several system table queries.

This section walks through the process of using EXPLAIN and the SVL_QLOG and SVL_QUERY_SUMMARY system tables to examine a hash join query that writes to disk.

We will examine the following query:

```sql
select t1.c1 x1, t2.c2 x2
from tbig t1, tbig t2
where t1.c1 = t2.c2;
```

1. To view the plan for the query, run the following EXPLAIN command:

```sql
explain select t1.c1 x1, t2.c2 x2 from tbig t1, tbig t2 where t1.c1 = t2.c2;
```

This command displays the following sample output:

```
| 0 | 16000000 | 55248 | scan tbl=116536 | f |
| 2 | 16000000 | 55248 | dist | f |
```

4 rows)

This plan shows that the query will be executed starting with the innermost nodes and continuing to the outer nodes. The query will execute with a hash. With a large data set, hashes, aggregates, and sorts are the relational operators that would be likely to write data to disk if the system does not have enough memory allocated for query processing.

2. To actually run the query, issue the following query:
select `t1.c1` `x1`, `t2.c2` `x2`
from `tbig` `t1`, `tbig` `t2`
where `t1.c1` = `t2.c2` ;

3. To use the SVL_QLOG view to obtain the query ID for the query that just executed, issue the following query:

```sql
select query, elapsed, substring
from svl_qlog order by query desc limit 5;
```

This command displays the last five queries to execute on Amazon Redshift.

<table>
<thead>
<tr>
<th>query</th>
<th>elapsed</th>
<th>substring</th>
</tr>
</thead>
<tbody>
<tr>
<td>1033</td>
<td>4592158</td>
<td><code>select </code>t1.c1<code> </code>x1<code>, </code>t2.c2<code> </code>x2<code>from</code>tbig<code> </code>t1<code>, </code>tbig<code> </code>t2`</td>
</tr>
<tr>
<td>1032</td>
<td>477285</td>
<td><code>select query, step, rows, memory, label,</code></td>
</tr>
<tr>
<td>1031</td>
<td>23742</td>
<td><code>select query, elapsed, substring from svl_qlog</code></td>
</tr>
<tr>
<td>1030</td>
<td>900222</td>
<td><code>select * from svl_query_summary limit 5;</code></td>
</tr>
<tr>
<td>1029</td>
<td>1304248</td>
<td><code>select query, step, rows, memory, label,</code></td>
</tr>
</tbody>
</table>

(5 rows)

You can see that the query that selects from TBIG is query 1033.

4. Using the query ID from the previous step, view the SVL_QUERY_SUMMARY information for the query to see which relational operators from the plan (if any) wrote to disk. This query groups the steps sequentially and then by rows to see how many rows were processed:

```sql
select query, step, rows, workmem, label, is_diskbased
from svl_query_summary
where query = 1033
order by workmem desc;
```

This query returns the following sample output:

<table>
<thead>
<tr>
<th>query</th>
<th>step</th>
<th>rows</th>
<th>workmem</th>
<th>label</th>
<th>is_diskbased</th>
</tr>
</thead>
<tbody>
<tr>
<td>1033</td>
<td>0</td>
<td>16000000</td>
<td><code>141557760 scan tbl=9</code></td>
<td><code>f</code></td>
<td></td>
</tr>
<tr>
<td>1033</td>
<td>2</td>
<td>16000000</td>
<td><code>135266304 hash tbl=142</code></td>
<td><code>t</code></td>
<td></td>
</tr>
<tr>
<td>1033</td>
<td>0</td>
<td>16000000</td>
<td><code>128974848 scan tbl=116536</code></td>
<td><code>f</code></td>
<td></td>
</tr>
<tr>
<td>1033</td>
<td>2</td>
<td>16000000</td>
<td><code>122683392 dist</code></td>
<td><code>f</code></td>
<td></td>
</tr>
</tbody>
</table>

(4 rows)

5. Compare the svl_query_summary output to the plan generated by the EXPLAIN statement.

You can view the results to see where the query steps correspond to the steps in the plan. If any of these steps go to disk, use this information to determine how many rows are being processed and how much memory is being used so that you can either alter your query or adjust your workload management (WLM) configuration. For more information, see Implementing workload management (p. 196).
Monitoring disk space

You can query the STV_PARTITIONS, STV_TBL_PERM, and STV_BLOCKLIST system tables to obtain information about disk space.

**Note**
To access these system tables, you must be logged in as a superuser.

You can monitor your disk space using the STV_PARTITIONS table.

The following query returns the raw disk space used and capacity, in 1 MB disk blocks. The raw disk space includes space that is reserved by Amazon Redshift for internal use, so it is larger than the nominal disk capacity, which is the amount of disk space available to the user. The **Percentage of Disk Space Used** metric on the **Performance** tab of the Amazon Redshift Management Console reports the percentage of nominal disk capacity used by your cluster. We recommend that you monitor the **Percentage of Disk Space Used** metric to maintain your usage within your cluster's nominal disk capacity.

**Important**
We strongly recommend that you do not exceed your cluster's nominal disk capacity. While it might be technically possible under certain circumstances, exceeding your nominal disk capacity decreases your cluster's fault tolerance and increases your risk of losing data.

```sql
select owner as node, diskno, used, capacity
from stv_partitions
order by 1, 2, 3, 4;
```

This query returns the following result for a single node cluster:

<table>
<thead>
<tr>
<th>node</th>
<th>diskno</th>
<th>used</th>
<th>capacity</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>245</td>
<td>1906185</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>200</td>
<td>1906185</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>200</td>
<td>1906185</td>
</tr>
</tbody>
</table>

(3 rows)

For more information, see **STV_PARTITIONS** (p. 657). If you start running out of disk space, you can increase the number of compute nodes or change to a higher capacity node type. See **Modifying a cluster**.

You can use STV_BLOCKLIST and STV_TBL_PERM to determine the amount of storage allocated to a database table.

The STV_BLOCKLIST system table contains information about the number of blocks allocated to each table in the data warehouse cluster, and the STV_TBL_PERM table contains the table IDs for all of the permanent tables in the database.

Issue the following query to find out how many blocks are allocated to the SALES table:

```sql
select tbl, count(*)
from stv_blocklist
where tbl in (select id
from stv_tbl_perm
where name='sales')
group by tbl;
```

The query returns the following result:
Each data block occupies 1 MB, so 100 blocks represents 100 MB of storage.

For more information, see STV_BLOCKLIST (p. 650) and STV_TBL_PERM (p. 662).

## Benchmarking with compiled code

Amazon Redshift generates code for each execution plan, compiles the code, and then sends compiled code segments to the compute nodes. The compiled code segments are stored in a least recently used (LRU) cache and shared across sessions in a cluster, so subsequent executions of the same query, even in different sessions and often even with different query parameters, will run faster because they can skip the initial generation and compilation steps. The LRU cache persists through cluster reboots, but is wiped by maintenance upgrades.

The compiled code executes much faster because it eliminates the overhead of using an interpreter; however, there is always some overhead cost the first time the code is generated and compiled, even for the cheapest query plans. As a result, the performance of a query the first time you run it can be misleading. The overhead cost might be especially noticeable when you run ad hoc queries. You should always run the query a second time to evaluate its performance.

When you compare the execution times for queries, you should not use the results for the first time you execute the query. Instead, compare the times for the second execution of each query. Similarly, be careful about comparing the performance of the same query sent from different clients. The execution engine generates different code for the JDBC connection protocols and for the ODBC and psql (libpq) connection protocols. If two clients use different protocols, each client will incur the first-time cost of generating compiled code, even for the same query. For example, SQLWorkbench uses the JDBC connection protocol, but the PostgreSQL query utility, psql, connects by using a set of library functions called libpq, so the execution engine generates two distinct versions of compiled code. Other clients that use the same protocol, however, will benefit from sharing the cached code. A client that uses ODBC and a client running psql with libpq can share the same compiled code.

To see the effect of generating compiled code for a query, you can query the SVL_COMPILE (p. 672) system table.

In the following example, queries 35878 and 35879 executed the same SQL statement sequentially. Note the difference in the duration of the compile time for the two queries.

```sql
SELECT userid, xid, pid, query, segment, locus,
       datediff(ms, starttime, endtime) as duration, compile
FROM svl_compile
WHERE query = 35878 OR query = 35879
ORDER BY query, segment;
```
Setting the JDBC fetch size parameter

**Note**
Fetch size is not supported for ODBC.

By default, the JDBC driver collects all the results for a query at one time. As a result, when you attempt to retrieve large result sets over a JDBC connection, you might encounter client-side out-of-memory errors. To enable your client to retrieve result sets in batches instead of in a single all-or-nothing fetch, set the JDBC fetch size parameter in your client application.

**Note**
If you need to extract large data sets, we recommend using an UNLOAD statement to transfer the data to Amazon S3. When you use UNLOAD, the compute nodes work in parallel to transfer the data directly. When you retrieve data using JDBC, the data is funneled to the client through the leader node.

For the best performance, set the fetch size to the highest value that does not lead to out of memory errors. A lower fetch size value results in more server trips, which prolongs execution times. The server reserves resources, including the WLM query slot and associated memory, until the client retrieves the entire result set or the query is canceled. When you tune the fetch size appropriately, those resources are released more quickly, making them available to other queries.

For more information about setting the JDBC fetch size parameter, see [Getting results based on a cursor](#) in the PostgreSQL documentation.

### Implementing workload management

**Topics**
- Defining query queues (p. 197)
- WLM queue assignment rules (p. 199)
- Modifying the WLM configuration (p. 201)
- Assigning queries to queues (p. 202)
- Monitoring workload management (p. 203)

You can use workload management (WLM) to define multiple query queues and to route queries to the appropriate queues at runtime.
Long-running queries can become performance bottlenecks. For example, suppose your data warehouse has two groups of users. One group submits occasional long-running queries that select and sort rows from several large tables. The second group frequently submits short queries that select only a few rows from one or two tables and run in a few seconds. In this situation, the short-running queries might have to wait in a queue for a long-running query to complete.

You can improve system performance and your users experience by modifying your WLM configuration to create separate queues for the long-running queries and the short-running queries. At runtime, you can route queries to the queues according to user groups or query groups.

**Defining query queues**

By default, a cluster is configured with one queue that can run five queries concurrently. In addition, Amazon Redshift reserves one dedicated Superuser queue on the cluster, which has a concurrency level of one. The Superuser queue is not configurable.

*Note*

The primary purpose of the Superuser queue is to aid in troubleshooting. You should not use it to perform routine queries.

You can modify the WLM configuration for a cluster to define up to eight query queues in addition to the Superuser queue.

Amazon Redshift allocates an equal share of server memory, by default, to each queue defined in the WLM configuration. To configure the proportion of memory used by query queues, you can set the WLM Percent to Use parameter. For more information, see WLM Memory Percent to Use (p. 198).

At runtime, Amazon Redshift assigns queries to queues based on user groups and query groups. For information about how to assign queries to user groups and query groups at runtime, see Assigning queries to queues (p. 202).

For each queue, you specify

- Concurrency level
- User groups
- Query groups
- Wildcards
- WLM memory percent to use
- WLM timeout

**Concurency level**

Queries in a queue run concurrently until they reach the concurrency level defined for that queue. Subsequent queries then wait in the queue. Each queue can be configured to run up to 50 queries concurrently. The maximum total concurrency level for all user-defined queues, not including the reserved Superuser queue, is 50. Amazon Redshift allocates an equal, fixed share of server memory to each queue, and, by default, an equal, fixed share of a queue’s memory to each query slot in the queue.

By default, WLM queues have a concurrency level of 5. Your workload might benefit from a higher concurrency level in certain cases, such as the following:

- If many small queries are forced to wait for long-running queries, create a separate queue with a higher concurrency level and assign the smaller queries to that queue. A queue with a higher concurrency level has less memory allocated to each query slot, but the smaller queries require less memory.
- If you have multiple queries that each access data on a single slice, set up a separate WLM queue to execute those queries concurrently. Amazon Redshift will assign concurrent queries to separate slices,
which allows multiple queries to execute in parallel on multiple slices. For example, if a query is a simple aggregate with a predicate on the distribution key, the data for the query will be located on a single slice.

As a best practice, we recommend using a concurrency level of 15 or lower. All of the compute nodes in a cluster, and all of the slices on the nodes, participate in parallel query execution. By increasing concurrency, you increase the contention for system resources and limit the overall throughput.

The memory that is allocated to each queue is divided among the query slots in that queue. The amount of memory available to a query is the memory allocated to the query slot in which the query is running, regardless of the number of queries that are actually running concurrently. A query that can run entirely in memory when the concurrency level is 5 might need to write intermediate results to disk if the concurrency level is increased to 20. The additional disk I/O could degrade performance.

If a specific query needs more memory than is allocated to a single query slot, you can increase the available memory by increasing the wlm_query_slot_count parameter. The following example sets wlm_query_slot_count to 10, performs a vacuum, and then resets wlm_query_slot_count to 1.

```
set wlm_query_slot_count to 10;
vacuum;
set wlm_query_slot_count to 1;
```

For more information, see Managing how queries use memory (p. 190).

**User groups**

You can assign a comma-separated list of user groups to a queue. When a member of a listed user group runs a query, that query runs in the corresponding queue. There is no set limit on the number of user groups that can be assigned to a queue.

**Query groups**

You can assign a comma-separated list of query groups for each queue. A query group is simply a label. At runtime, you can assign the query group label to a series of queries. Any queries that are assigned to a listed query group will run in the corresponding queue. There is no set limit to the number of query groups that can be assigned to a queue.

**Wildcards**

You can assign user groups and query groups to a queue either individually or by using wildcards. For example, if you add `dba_*` to the list of user groups for a queue, any query that is run by a user with a user name that begins with `dba_` is assigned to that queue. Wildcards are disabled by default. To enable a queue to use wildcards through the AWS Management Console, select the Enable User Group Wildcards check box or the Enable Query Group Wildcards check box on the Workload Management Configuration tab. To enable wildcards through the API or CLI, set the query_group_wild_card value or the user_group_wild_card value to 1.

**WLM Memory Percent to Use**

To specify the amount of available memory that is allocated to a query, you can set the WLM Memory Percent to Use parameter. By default, each user-defined queue is allocated an equal portion of the memory that is available for user-defined queries. For example, if you have four user-defined queues, each queue is allocated 25% of the available memory. The superuser queue has its own allocated memory and cannot be modified. You can assign an integer percentage of memory to each queue, up to a total
of 100%. If you attempt to assign more than 100%, the WLM configuration is rejected. If you specify a WLM Memory Percent to Use value for one or more queues but not all of the queues, then the unallocated memory is divided evenly among the remaining queues.

For example, if you configure four queues and assign 30% and 40% to two queues, then 30% is left for the other two queues. The result is that queues are allocated as follows: 30%, 40%, 15%, 15%. If you allocate memory to all of the queues and the total allocation is less than 100%, then the remainder is allocated evenly to all of the queues.

To set the WLM Memory Percent to Use parameter by using the AWS Management Console, go to the Workload Management Configuration tab and specify an integer in the WLM Memory Percent to Use field for each queue. To set the WLM Memory Percent to Use value by using the API or CLI, use the memory_percent_to_use parameter. For information about modifying parameter groups, see Amazon Redshift Parameter Groups.

**WLM timeout**

To limit the amount of time that queries in a given WLM queue are permitted to use, you can set the WLM timeout value for each queue. The timeout parameter specifies the amount of time, in milliseconds, that Amazon Redshift will wait for a query in a queue to execute before canceling the query.

The function of WLM timeout is similar to the statement_timeout (p. 697) configuration parameter, except that, where the statement_timeout configuration parameter applies to the entire cluster, WLM timeout is specific to a single queue in the WLM configuration.

To set the WLM timeout parameter by using the AWS Management Console, go to the Workload Management Configuration tab and specify a number of milliseconds in the Timeout field for each queue. Specify 0 or leave the field blank to disable WLM timeout. WLM timeout is disabled by default.

To set the WLM timeout value by using the API or CLI, use the max_execution_time parameter. To set the statement_timeout configuration parameter for a cluster, modify the Parameter Group configuration for the cluster. For information about modifying parameter groups, see Amazon Redshift Parameter Groups.

If both WLM timeout (max_execution_time) and statement_timeout are specified, the shorter timeout is used.

**Default queue**

The last queue defined in the WLM configuration is the default queue. You can set the concurrency level and the timeout for the default queue, but it cannot include user groups or query groups. The default queue counts against the limit of eight query queues and the limit of 50 concurrent queries.

**Superuser queue**

To run a query in the Superuser queue, a user must be logged in as a superuser and must run the query within the predefined 'superuser' query group.

To view a list of superusers, query the PG_USER system catalog table.

```sql
select * from pg_user where usesuper = 'true';
```

**WLM queue assignment rules**

When a user runs a query, WLM assigns the query to the first matching queue, based on these rules.
1. If a user logged in as a superuser and runs a query in the query group labeled ‘superuser’, the query is assigned to the Superuser queue.
2. If a user belongs to a listed user group, the query is assigned to the first corresponding queue.
3. If a user runs a query within a listed query group, the query is assigned to the first corresponding queue.
4. If a query does not meet any criteria, the query is assigned to the default queue, which is the last queue defined in the WLM configuration.

The following table shows a WLM configuration with the Superuser queue and four user-defined queues.

<table>
<thead>
<tr>
<th>Queue</th>
<th>Concurrency</th>
<th>User Groups</th>
<th>Query Groups</th>
</tr>
</thead>
<tbody>
<tr>
<td>Superuser</td>
<td>1</td>
<td></td>
<td>superuser</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>user_group_1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td></td>
<td>query_group_A</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>user_group_2</td>
<td>query_group_B</td>
</tr>
<tr>
<td>Default</td>
<td>4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Queue assignments example**

The following example shows how queries are assigned to the queues in the previous example according to user groups and query groups. For information about how to assign queries to user groups and query groups at runtime, see Assigning queries to queues (p. 202) later in this section.
In this example, the first set of statements shows three ways to assign users to user groups. The user `analyst1` belongs to the user group `user_group_2`. The query run by `analyst1` is a member of `query_group_A`, but the query is assigned to Queue 3 because the check for user group occurs before the check for query group.

### Modifying the WLM configuration

Query queues are defined in the WLM configuration. The WLM configuration is an editable parameter (wlm_json_configuration) in a parameter group, which can be associated with one or more clusters.

The easiest way to create a WLM configuration is by using the Amazon Redshift management console to define a set of queues. You can also use the Amazon Redshift command line interface (CLI) or the Amazon Redshift API.

For information about modifying WLM configurations, see Amazon Redshift Parameter Groups.

**Important**

You must reboot the cluster after changing the WLM configuration.
Assigning queries to queues

The following examples assign queries to queues according to user groups and query groups.

Assigning queries to queues based on user groups

If a user group name is listed in a queue definition, queries run by members of that user group will be assigned to the corresponding queue. The following example creates user groups and adds users to groups by using the SQL commands CREATE USER (p. 325), CREATE GROUP (p. 308), and ALTER GROUP (p. 259).

```
create group admin_group with user admin246, admin135, sec555;
create user vp1234 in group ad_hoc_group password 'vpPass1234';
alter group admin_group add user analyst44, analyst45, analyst46;
```

Assigning a query to a query group

You can assign a query to a queue at runtime by assigning your query to the appropriate query group. Use the SET command to begin a query group.

```
SET query_group TO 'group_label'
```

Where 'group_label' is a query group label that is listed in the WLM configuration.

All queries that you run after the SET query_group command will run as members of the specified query group until you either reset the query group or end your current login session. For information about setting and resetting Amazon Redshift objects, see SET (p. 389) and RESET (p. 355) in the SQL Command Reference.

The query group labels that you specify must be included in the current WLM configuration; otherwise, the SET query_group command has no effect on query queues.

The label defined in the TO clause is captured in the query logs so that you can use the label for troubleshooting. For information about the query_group configuration parameter, see query_group (p. 695) in the Configuration Reference.

The following example runs two queries as part of the query group 'priority' and then resets the query group.

```
set query_group to 'priority';
select count(*)from stv_blocklist;
select query, elapsed, substring from svl_qlog order by query desc limit 5;
reset query_group;
```

Assigning queries to the Superuser queue

To assign a query to the Superuser queue, log in to Amazon Redshift as a superuser and then run the query in the 'superuser' group. When you are done, reset the query group so that subsequent queries do not run in the Superuser queue.

This example assigns two commands to run in the Superuser queue.
Monitoring workload management

WLM configures query queues according to internally-defined WLM service classes. Amazon Redshift creates several internal queues according to these service classes along with the queues defined in the WLM configuration. The terms queue and service class are often used interchangeably in the system tables.

You can view the status of queries, queues, and service classes by using WLM-specific system tables. Query the following system tables to do the following:

• View which queries are being tracked and what resources are allocated by the workload manager.
• See which queue a query has been assigned to.
• View the status of a query that is currently being tracked by the workload manager.

<table>
<thead>
<tr>
<th>Table name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>STL_WLM_ERROR (p. 645)</td>
<td>Contains a log of WLM-related error events.</td>
</tr>
<tr>
<td>STL_WLM_QUERY (p. 645)</td>
<td>Lists queries that are being tracked by WLM.</td>
</tr>
<tr>
<td>STV_WLM_CLASSIFICATION_CONFIG (p.665)</td>
<td>Shows the current classification rules for WLM.</td>
</tr>
<tr>
<td>STV_WLM_QUERY_QUEUE_STATE (p.666)</td>
<td>Records the current state of the query queues.</td>
</tr>
<tr>
<td>STV_WLM_QUERY_STATE (p.667)</td>
<td>Provides a snapshot of the current state of queries that are being tracked by WLM.</td>
</tr>
<tr>
<td>STV_WLM_QUERY_TASK_STATE (p.668)</td>
<td>Contains the current state of query tasks.</td>
</tr>
<tr>
<td>STV_WLM_SERVICE_CLASS_CONFIG (p.669)</td>
<td>Records the service class configurations for WLM.</td>
</tr>
<tr>
<td>STV_WLM_SERVICE_CLASS_STATE (p.670)</td>
<td>Contains the current state of the service classes.</td>
</tr>
</tbody>
</table>

You use the task ID to track a query in the system tables. The following example shows how to obtain the task ID of the most recently submitted user query:

```
select task from stl_wlm_query where exec_start_time = (select max(exec_start_time) from stl_wlm_query);
```

```
<table>
<thead>
<tr>
<th>task</th>
</tr>
</thead>
<tbody>
<tr>
<td>137</td>
</tr>
</tbody>
</table>

The following example displays queries that are currently executing or waiting in various service classes (queues). This query is useful in tracking the overall concurrent workload for Amazon Redshift:

```
select * from stv_wlm_query_state order by query;
```
The previous example shows four queries:

- The currently executing system table query (`select * from stv_wlm_query_state ... ;`).
- A query returning results in service class 3.
- A query waiting in the queue in service class 3. Service class 3 has one query task so that one query can execute at a time in that service class.
- A query currently executing in service class 2.
Amazon Redshift SQL

Topics

• SQL functions supported on the leader node (p. 205)
• Amazon Redshift and PostgreSQL (p. 206)

Amazon Redshift is built around industry-standard SQL, with added functionality to manage very large datasets and support high-performance analysis and reporting of those data.

Note
The maximum size for a single Amazon Redshift SQL statement is 16 MB.

SQL functions supported on the leader node

Some Amazon Redshift queries are distributed and executed on the compute nodes, and other queries execute exclusively on the leader node.

The leader node distributes SQL to the compute nodes whenever a query references user-created tables or system tables (tables with an STL or STV prefix and system views with an SVL or SVV prefix). A query that references only catalog tables (tables with a PG prefix, such as PG_TABLE_DEF, which reside on the leader node) or that does not reference any tables, runs exclusively on the leader node.

Some Amazon Redshift SQL functions are supported only on the leader node and are not supported on the compute nodes. A query that uses a leader-node function must execute exclusively on the leader node, not on the compute nodes, or it will return an error.
The documentation for each function that must run exclusively on the leader node includes a note stating that the function will return an error if it references user-defined tables or Amazon Redshift system tables. See Leader-node only functions (p. 414) for a list of functions that run exclusively on the leader node.

Examples

The CURRENT_SCHEMA function is a leader-node only function. In this example, the query does not reference a table, so it runs exclusively on the leader node.

```sql
select current_schema();
```

Result

```
current_schema
-----------
public
(1 row)
```

In the next example, the query references a system catalog table, so it runs exclusively on the leader node.

```sql
select * from pg_table_def
where schemaname = current_schema() limit 1;
```

<table>
<thead>
<tr>
<th>schemaname</th>
<th>tablename</th>
<th>column</th>
<th>type</th>
<th>encoding</th>
<th>distkey</th>
<th>sortkey</th>
<th>notnull</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>category</td>
<td>catid</td>
<td>smallint</td>
<td>none</td>
<td>t</td>
<td></td>
<td>t</td>
</tr>
</tbody>
</table>

(1 row)

In the next example, the query references an Amazon Redshift system table that resides on the compute nodes, so it returns an error.

```sql
select current_schema(), userid from users;
```

INFO: Function "current_schema()" not supported.
ERROR: Specified types or functions (one per INFO message) not supported on Redshift tables.

Amazon Redshift and PostgreSQL

Topics

- Amazon Redshift and PostgreSQL JDBC and ODBC (p. 207)
- Features that are implemented differently (p. 207)
- Unsupported PostgreSQL features (p. 208)
- Unsupported PostgreSQL data types (p. 209)
- Unsupported PostgreSQL functions (p. 210)

Amazon Redshift is based on PostgreSQL 8.0.2. Amazon Redshift and PostgreSQL have a number of very important differences that you must be aware of as you design and develop your data warehouse applications.
Amazon Redshift is specifically designed for online analytic processing (OLAP) and business intelligence (BI) applications, which require complex queries against large datasets. Because it addresses very different requirements, the specialized data storage schema and query execution engine that Amazon Redshift uses are completely different from the PostgreSQL implementation. For example, where online transaction processing (OLTP) applications typically store data in rows, Amazon Redshift stores data in columns, using specialized data compression encodings for optimum memory usage and disk I/O. Some PostgreSQL features that are suited to smaller-scale OLTP processing, such as secondary indexes and efficient single-row data manipulation operations, have been omitted to improve performance.

See Amazon Redshift System Overview (p. 4) for a detailed explanation of the Amazon Redshift data warehouse system architecture.

PostgreSQL 9.x includes some features that are not supported in Amazon Redshift. In addition, there are important differences between Amazon Redshift SQL and PostgreSQL 8.0.2 that you must be aware of. This section highlights the differences between Amazon Redshift and PostgreSQL 8.0.2 and provides guidance for developing a data warehouse that takes full advantage of the Amazon Redshift SQL implementation.

### Amazon Redshift and PostgreSQL JDBC and ODBC

Amazon Redshift is based on PostgreSQL version 8.0.2. Client applications and tools can use PostgreSQL JDBC or ODBC drivers to connect to Amazon Redshift clusters. We recommend using the 8.x JDBC drivers and the 9.x ODBC drivers to help ensure compatibility. The PostgreSQL 9.x JDBC drivers might not work correctly with all applications when accessing Amazon Redshift. For more information about drivers and configuring connections, see JDBC and ODBC Drivers for Amazon Redshift in the Amazon Redshift Cluster Management Guide.

To avoid client-side out of memory errors when retrieving large data sets using JDBC, you can enable your client to fetch data in batches by setting the JDBC fetch size parameter. For more information, see Setting the JDBC fetch size parameter (p. 196).

Amazon Redshift does not recognize the JDBC maxRows parameter. Instead, specify a LIMIT (p. 383) clause to restrict the result set. You can also use an OFFSET (p. 383) clause to skip to a specific starting point in the result set.

### Features that are implemented differently

Many Amazon Redshift SQL language elements have different performance characteristics and use syntax and semantics and that are quite different from the equivalent PostgreSQL implementation.

**Important**

Do not assume that the semantics of elements that Redshift and PostgreSQL have in common are identical. Make sure to consult the Amazon Redshift Developer Guide SQL Commands (p.256) to understand the often subtle differences.

One example in particular is the VACUUM (p. 411) command, which is used to clean up and reorganize tables. VACUUM functions differently and uses a different set of parameters than the PostgreSQL version. See Vacuuming tables (p. 161) for more about information about using VACUUM in Amazon Redshift.

Often, database management and administration features and tools are different as well. For example, Amazon Redshift maintains a set of system tables and views that provide information about how the system is functioning. See System tables and views (p. 582) for more information.

The following list includes some examples of SQL features that are implemented differently in Amazon Redshift.

- CREATE TABLE (p. 310)
Amazon Redshift does not support tablespaces, table partitioning, inheritance, and certain constraints. The Amazon Redshift implementation of CREATE TABLE enables you to define the sort and distribution algorithms for tables to optimize parallel processing.

- **ALTER TABLE (p. 261)**

  ALTER COLUMN actions are not supported.

  ADD COLUMN supports adding only one column in each ALTER TABLE statement.

- **COPY (p. 276)**

  The Amazon Redshift COPY command is highly specialized to enable the loading of data from Amazon S3 buckets and Amazon DynamoDB tables and to facilitate automatic compression. See the Loading Data (p. 117) section and the COPY command reference for details.

- **SELECT (p. 359)**

  ORDER BY ... NULLS FIRST/LAST is not supported.

- **INSERT (p. 349), UPDATE (p. 407), and DELETE (p. 331)**

  WITH is not supported.

- **VACUUM (p. 411)**

  The parameters for VACUUM are entirely different. For example, the default VACUUM operation in PostgreSQL simply reclaims space and makes it available for re-use; however, the default VACUUM operation in Amazon Redshift is VACUUM FULL, which reclaims disk space and resorts all rows.

### Unsupported PostgreSQL features

These PostgreSQL features are not supported in Amazon Redshift.

**Important**

Do not assume that the semantics of elements that Redshift and PostgreSQL have in common are identical. Make sure to consult the [Amazon Redshift Developer Guide SQL Commands (p. 256)](http://www.postgresql.org/docs/8.0/static/ddl-system-columns.html) to understand the often subtle differences.

- Only the 8.x version of the PostgreSQL query tool `psql` is supported.
- Table partitioning (range and list partitioning)
- Tablespaces
- Constraints
  - Unique
  - Foreign key
  - Primary key
  - Check constraints
  - Exclusion constraints

Unique, primary key, and foreign key constraints are permitted, but they are informational only. They are not enforced by the system, but they are used by the query planner.

- Inheritance
- Postgres system columns

Amazon Redshift SQL does not implicitly define system columns. However, the PostgreSQL system column names cannot be used as names of user-defined columns. See [http://www.postgresql.org/docs/8.0/static/ddl-system-columns.html](http://www.postgresql.org/docs/8.0/static/ddl-system-columns.html)

- Indexes
- NULLS clause in Window functions
• Collations

Amazon Redshift does not support locale-specific or user-defined collation sequences. See Collation sequences (p. 235).

• Value expressions
  • Subscripted expressions
  • Array constructors
  • Row constructors

• User-defined functions and stored procedures
• Triggers
• Management of External Data (SQL/MED)

• Table functions
  • VALUES list used as constant tables
  • Recursive common table expressions

• Sequences
• Full text search

### Unsupported PostgreSQL data types

Generally, if a query attempts to use an unsupported data type, including explicit or implicit casts, it will return an error. However, some queries that use unsupported data types will run on the leader node but not on the compute nodes. See SQL functions supported on the leader node (p. 205).

For a list of the supported data types, see Data types (p. 214).

These PostgreSQL data types are not supported in Amazon Redshift.

• Arrays
• BIT, BIT VARYING
• BYTEA
• Composite Types
• Date/Time Types
  • INTERVAL
  • TIME
  • TIMESTAMP WITH TIMEZONE
• Enumerated Types
• Geometric Types
• JSON
• Network Address Types
• Numeric Types
  • SERIAL, BIGSERIAL, SMALLSERIAL
  • MONEY
• Object Identifier Types
• Pseudo-Types
• Range Types
• Text Search Types
  • TXID_SNAPSHOT
• UUID
• XML
Unsupported PostgreSQL functions

Many functions that are not excluded have different semantics or usage. For example, some supported functions will run only on the leader node. Also, some unsupported functions will not return an error when run on the leader node. The fact that these functions do not return an error in some cases should not be taken to indicate that the function is supported by Amazon Redshift.

**Important**

Do not assume that the semantics of elements that Redshift and PostgreSQL have in common are identical. Make sure to consult the *Amazon Redshift Developers Guide SQL Commands* (p. 256) to understand the often subtle differences.

For more information, see SQL functions supported on the leader node (p. 205).

These PostgreSQL functions are not supported in Amazon Redshift.

- Access privilege inquiry functions
- Advisory lock functions
- Aggregate functions
  - STRING_AGG()
  - ARRAY_AGG()
  - EVERY()
  - XML_AGG()
  - CORR()
  - COVAR_POP()
  - COVAR_SAMP()
  - REGR_AVGX(), REGR_AVGY()
  - REGR_COUNT()
  - REGR_INTERCEPT()
  - REGR_R2()
  - REGR_SLOPE()
  - REGR_SXX(), REGR_SXY(), REGR_SYY()
- Array functions and operators
- Backup control functions
- Comment information functions
- Database object location functions
- Database object size functions
- Date/Time functions and operators
  - CLOCK_TIMESTAMP()
  - JUSTIFY_DAYS(), JUSTIFY_HOURS(), JUSTIFY_INTERVAL()
  - TRANSACTION_TIMESTAMP()
- Data type formatting functions
  - TO_TIMESTAMP()
- ENUM support functions
- Geometric functions and operators
- Generic file access functions
- GREATEST(), LEAST()
- IS DISTINCT FROM
- Network address functions and operators
- Mathematical functions
• DIV()
• SETSEED()
• WIDTH_BUCKET()

Set returning functions
• GENERATE_SERIES()
• GENERATE_SUBSCRIPTS()

Range functions and operators
• Recovery control functions
• Recovery information functions
• Schema visibility inquiry functions
• Server signaling functions
• Snapshot synchronization functions
• Sequence manipulation functions

String functions
• BIT_LENGTH()
• OVERLAY()
• CONVERT(), CONVERT_FROM(), CONVERT_TO()
• ENCODE()
• FORMAT()
• QUOTE_NONNULL()
• REGEXP_MATCHES()
• REGEXP_REPLACE()
• REGEXP_SPLIT_TO_ARRAY()
• REGEXP_SPLIT_TO_TABLE()
• SUBSTR()
• TRANSLATE()

System catalog information functions
• System information functions
• CURRENT_CATALOG CURRENT_QUERY()
• INET_CLIENT_ADDR()
• INET_CLIENT_PORT()
• INET_SERVER_ADDR() INET_SERVER_PORT()
• PG_CONF_LOAD_TIME()
• PG_IS_OTHER_TEMP_SCHEMA()
• PG_LISTENING_CHANNELS()
• PG_MY_TEMP_SCHEMA()
• PG_POSTMASTER_START_TIME()
• PG_TRIGGER_DEPTH()

Text search functions and operators
• Transaction IDs and snapshots functions
• Trigger functions
• Window Functions
• PERCENT_RANK()
• CUME_DIST()
• XML functions
Using SQL

Topics
- SQL reference conventions (p. 212)
- Basic elements (p. 212)
- Expressions (p. 235)
- Conditions (p. 239)

The SQL language consists of commands and functions that you use to work with databases and database objects. The language also enforces rules regarding the use of data types, expressions, and literals.

SQL reference conventions

This section explains the conventions that are used to write the synopses for the SQL expressions, commands, and functions described in the SQL reference section.

<table>
<thead>
<tr>
<th>Character</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAPS</td>
<td>Words in capital letters are key words.</td>
</tr>
<tr>
<td>[]</td>
<td>Square brackets denote optional arguments. Multiple arguments in square brackets indicate that you can choose any number of the arguments. In addition, arguments in brackets on separate lines indicate that the Amazon Redshift parser expects the arguments to be in the order that they are listed in the synopsis. For an example, see SELECT (p. 359).</td>
</tr>
<tr>
<td>{}</td>
<td>Braces indicate that you are required to choose one of the arguments inside the braces.</td>
</tr>
<tr>
<td></td>
<td>Pipes indicate that you can choose between the arguments.</td>
</tr>
<tr>
<td>italics</td>
<td>Words in italics indicate placeholders. You must insert the appropriate value in place of the word in italics.</td>
</tr>
<tr>
<td>. . .</td>
<td>An ellipsis indicates that you can repeat the preceding element.</td>
</tr>
<tr>
<td>'</td>
<td>Words in single quotes indicate that you must type the quotes.</td>
</tr>
</tbody>
</table>

Basic elements

Topics
- Names and identifiers (p. 213)
- Literals (p. 214)
- Nulls (p. 214)
- Data types (p. 214)
- Collation sequences (p. 235)

This section covers the rules for working with database object names, literals, nulls, and data types.
Names and identifiers

Names identify database objects, including tables and columns, as well as users and passwords. The terms *name* and *identifier* can be used interchangeably. There are two types of identifiers, standard identifiers and quoted or delimited identifiers. Standard and delimited identifiers are case-insensitive and are folded to lower case. Identifiers must consist of only ASCII characters. Multi-byte characters are not supported.

**Standard identifiers**

Standard SQL identifiers adhere to a set of rules and must:

- Contain only ASCII letters, digits, underscore characters (_), or dollar signs ($).
- Begin with an alphabetic character or underscore character. Subsequent characters may include letters, digits, underscores, or dollar signs.
- Be between 1 and 127 characters in length, not including quotes for delimited identifiers.
- Contain no quotation marks and no spaces.
- Not be a reserved SQL key word.

**Delimited identifiers**

Delimited identifiers (also known as quoted identifiers) begin and end with double quotation marks ("."). If you use a delimited identifier, you must use the double quotation marks for every reference to that object. The identifier can contain any character other than the double quote itself. Therefore, you can create column or table names with any string, including otherwise illegal characters, such as spaces or the percent symbol.

Delimited identifiers are case-insensitive, and are folded to lower case. To use a double quote in a string, you must precede it with another double quote character.

**Examples**

This table shows examples of delimited identifiers, the resulting output, and a discussion:

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Result</th>
<th>Discussion</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;group&quot;</td>
<td>group</td>
<td>GROUP is a reserved word, so usage of it within an identifier requires double quotes.</td>
</tr>
<tr>
<td>&quot;&quot;WHERE&quot;&quot;</td>
<td>&quot;where&quot;</td>
<td>WHERE is also a reserved word. To include quotation marks in the string, escape them with additional quotation marks.</td>
</tr>
<tr>
<td>&quot;This name&quot;</td>
<td>this name</td>
<td>Double quotes are required in order to preserve the space.</td>
</tr>
<tr>
<td>&quot;This &quot;IS IT&quot;&quot;</td>
<td>this &quot;is it&quot;</td>
<td>The quotes surrounding IS IT must each be preceded by an extra quote in order to become part of the name.</td>
</tr>
</tbody>
</table>

To create a table named group with a column named this "is it":

```sql
create table "group" (
  "This "IS IT"" char(10));
```

The following queries return the same result:
select "This ""IS IT"""
from "group";
this "is it"
-------------
(0 rows)

select "this ""is it"""
from "group";
this "is it"
-------------
(0 rows)

The following fully qualified table.column syntax also returns the same result:

select "group"."this ""is it"""
from "group";
this "is it"
-------------
(0 rows)

**Literals**

A literal or constant is a fixed data value, composed of a sequence of characters or a numeric constant. Amazon Redshift supports several types of literals, including:

- Numeric literals for integer, decimal, and floating-point numbers
- Character literals, also referred to as strings, character strings, or character constants
- Datetime and interval literals, used with datetime data types

**Nulls**

If a column in a row is missing, unknown, or not applicable, it is a null value or is said to contain null. Nulls can appear in fields of any data type that are not restricted by primary key or NOT NULL constraints. A null is not equivalent to the value zero or to an empty string.

Any arithmetic expression containing a null always evaluates to a null. All operators except concatenation return a null when given a null argument or operand.

To test for nulls, use the comparison conditions IS NULL and IS NOT NULL. Because null represents a lack of data, a null is not equal or unequal to any value or to another null.

**Data types**

**Topics**

- Multi-byte characters (p. 215)
- Numeric types (p. 216)
- Character types (p. 223)
- Datetime types (p. 226)
- Boolean type (p. 231)
Each value that Amazon Redshift stores or retrieves has a data type with a fixed set of associated properties. Data types are declared when tables are created. A data type constrains the set of values that a column or argument can contain.

The following table lists the data types that you can use in Amazon Redshift tables.

<table>
<thead>
<tr>
<th>Data type</th>
<th>Aliases</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMALLINT</td>
<td>INT2</td>
<td>Signed two-byte integer</td>
</tr>
<tr>
<td>INTEGER</td>
<td>INT, INT4</td>
<td>Signed four-byte integer</td>
</tr>
<tr>
<td>BIGINT</td>
<td>INT8</td>
<td>Signed eight-byte integer</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>NUMERIC</td>
<td>Exact numeric of selectable precision</td>
</tr>
<tr>
<td>REAL</td>
<td>FLOAT4</td>
<td>Single precision floating-point number</td>
</tr>
<tr>
<td>DOUBLE PRECISION</td>
<td>FLOAT8, FLOAT</td>
<td>Double precision floating-point number</td>
</tr>
<tr>
<td>BOOLEAN</td>
<td>BOOL</td>
<td>Logical Boolean (true/false)</td>
</tr>
<tr>
<td>CHAR</td>
<td>CHARACTER, NCHAR, BPCHAR</td>
<td>Fixed-length character string</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>CHARACTER VARYING, NVARCHAR, TEXT</td>
<td>Variable-length character string with a user-defined limit</td>
</tr>
<tr>
<td>DATE</td>
<td></td>
<td>Calendar date (year, month, day)</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td></td>
<td>Date and time (without time zone)</td>
</tr>
</tbody>
</table>

**Multi-byte characters**

The VARCHAR data type supports UTF-8 multi-byte characters up to a maximum of four bytes. Five-byte or longer characters are not supported. To calculate the size of a VARCHAR column that contains multi-byte characters, multiply the number of characters by the number of bytes per character. For example, if a string has four Chinese characters, and each character is three bytes long, then you will need a VARCHAR(12) column to store the string.

VARCHAR does not support the following invalid UTF-8 codepoints:

- 0xD800 - 0xDFFF
  (Byte sequences: ED A0 80 - ED BF BF)
- 0xFD00 - 0xFDFF, 0xFEFF, and 0xFFFF
  (Byte sequences: EF B7 90 - EF B7 AF, EF BF BE, and EF BF BF)

The CHAR data type does not support multi-byte characters.
Numeric types

Topics
- Integer types (p. 216)
- DECIMAL or NUMERIC type (p. 216)
- Notes about using 128-bit DECIMAL or NUMERIC columns (p. 217)
- Floating-point types (p. 217)
- Computations with numeric values (p. 218)
- Integer and floating-point literals (p. 221)
- Examples with numeric types (p. 222)

Numeric data types include integers, decimals, and floating-point numbers.

Integer types

Use the SMALLINT, INTEGER, and BIGINT data types to store whole numbers of various ranges. You cannot store values outside of the allowed range for each type.

<table>
<thead>
<tr>
<th>Name</th>
<th>Storage</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMALLINT or INT2</td>
<td>2 bytes</td>
<td>-32768 to +32767</td>
</tr>
<tr>
<td>INTEGER, INT, or INT4</td>
<td>4 bytes</td>
<td>-2147483648 to +2147483647</td>
</tr>
<tr>
<td>BIGINT or INT8</td>
<td>8 bytes</td>
<td>-922372036854775808 to 922372036854775807</td>
</tr>
</tbody>
</table>

DECIMAL or NUMERIC type

Use the DECIMAL or NUMERIC data type to store values with a user-defined precision. The DECIMAL and NUMERIC keywords are interchangeable. In this document, decimal is the preferred term for this data type. The term numeric is used generically to refer to integer, decimal, and floating-point data types.

<table>
<thead>
<tr>
<th>Storage</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable, up to 128 bits for uncompressed DECIMAL types.</td>
<td>128-bit signed integers with up to 38 digits of precision.</td>
</tr>
</tbody>
</table>

Define a DECIMAL column in a table by specifying a precision and scale:

```
decimal(precision, scale)
```

**precision**

The total number of significant digits in the whole value: the number of digits on both sides of the decimal point. For example, the number 48.2891 has a precision of 6 and a scale of 4. The default precision, if not specified, is 18. The maximum precision is 38.

If the number of digits to the left of the decimal point in an input value exceeds the precision of the column minus its scale, the value cannot be copied into the column (or inserted or updated). This rule applies to any value that falls outside the range of the column definition. For example, the allowed range of values for a numeric(5,2) column is -999.99 to 999.99.
**scale**

The number of decimal digits in the fractional part of the value, to the right of the decimal point. Integers have a scale of zero. In a column specification, the scale value must be less than or equal to the precision value. The default scale, if not specified, is 0. The maximum scale is 37.

If the scale of an input value that is loaded into a table is greater than the scale of the column, the value is rounded to the specified scale. For example, the PRICEPAID column in the SALES table is a DECIMAL(8,2) column. If a DECIMAL(8,4) value is inserted into the PRICEPAID column, the value is rounded to a scale of 2.

```sql
insert into sales
values (0, 8, 1, 1, 2000, 14, 5, 4323.8951, 11.00, null);

select pricepaid, salesid from sales where salesid=0;

<table>
<thead>
<tr>
<th>pricepaid</th>
<th>salesid</th>
</tr>
</thead>
<tbody>
<tr>
<td>4323.90</td>
<td>0</td>
</tr>
</tbody>
</table>

(1 row)
```

However, results of explicit casts of values selected from tables are not rounded.

**Note**

The maximum positive value that you can insert into a DECIMAL(19,0) column is 9223372036854775807 (2\(63\)-1). The maximum negative value is \(-9223372036854775807\). For example, an attempt to insert the value 9999999999999999999 (19 nines) will cause an overflow error. Regardless of the placement of the decimal point, the largest string that Amazon Redshift can represent as a DECIMAL number is 9223372036854775807. For example, the largest value that you can load into a DECIMAL(19,18) column is 9.223372036854775807. These rules derive from the internal storage of DECIMAL values as 8-byte integers. Amazon Redshift recommends that you do not define DECIMAL values with 19 digits of precision unless that precision is necessary.

**Notes about using 128-bit DECIMAL or NUMERIC columns**

Note the following restrictions on using DECIMAL or NUMERIC columns with a precision that is greater than 19:

- Amazon Redshift does not implicitly convert 64-bit DECIMAL values to 128-bit values. You must explicitly convert 64-bit values to a higher precision by using functions such as the CAST and CONVERT functions (p. 556).
- Do not arbitrarily assign maximum precision to DECIMAL columns unless you are certain that your application requires that precision. 128-bit values use twice as much disk space as 64-bit values and can slow down query execution time.

**Floating-point types**

Use the REAL and DOUBLE PRECISION data types to store numeric values with variable precision. These types are inexact types, meaning that some values are stored as approximations, such that storing and returning a specific value may result in slight discrepancies. If you require exact storage and calculations (such as for monetary amounts), use the DECIMAL data type.

<table>
<thead>
<tr>
<th>Name</th>
<th>Storage</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>REAL or FLOAT4</td>
<td>4 bytes</td>
<td>6 significant digits of precision</td>
</tr>
</tbody>
</table>
For example, note the results of the following inserts into a REAL column:

```sql
create table real1(realcol real);
insert into real1 values(12345.12345);
insert into real1 values(123456.12345);
select * from real1;
realcol
---------
123456
12345.1
(2 rows)
```

These inserted values are truncated to meet the limitation of 6 significant digits of precision for REAL columns.

**Computations with numeric values**

In this context, *computation* refers to binary mathematical operations: addition, subtraction, multiplication, and division. This section describes the expected return types for these operations, as well as the specific formula that is applied to determine precision and scale when DECIMAL data types are involved.

When numeric values are computed during query processing, you might encounter cases where the computation is impossible and the query returns a numeric overflow error. You might also encounter cases where the scale of computed values varies or is unexpected. For some operations, you can use explicit casting (type promotion) or Amazon Redshift configuration parameters to work around these problems.

For information about the results of similar computations with SQL functions, see Aggregate functions (p. 415).

**Return types for computations**

Given the set of numeric data types supported in Amazon Redshift, the following table shows the expected return types for addition, subtraction, multiplication, and division operations. The first column on the left side of the table represents the first operand in the calculation, and the top row represents the second operand.
**Precision and scale of computed DECIMAL results**

The following table summarizes the rules for computing resulting precision and scale when mathematical operations return DECIMAL results. In this table, \( p_1 \) and \( s_1 \) represent the precision and scale of the first operand in a calculation and \( p_2 \) and \( s_2 \) represent the precision and scale of the second operand. (Regardless of these calculations, the maximum result precision is 38, and the maximum result scale is 38.)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result precision and scale</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ or -</td>
<td>Scale = max(s1, s2)</td>
</tr>
<tr>
<td></td>
<td>Precision = max(p1-s1, p2-s2)+1+scale</td>
</tr>
<tr>
<td>*</td>
<td>Scale = s1+s2</td>
</tr>
<tr>
<td></td>
<td>Precision = p1+p2+1</td>
</tr>
<tr>
<td>/</td>
<td>Scale = max(4, s1+p2-s2+1)</td>
</tr>
<tr>
<td></td>
<td>Precision = p1-s1+ s2+scale</td>
</tr>
</tbody>
</table>

For example, the PRICEPAID and COMMISSION columns in the SALES table are both DECIMAL(8,2) columns. If you divide PRICEPAID by COMMISSION (or vice versa), the formula is applied as follows:

\[
\text{Precision} = 8-2 + 2 + \max(4,2+8-2+1) \\
= 6 + 2 + 9 = 17 \\
\text{Scale} = \max(4,2+8-2+1) = 9 \\
\text{Result} = \text{DECIMAL}(17,9)
\]

The following calculation is the general rule for computing the resulting precision and scale for operations performed on DECIMAL values with set operators such as UNION, INTERSECT, and EXCEPT or functions such as COALESCE and DECODE:

\[
\text{Scale} = \max(s1,s2) \\
\text{Precision} = \min(\max(p1-s1,p2-s2)+\text{scale},19)
\]

For example, a DEC1 table with one DECIMAL(7,2) column is joined with a DEC2 table with one DECIMAL(15,3) column to create a DEC3 table. The schema of DEC3 shows that the column becomes a NUMERIC(15,3) column.

create table dec3 as select * from dec1 union select * from dec2;

Result

```
select "column", type, encoding, distkey, sortkey
from pg_table_def where tablename = 'dec3';
```

```
column |     type      | encoding | distkey | sortkey
-------+---------------+----------+---------+---------
c1     | numeric(15,3) | none     | f       | 0
```
In the above example, the formula is applied as follows:

\[
\begin{align*}
\text{Precision} &= \min(\max(7-2,15-3) + \max(2,3), 19) \\
&= 12 + 3 = 15 \\
\text{Scale} &= \max(2,3) = 3 \\
\text{Result} &= \text{DECIMAL}(15,3)
\end{align*}
\]

**Notes on division operations**

For division operations, divide-by-zero conditions return errors.

The scale limit of 100 is applied after the precision and scale are calculated. If the calculated result scale is greater than 100, division results are scaled as follows:

- Precision = precision - (scale - max_scale)
- Scale = max_scale

If the calculated precision is greater than the maximum precision (38), the precision is reduced to 38, and the scale becomes the result of: \(\max(\text{scale} - (\text{precision} - 38), \min(4, 100))\)

**Overflow conditions**

Overflow is checked for all numeric computations. DECIMAL data with a precision of 19 or less is stored as 64-bit integers. DECIMAL data with a precision that is greater than 19 is stored as 128-bit integers. The maximum precision for all DECIMAL values is 38, and the maximum scale is 37. Overflow errors occur when a value exceeds these limits, which apply to both intermediate and final result sets:

- Explicit casting results in run-time overflow errors when specific data values do not fit the requested precision or scale specified by the cast function. For example, you cannot cast all values from the PRICEPAID column in the SALES table (a DECIMAL(8,2) column) and return a DECIMAL(7,3) result:

  ```sql
  select pricepaid::decimal(7,3) from sales;
  ERROR: Numeric data overflow (result precision)
  ```

  This error occurs because some of the larger values in the PRICEPAID column cannot be cast.

- Multiplication operations produce results in which the result scale is the sum of the scale of each operand. If both operands have a scale of 4, for example, the result scale is 8, leaving only 10 digits for the left side of the decimal point. Therefore, it is relatively easy to run into overflow conditions when multiplying two large numbers that both have significant scale.

- Implicitly casting 64-bit DECIMAL values to a higher precision causes numeric overflow errors. To avoid overflow errors, explicitly cast 64-bit DECIMAL values to a higher precision by using functions such as the **CAST** and **CONVERT** functions (p.556). For example, the PRICEPAID column in the SALES table is a DECIMAL(8,2) column. To multiply the values in this column by a value that increases the precision to greater than 19 digits, such as 10000000000000000000000, cast the expression as follows:

  ```sql
  select salesid, pricepaid::decimal(38,2) * 10000000000000000000000 as value from sales where salesid=2;
  ```

  ```sql
  salesid | value
  --------+---------------------------
  2 | 7600000000000000000000.00
  (1 row)
  ```
**Numeric calculations with INTEGER and DECIMAL types**

When one of the operands in a calculation has an INTEGER data type and the other operand is DECIMAL, the INTEGER operand is implicitly cast as a DECIMAL:

- INT2 (SMALLINT) is cast as DECIMAL(5,0)
- INT4 (INTEGER) is cast as DECIMAL(10,0)
- INT8 (BIGINT) is cast as DECIMAL(19,0)

For example, if you multiply SALES.COMMISSION, a DECIMAL(8,2) column, and SALES.QTYSOLD, a SMALLINT column, this calculation is cast as:

```
DECIMAL(8,2) * DECIMAL(5,0)
```

**Integer and floating-point literals**

Literals or constants that represent numbers can be integer or floating-point.

**Integer literals**

An integer constant is a sequence of the digits 0-9, with an optional positive (+) or negative (-) sign preceding the digits.

**Synopsis**

```
[ + | - ] digit ... 
```

**Examples**

Valid integers include:

- 23
- -555
- +17

**Floating-point literals**

Floating-point literals (also referred to as decimal, numeric, or fractional literals) are sequences of digits that can include a decimal point, and optionally the exponent marker (e).

**Synopsis**

```
[ + | - ] digit ... [ . ] [ digit ...]
[ e | E [ + | - ] digit ... ]
```

**Arguments**

- `e` or `E` indicates that the number is specified in scientific notation.

**Examples**

Valid floating-point literals include:
Examples with numeric types

**CREATE TABLE statement**

The following CREATE TABLE statement demonstrates the declaration of different numeric data types:

```sql
create table film (
    film_id integer,
    language_id smallint,
    original_language_id smallint,
    rental_duration smallint default 3,
    rental_rate numeric(4,2) default 4.99,
    length smallint,
    replacement_cost real default 25.00);
```

**Attempt to insert an integer that is out of range**

The following example attempts to insert the value 33000 into an INT column.

```sql
insert into film(language_id) values(33000);
```

The range for INT is -32768 to +32767, so Amazon Redshift returns an error.

An error occurred when executing the SQL command:
```
insert into film(language_id) values(33000)
```
ERROR: smallint out of range [SQL State=22003]

**Insert a decimal value into an integer column**

The following example inserts a decimal value into an INT column.

```sql
insert into film(language_id) values(1.5);
```

This value is inserted but rounded up to the integer value 2.

**Insert a decimal that succeeds because its scale is rounded**

The following example inserts a decimal value that has higher precision than the column.

```sql
insert into film(rental_rate) values(35.512);
```

In this case, the value 35.51 is inserted into the column.

**Attempt to insert a decimal value that is out of range**

In this case, the value 350.10 is out of range. The number of digits for values in DECIMAL columns is equal to the column's precision minus its scale (4 minus 2 for the RENTAL_RATE column). In other words, the allowed range for a `DECIMAL(4,2)` column is -99.99 through 99.99.
insert into film(rental_rate) values (350.10);
ERROR: numeric field overflow
DETAIL: The absolute value is greater than or equal to 10^2 for field with precision 4, scale 2.

Insert variable-precision values into a REAL column

The following example inserts variable-precision values into a REAL column.

insert into film(replacement_cost) values(1999.99);
insert into film(replacement_cost) values(19999.99);
select replacement_cost from film;
replacement_cost
--------------
20000
1999.99
...

The value 19999.99 is converted to 20000 to meet the 6-digit precision requirement for the column. The value 1999.99 is loaded as is.

Character types

Topics
• Storage and ranges (p. 223)
• CHAR or CHARACTER (p. 224)
• VARCHAR or CHARACTER VARYING (p. 224)
• NCHAR and NVARCHAR types (p. 224)
• TEXT and BPCHAR types (p. 224)
• Significance of trailing blanks (p. 225)
• Examples with character types (p. 225)

Character data types include CHAR (character) and VARCHAR (character varying).

Storage and ranges

CHAR and VARCHAR data types are defined in terms of bytes, not characters. A CHAR column can only contain single-byte characters, so a CHAR(10) column can contain a string with a maximum length of 10 bytes. A VARCHAR can contain multi-byte characters, up to a maximum of four bytes per character. For example, a VARCHAR(12) column can contain 12 single-byte characters, 6 two-byte characters, 4 three-byte characters, or 3 four-byte characters.

<table>
<thead>
<tr>
<th>Name</th>
<th>Storage</th>
<th>Range (width of column)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR, CHARACTER or NCHAR</td>
<td>Length of string, including trailing blanks (if any)</td>
<td>4096 bytes</td>
</tr>
</tbody>
</table>
### Table: Storage and Range for Different Data Types

<table>
<thead>
<tr>
<th>Name</th>
<th>Storage</th>
<th>Range (width of column)</th>
</tr>
</thead>
<tbody>
<tr>
<td>VARCHAR, CHARACTER VARYING, or NVARCHAR</td>
<td>4 bytes + total bytes for characters, where each character can be 1 to 4 bytes.</td>
<td>65535 bytes (64K -1)</td>
</tr>
<tr>
<td>BPCHAR</td>
<td>Converted to fixed-length CHAR(256).</td>
<td>256 bytes</td>
</tr>
<tr>
<td>TEXT</td>
<td>Converted to VARCHAR(256).</td>
<td>260 bytes</td>
</tr>
</tbody>
</table>

### Note

The CREATE TABLE syntax supports the MAX keyword for character data types. For example:

```sql
create table test(col1 varchar(max));
```

The MAX setting defines the width of the column as 4096 bytes for CHAR or 65535 bytes for VARCHAR.

### CHAR or CHARACTER

Use a CHAR or CHARACTER column to store fixed-length strings. These strings are padded with blanks, so a CHAR(10) column always occupies 10 bytes of storage.

```sql
char(10)
```

A CHAR column without a length specification results in a CHAR(1) column.

### VARCHAR or CHARACTER VARYING

Use a VARCHAR or CHARACTER VARYING column to store variable-length strings with a fixed limit. These strings are not padded with blanks, so a VARCHAR(120) column consists of a maximum of 120 single-byte characters, 60 two-byte characters, 40 three-byte characters, or 30 four-byte characters.

```sql
varchar(120)
```

If you use the VARCHAR data type without a length specifier, the default length is 256.

### NCHAR and NVARCHAR types

You can create columns with the NCHAR and NVARCHAR types (also known as NATIONAL CHARACTER and NATIONAL CHARACTER VARYING types). These types are converted to CHAR and VARCHAR types, respectively, and are stored in the specified number of bytes.

An NCHAR column without a length specification is converted to a CHAR(1) column.

An NVARCHAR column without a length specification is converted to a VARCHAR(256) column.

### TEXT and BPCHAR types

You can create an Amazon Redshift table with a TEXT column, but it is converted to a VARCHAR(256) column that accepts variable-length values with a maximum of 256 characters.
You can create an Amazon Redshift column with a BPCHAR (blank-padded character) type, which Amazon Redshift converts to a fixed-length CHAR(256) column.

**Significance of trailing blanks**

Both CHAR and VARCHAR data types store strings up to \( n \) bytes in length. An attempt to store a longer string into a column of these types results in an error, unless the extra characters are all spaces (blanks), in which case the string is truncated to the maximum length. If the string is shorter than the maximum length, CHAR values are padded with blanks, but VARCHAR values store the string without blanks.

Trailing blanks in CHAR values are always semantically insignificant. They are disregarded when you compare two CHAR values, not included in LENGTH calculations, and removed when you convert a CHAR value to another string type.

Trailing spaces in VARCHAR and CHAR values are treated as semantically insignificant when values are compared.

Length calculations return the length of VARCHAR character strings with trailing spaces included in the length. Trailing blanks are not counted in the length for fixed-length character strings.

**Examples with character types**

**CREATE TABLE statement**

The following CREATE TABLE statement demonstrates the use of VARCHAR and CHAR data types:

```
create table address(
    address_id integer,
    address1 varchar(100),
    address2 varchar(50),
    district varchar(20),
    city_name char(20),
    state char(2),
    postal_code char(5)
);
```

The following examples use this table.

**Trailing blanks in variable-length character strings**

Because ADDRESS1 is a VARCHAR column, the trailing blanks in the second inserted address are semantically insignificant. In other words, these two inserted addresses match.

```
insert into address(address1) values('9516 Magnolia Boulevard ');
insert into address(address1) values('9516 Magnolia Boulevard');
```

```
select count(*) from address
where address1='9516 Magnolia Boulevard';
```

```
count
-------
  2
(1 row)
```

If the ADDRESS1 column were a CHAR column and the same values were inserted, the COUNT(\(*)\) query would recognize the character strings as the same and return 2.
Results of the LENGTH function

The LENGTH function recognizes trailing blanks in VARCHAR columns:

```sql
SELECT length(address1) FROM address;
```

<table>
<thead>
<tr>
<th>length</th>
</tr>
</thead>
<tbody>
<tr>
<td>23</td>
</tr>
<tr>
<td>25</td>
</tr>
</tbody>
</table>

(2 rows)

A value of Augusta in the CITY_NAME column, which is a CHAR column, would always return a length of 7 characters, regardless of any trailing blanks in the input string.

Values that exceed the length of the column

Character strings are not truncated to fit the declared width of the column:

```sql
INSERT INTO address(city_name) VALUES('City of South San Francisco');
ERROR: value too long for type character(20)
```

A workaround for this problem is to cast the value to the size of the column:

```sql
INSERT INTO address(city_name) VALUES('City of South San Francisco'::char(20));
```

In this case, the first 20 characters of the string (City of South San Fr) would be loaded into the column.

Datetime types

Topics
- Storage and ranges (p. 226)
- DATE (p. 226)
- TIMESTAMP (p. 227)
- Examples with datetime types (p. 227)
- Date and timestamp literals (p. 228)
- Interval literals (p. 229)

Datetime data types include DATE and TIMESTAMP.

Storage and ranges

<table>
<thead>
<tr>
<th>Name</th>
<th>Storage</th>
<th>Range</th>
<th>Resolution</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATE</td>
<td>4 bytes</td>
<td>4713 BC to 5874897 AD</td>
<td>1 day</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>8 bytes</td>
<td>4713 BC to 5874897 AD</td>
<td>1 microsecond</td>
</tr>
</tbody>
</table>

DATE

Use the DATE data type to store simple calendar dates without timestamps.
**TIMESTAMP**

Use the TIMESTAMP data type to store complete timestamp values that include the date and the time of day.

TIMESTAMP columns store values with up to a maximum of 6 digits of precision for fractional seconds.

If you insert a date into a timestamp column, or a date with a partial timestamp, the value is implicitly converted into a full timestamp value with default values (00) for missing hours, minutes, and seconds.

TIMESTAMP values are UTC, not local time, in both user tables and Amazon Redshift system tables.

**Note**

Timestamps with time zones are not supported.

**Examples with datetime types**

**Date examples**

Insert dates that have different formats and display the output:

```sql
create table datetable (start_date date, end_date date);

insert into datetable values ('2008-06-01','2008-12-31');
insert into datetable values ('Jun 1,2008','20081231');

select * from datetable order by 1;

<table>
<thead>
<tr>
<th>start_date</th>
<th>end_date</th>
</tr>
</thead>
<tbody>
<tr>
<td>2008-06-01</td>
<td>2008-12-31</td>
</tr>
<tr>
<td>2008-06-01</td>
<td>2008-12-31</td>
</tr>
</tbody>
</table>
```

If you attempt to insert a timestamp value into a DATE column, the time literal is truncated and the date itself is loaded.

**Timestamp examples**

If you insert a date into a TIMESTAMP column, the time defaults to midnight. For example, if you insert the literal `20081231`, the stored value is `2008-12-31 00:00:00`.

Insert timestamps that have different formats and display the output:

```sql
create table tstamp(timeofday timestamp);

insert into tstamp values('20081231 09:59:59');
insert into tstamp values('20081231 18:20');

select * from tstamp order by 1;

timeofday
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2008-12-31 09:59:59</td>
</tr>
<tr>
<td>2008-12-31 18:20</td>
</tr>
</tbody>
</table>
```
Date and timestamp literals

Dates

The following input dates are all valid examples of literal date values that you can load into Amazon Redshift tables. The default MDY DateStyle mode is assumed to be in effect, which means that the month value precedes the day value in strings such as **1999-01-08** and **01/02/00**.

**Note**

A date or timestamp literal must be enclosed in quotes when you load it into a table.

<table>
<thead>
<tr>
<th>Input date</th>
<th>Full date</th>
</tr>
</thead>
<tbody>
<tr>
<td>January 8, 1999</td>
<td>January 8, 1999</td>
</tr>
<tr>
<td>1999-01-08</td>
<td>January 8, 1999</td>
</tr>
<tr>
<td>1/8/1999</td>
<td>January 8, 1999</td>
</tr>
<tr>
<td>01/02/00</td>
<td>January 2, 2000</td>
</tr>
<tr>
<td>2000-Jan-31</td>
<td>January 31, 2000</td>
</tr>
<tr>
<td>Jan-31-2000</td>
<td>January 31, 2000</td>
</tr>
<tr>
<td>31-Jan-2000</td>
<td>January 31, 2000</td>
</tr>
<tr>
<td>20080215</td>
<td>February 15, 2008</td>
</tr>
<tr>
<td>080215</td>
<td>February 15, 2008</td>
</tr>
<tr>
<td>2008.366</td>
<td>December 31, 2008 (3-digit part of date must be between 001 and 366)</td>
</tr>
</tbody>
</table>

Timestamps

The following input timestamps are all valid examples of literal time values that you can load into Amazon Redshift tables. All of the valid date literals can be combined with the following time literals.

<table>
<thead>
<tr>
<th>Input timestamps (concatenated dates and times)</th>
<th>Description (of time part)</th>
</tr>
</thead>
<tbody>
<tr>
<td>20080215 04:05:06.789</td>
<td>4:05 am and 6.789 seconds</td>
</tr>
<tr>
<td>20080215 04:05:06</td>
<td>4:05 am and 6 seconds</td>
</tr>
<tr>
<td>20080215 04:05</td>
<td>4:05 am exactly</td>
</tr>
<tr>
<td>20080215 04:0506</td>
<td>4:05 am and 6 seconds</td>
</tr>
<tr>
<td>20080215 04:05 AM</td>
<td>4:05 am exactly; AM is optional</td>
</tr>
<tr>
<td>20080215 04:05 PM</td>
<td>4:05 pm exactly; hour value must be &lt; 12.</td>
</tr>
<tr>
<td>20080215 16:05</td>
<td>4:05 05 pm exactly</td>
</tr>
</tbody>
</table>
**Input timestamps (concatenated dates and times)** | **Description (of time part)**
---|---
20080215 | Midnight (by default)

### Special datetime values

The following special values can be used as datetime literals and as arguments to date functions. They require single quotes and are converted to regular timestamp values during query processing.

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>now</td>
</tr>
<tr>
<td>today</td>
</tr>
<tr>
<td>tomorrow</td>
</tr>
<tr>
<td>yesterday</td>
</tr>
</tbody>
</table>

The following examples show how `now` and `today` work in conjunction with the `DATEADD` function:

```sql
select dateadd(day,1,'today');
```

```plaintext
date_add
---------------------
2009-11-17 00:00:00
(1 row)
```

```sql
select dateadd(day,1,'now');
```

```plaintext
date_add
---------------------
2009-11-17 10:45:32.021394
(1 row)
```

### Interval literals

Use an interval literal to identify specific periods of time, such as 12 hours or 6 weeks. You can use these interval literals in conditions and calculations that involve datetime expressions.

**Note**

You cannot use the `INTERVAL` data type for columns in Amazon Redshift tables.

An interval is expressed as a combination of the `INTERVAL` keyword with a numeric quantity and a supported datepart; for example: `INTERVAL '7 days'` or `INTERVAL '59 minutes'`. Several quantities and units can be connected to form a more precise interval; for example: `INTERVAL '7 days, 3 hours, 59 minutes'`. Abbreviations and plurals of each unit are also supported; for example: `5 s, 5 second, and 5 seconds` are equivalent intervals.

If you do not specify a datepart, the interval value represents seconds. You can specify the quantity value as a fraction (for example: `0.5 days`).
Examples

The following examples show a series of calculations with different interval values.

Add 1 second to the specified date:

```
select caldate + interval '1 second' as dateplus from date
where caldate='12-31-2008';
dateplus
---------------------
2008-12-31 00:00:01
(1 row)
```

Add 1 minute to the specified date:

```
select caldate + interval '1 minute' as dateplus from date
where caldate='12-31-2008';
dateplus
---------------------
2008-12-31 00:01:00
(1 row)
```

Add 3 hours and 35 minutes to the specified date:

```
select caldate + interval '3 hours, 35 minutes' as dateplus from date
where caldate='12-31-2008';
dateplus
---------------------
2008-12-31 03:35:00
(1 row)
```

Add 52 weeks to the specified date:

```
select caldate + interval '52 weeks' as dateplus from date
where caldate='12-31-2008';
dateplus
---------------------
2009-12-30 00:00:00
(1 row)
```

Add 1 week, 1 hour, 1 minute, and 1 second to the specified date:

```
select caldate + interval '1w, 1h, 1m, 1s' as dateplus from date
where caldate='12-31-2008';
dateplus
---------------------
2009-01-07 01:01:01
(1 row)
```

Add 12 hours (half a day) to the specified date:

```
select caldate + interval '0.5 days' as dateplus from date
where caldate='12-31-2008';
```
**Boolean type**

Use the BOOLEAN data type to store true and false values in a single-byte column. The following table describes the three possible states for a Boolean value and the literal values that result in that state. Regardless of the input string, a Boolean column stores and outputs “t” for true and “f” for false.

<table>
<thead>
<tr>
<th>State</th>
<th>Valid literal values</th>
<th>Storage</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>TRUE 't'</td>
<td>1 byte</td>
</tr>
<tr>
<td></td>
<td>'true' 'y'</td>
<td></td>
</tr>
<tr>
<td></td>
<td>'yes' '1'</td>
<td></td>
</tr>
<tr>
<td>False</td>
<td>FALSE 'f'</td>
<td>1 byte</td>
</tr>
<tr>
<td></td>
<td>'false' 'n'</td>
<td></td>
</tr>
<tr>
<td></td>
<td>'no' '0'</td>
<td></td>
</tr>
<tr>
<td>Unknown</td>
<td>NULL</td>
<td>1 byte</td>
</tr>
</tbody>
</table>

**Examples**

You could use a BOOLEAN column to store an “Active/Inactive” state for each customer in a CUSTOMER table:

```sql
create table customer(
custid int,
active_flag boolean default true
);
```

```sql
insert into customer values(100, default);
```

```sql
select * from customer;
custid | active_flag
--------+----------------
100 | t
```

If no default value (true or false) is specified in the CREATE TABLE statement, inserting a default value means inserting a null.

In this example, the query selects users from the USERS table who like sports but do not like theatre:

```sql
select firstname, lastname, likesports, liketheatre from users
where likesports is true and liketheatre is false
order by userid limit 10;
```

<table>
<thead>
<tr>
<th>firstname</th>
<th>lastname</th>
<th>likesports</th>
<th>liketheatre</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
This example selects users from the USERS table for whom is it unknown whether they like rock music:

```sql
select firstname, lastname, likerock
from users
where likerock is unknown
order by userid limit 10;
```

<table>
<thead>
<tr>
<th>firstname</th>
<th>lastname</th>
<th>likerock</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rafael</td>
<td>Taylor</td>
<td></td>
</tr>
<tr>
<td>Vladimir</td>
<td>Humphrey</td>
<td></td>
</tr>
<tr>
<td>Barry</td>
<td>Roy</td>
<td></td>
</tr>
<tr>
<td>Tamekah</td>
<td>Juarez</td>
<td></td>
</tr>
<tr>
<td>Mufutau</td>
<td>Watkins</td>
<td></td>
</tr>
<tr>
<td>Naida</td>
<td>Calderon</td>
<td></td>
</tr>
<tr>
<td>Anika</td>
<td>Huff</td>
<td></td>
</tr>
<tr>
<td>Bruce</td>
<td>Beck</td>
<td></td>
</tr>
<tr>
<td>Mallory</td>
<td>Farrell</td>
<td></td>
</tr>
<tr>
<td>Scarlett</td>
<td>Mayer</td>
<td></td>
</tr>
</tbody>
</table>

(10 rows)

**Type compatibility and conversion**

**Compatibility**

Data type matching and matching of literal values and constants to data types occurs during various database operations, including:

- DML operations on tables
- UNION, INTERSECT, and EXCEPT queries
- CASE expressions
- Evaluation of predicates, such as LIKE and IN
- Evaluation of SQL functions that do comparisons or extractions of data
- Comparisons with mathematical operators

The results of these operations depend on type conversion rules and data type compatibility. **Compatibility** implies that a one-to-one matching of a certain value and a certain data type is not always required. Because some data types are compatible, an implicit conversion, or coercion, is possible (see Implicit conversion types (p. 233)). When data types are incompatible, it is sometimes possible to convert a value from one data type to another by using an explicit conversion function.
General compatibility and conversion rules

Note the following compatibility and conversion rules:

• In general, data types that fall into the same type category (such as different numeric data types) are compatible and can be implicitly converted.

For example, a decimal value can be inserted into an integer column; the decimal is rounded to produce a whole number. Secondly, a numeric value, such as 2008, can be extracted from a date and inserted into an integer column.

• Numeric data types enforce overflow conditions that occur when you attempt to insert out-of-range values. For example, a decimal value with a precision of 5 does not fit into a decimal column that was defined with a precision of 4. An integer or the whole part of a decimal is never truncated; however, the fractional part of a decimal can be rounded up or down, as appropriate.

• Different types of character strings are compatible; VARCHAR column strings containing single-byte data and CHAR column strings are comparable and implicitly convertible. VARCHAR strings that contain multi-byte data are not comparable. Also, a character string can be converted to a date, timestamp, or numeric value if the string is an appropriate literal value; any leading or trailing spaces are ignored. Conversely, a date, timestamp, or numeric value can be converted to a fixed-length or variable-length character string.

  **Note**
  A character string that you want to cast to a numeric type must contain a character representation of a number. For example, you can cast the strings ‘1.0’ or ‘5.9’ to decimal values, but you cannot cast the string ‘ABC’ to any numeric type.

• Numeric values that are compared with character strings are converted to character strings. To enforce the opposite conversion (convert character strings to numerics), use an explicit function, such as CAST or CONVERT.

• To convert 64-bit DECIMAL or NUMERIC values to a higher precision, you must use an explicit conversion function such as the CAST and CONVERT functions (p. 556).

Implicit conversion types

There are two types of implicit conversions: implicit conversions in assignments, such as setting values in INSERT or UPDATE commands, and implicit conversions in expressions, such as performing comparisons in the WHERE clause. The tables below list the data types that can be converted implicitly in assignments and expressions. You can also use an explicit conversion function to perform these conversions.

The following table lists the data types that can be converted implicitly in assignments or expressions.

<table>
<thead>
<tr>
<th>From type</th>
<th>To type</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIGINT (INT8)</td>
<td>BOOLEAN</td>
</tr>
<tr>
<td></td>
<td>CHAR</td>
</tr>
<tr>
<td></td>
<td>DECIMAL (NUMERIC)</td>
</tr>
<tr>
<td></td>
<td>DOUBLE PRECISION (FLOAT8)</td>
</tr>
<tr>
<td></td>
<td>INTEGER (INT, INT4)</td>
</tr>
<tr>
<td></td>
<td>REAL (FLOAT4)</td>
</tr>
<tr>
<td></td>
<td>SMALLINT (INT2)</td>
</tr>
<tr>
<td></td>
<td>VARCHAR</td>
</tr>
<tr>
<td>From type</td>
<td>To type</td>
</tr>
<tr>
<td>------------------------------</td>
<td>--------------------------------</td>
</tr>
<tr>
<td>CHAR</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>DATE</td>
<td>CHAR</td>
</tr>
<tr>
<td></td>
<td>VARCHAR</td>
</tr>
<tr>
<td></td>
<td>TIMESTAMP</td>
</tr>
<tr>
<td>DECIMAL (NUMERIC)</td>
<td>BIGINT (INT8)</td>
</tr>
<tr>
<td></td>
<td>CHAR</td>
</tr>
<tr>
<td></td>
<td>DOUBLE PRECISION (FLOAT8)</td>
</tr>
<tr>
<td></td>
<td>INTEGER (INT, INT4)</td>
</tr>
<tr>
<td></td>
<td>REAL (FLOAT4)</td>
</tr>
<tr>
<td></td>
<td>SMALLINT (INT2)</td>
</tr>
<tr>
<td></td>
<td>VARCHAR</td>
</tr>
<tr>
<td>DOUBLE PRECISION (FLOAT8)</td>
<td>BIGINT (INT8)</td>
</tr>
<tr>
<td></td>
<td>CHAR</td>
</tr>
<tr>
<td></td>
<td>DECIMAL (NUMERIC)</td>
</tr>
<tr>
<td></td>
<td>INTEGER (INT, INT4)</td>
</tr>
<tr>
<td></td>
<td>REAL (FLOAT4)</td>
</tr>
<tr>
<td></td>
<td>SMALLINT (INT2)</td>
</tr>
<tr>
<td></td>
<td>VARCHAR</td>
</tr>
<tr>
<td>INTEGER (INT, INT4)</td>
<td>BIGINT (INT8)</td>
</tr>
<tr>
<td></td>
<td>BOOLEAN</td>
</tr>
<tr>
<td></td>
<td>CHAR</td>
</tr>
<tr>
<td></td>
<td>DECIMAL (NUMERIC)</td>
</tr>
<tr>
<td></td>
<td>DOUBLE PRECISION (FLOAT8)</td>
</tr>
<tr>
<td></td>
<td>REAL (FLOAT4)</td>
</tr>
<tr>
<td></td>
<td>SMALLINT (INT2)</td>
</tr>
<tr>
<td></td>
<td>VARCHAR</td>
</tr>
<tr>
<td>REAL (FLOAT4)</td>
<td>BIGINT (INT8)</td>
</tr>
<tr>
<td></td>
<td>CHAR</td>
</tr>
<tr>
<td></td>
<td>DECIMAL (NUMERIC)</td>
</tr>
<tr>
<td></td>
<td>INTEGER (INT, INT4)</td>
</tr>
<tr>
<td></td>
<td>SMALLINT (INT2)</td>
</tr>
<tr>
<td></td>
<td>VARCHAR</td>
</tr>
</tbody>
</table>
Collation sequences

Amazon Redshift does not support locale-specific or user-defined collation sequences. In general, the results of any predicate in any context could be affected by the lack of locale-specific rules for sorting and comparing data values. For example, ORDER BY expressions and functions such as MIN, MAX, and RANK return results based on binary UTF8 ordering of the data that does not take locale-specific characters into account.

Expressions

Topics
- Simple expressions (p. 235)
- Compound expressions (p. 236)
- Expression lists (p. 237)
- Scalar subqueries (p. 238)
- Function expressions (p. 238)

An expression is a combination of one or more values, operators, or functions that evaluate to a value. The data type of an expression is generally that of its components.

Simple expressions

A simple expression is one of the following:

- A constant or literal value
- A column name or column reference
- A scalar function
- An aggregate (set) function
- A window function
- A scalar subquery

<table>
<thead>
<tr>
<th>From type</th>
<th>To type</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMALLINT (INT2)</td>
<td>BIGINT (INT8)</td>
</tr>
<tr>
<td></td>
<td>BOOLEAN</td>
</tr>
<tr>
<td></td>
<td>CHAR</td>
</tr>
<tr>
<td></td>
<td>DECIMAL (NUMERIC)</td>
</tr>
<tr>
<td></td>
<td>DOUBLE PRECISION (FLOAT8)</td>
</tr>
<tr>
<td></td>
<td>INTEGER (INT, INT4)</td>
</tr>
<tr>
<td></td>
<td>REAL (FLOAT4)</td>
</tr>
<tr>
<td></td>
<td>VARCHAR</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>CHAR</td>
</tr>
<tr>
<td></td>
<td>DATE</td>
</tr>
<tr>
<td></td>
<td>VARCHAR</td>
</tr>
</tbody>
</table>
Examples of simple expressions include:

<table>
<thead>
<tr>
<th>5+12</th>
</tr>
</thead>
<tbody>
<tr>
<td>dateid</td>
</tr>
<tr>
<td>sales.qtysold * 100</td>
</tr>
<tr>
<td>sqrt (4)</td>
</tr>
<tr>
<td>max (qtysold)</td>
</tr>
<tr>
<td>(select max (qtysold) from sales)</td>
</tr>
</tbody>
</table>

**Compound expressions**

A compound expression is a series of simple expressions joined by arithmetic operators. A simple expression used in a compound expression must return a numeric value.

**Synopsis**

```
expression
operator
expression | (compound_expression)
```

**Arguments**

- `expression`: A simple expression that evaluates to a value.
- `operator`: A compound arithmetic expression can be constructed using the following operators, in this order of precedence:
  - `( )`: parentheses to control the order of evaluation
  - `+`, `-`: positive and negative sign/operator
  - `^`, `/`, `||`: exponentiation, square root, cube root
  - `*`, `/`, `%`: multiplication, division, and modulo operators
  - `@`: absolute value
  - `+`, `-`: addition and subtraction
  - `&`, `|`, `#`, `~`, `<<`, `>>`: AND, OR, XOR, NOT, shift left, shift right bitwise operators
  - `||`: concatenation

**Examples**

Examples of compound expressions include:

| ('SMITH' || 'JONES') |
|----------------------|
| sum(x) / y           |
| sqrt(256) * avg(column) |
| rank() over (order by qtysold) / 100 |
| (select (pricepaid - commission) from sales where dateid = 1882) * (qtysold) |

Some functions can also be nested within other functions. For example, any scalar function can nest within another scalar function. The following example returns the sum of the absolute values of a set of numbers:
Window functions cannot be used as arguments for aggregate functions or other window functions. The following expression would return an error:

```
avg(rank() over (order by qtysold))
```

Window functions can have a nested aggregate function. The following expression sums sets of values and then ranks them:

```
rank() over (order by sum(qtysold))
```

## Expression lists

An expression list is a combination of expressions, and can appear in membership and comparison conditions (WHERE clauses) and in GROUP BY clauses.

### Synopsis

```
expression , expression , ... | {expression, expression, ...}
```

### Arguments

**expression**

A simple expression that evaluates to a value. An expression list can contain one or more comma-separated expressions or one or more sets of comma-separated expressions. When there are multiple sets of expressions, each set must contain the same number of expressions, and be separated by parentheses. The number of expressions in each set must match the number of expressions before the operator in the condition.

### Examples

The following are examples of expression lists in conditions:

```
(1, 5, 10)
('THESE', 'ARE', 'STRINGS')
(('one', 'two', 'three'), ('blue', 'yellow', 'green'))
```

The number of expressions in each set must match the number in the first part of the statement:

```
select * from venue
where (venuecity, venuestate) in (('Miami', 'FL'), ('Tampa', 'FL'))
order by venueid;
```

<table>
<thead>
<tr>
<th>venueid</th>
<th>venuename</th>
<th>venuecity</th>
<th>venuestate</th>
<th>venueseats</th>
</tr>
</thead>
<tbody>
<tr>
<td>28</td>
<td>American Airlines Arena</td>
<td>Miami</td>
<td>FL</td>
<td>0</td>
</tr>
<tr>
<td>54</td>
<td>St. Pete Times Forum</td>
<td>Tampa</td>
<td>FL</td>
<td>0</td>
</tr>
<tr>
<td>91</td>
<td>Raymond James Stadium</td>
<td>Tampa</td>
<td>FL</td>
<td>65647</td>
</tr>
</tbody>
</table>

(3 rows)
Scalar subqueries
A scalar subquery is a regular SELECT query in parentheses that returns exactly one value: one row with one column. The query is executed and the returned value is used in the outer query. If the subquery returns zero rows, the value of the subquery expression is null. If it returns more than one row, Amazon Redshift returns an error. The subquery can refer to variables from the parent query, which will act as constants during any one invocation of the subquery.

You can use scalar subqueries in most statements that call for an expression. Scalar subqueries are not valid expressions in the following cases:

- As default values for expressions
- In WHEN conditions of CASE expressions
- In GROUP BY and HAVING clauses

Example
The following subquery computes the average price paid per sale across the entire year of 2008, then the outer query uses that value in the output to compare against the average price per sale per quarter:

```
select qtr, avg(pricepaid) as avg_saleprice_per_qtr,
     (select avg(pricepaid)
      from sales join date on sales.dateid=date.dateid
      where year = 2008) as avg_saleprice_yearly
from sales join date on sales.dateid=date.dateid
where year = 2008
group by qtr
order by qtr;
```

<table>
<thead>
<tr>
<th>qtr</th>
<th>avg_saleprice_per_qtr</th>
<th>avg_saleprice_yearly</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>647.64</td>
<td>642.28</td>
</tr>
<tr>
<td>2</td>
<td>646.86</td>
<td>642.28</td>
</tr>
<tr>
<td>3</td>
<td>636.79</td>
<td>642.28</td>
</tr>
<tr>
<td>4</td>
<td>638.26</td>
<td>642.28</td>
</tr>
</tbody>
</table>

Function expressions
Syntax
Any built-in can be used as an expression. The syntax for a function call is the name of a function followed by its argument list in parentheses.

```
function ( [expression [, expression...]] )
```

Arguments

- `function`  
  Any built-in function.
- `expression`  
  Any expression(s) matching the data type and parameter count expected by the function.
Examples

<table>
<thead>
<tr>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs (variable)</td>
</tr>
<tr>
<td>select avg (qtysold + 3) from sales;</td>
</tr>
<tr>
<td>select dateadd (day,30,caldate) as plus30days from date;</td>
</tr>
</tbody>
</table>

Conditions

Topics

- Synopsis (p. 239)
- Comparison condition (p. 239)
- Logical condition (p. 241)
- Pattern-matching conditions (p. 244)
- Range condition (p. 252)
- Null condition (p. 253)
- EXISTS condition (p. 254)
- IN condition (p. 255)

A condition is a statement of one or more expressions and logical operators that evaluates to true, false, or unknown. Conditions are also sometimes referred to as predicates.

Note

All string comparisons and LIKE pattern matches are case-sensitive. For example, ‘A’ and ‘a’ do not match. However, you can do a case-insensitive pattern match by using the ILIKE predicate.

Synopsis

```
<table>
<thead>
<tr>
<th>comparison_condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>logical_condition</td>
</tr>
<tr>
<td>range_condition</td>
</tr>
<tr>
<td>pattern_matching_condition</td>
</tr>
<tr>
<td>null_condition</td>
</tr>
<tr>
<td>EXISTS_condition</td>
</tr>
<tr>
<td>IN_condition</td>
</tr>
</tbody>
</table>
```

Comparison condition

Comparison conditions state logical relationships between two values. All comparison conditions are binary operators with a Boolean return type. Amazon Redshift supports the comparison operators described in the following table:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;</td>
<td>a &lt; b</td>
<td>Value a is less than value b.</td>
</tr>
<tr>
<td>&gt;</td>
<td>a &gt; b</td>
<td>Value a is greater than value b.</td>
</tr>
<tr>
<td>&lt;=</td>
<td>a &lt;= b</td>
<td>Value a is less than or equal to value b.</td>
</tr>
<tr>
<td>&gt;=</td>
<td>a &gt;= b</td>
<td>Value a is greater than or equal to value b.</td>
</tr>
<tr>
<td>=</td>
<td>a = b</td>
<td>Value a is equal to value b.</td>
</tr>
</tbody>
</table>

API Version 2012-12-01

Amazon Redshift Database Developer Guide

Conditions
### Conditions

<table>
<thead>
<tr>
<th>Operator</th>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;&gt; !</td>
<td>a &lt;&gt; b or a != b</td>
<td>Value a is not equal to value b.</td>
</tr>
<tr>
<td>ANY</td>
<td>a = ANY(subquery)</td>
<td>Value a is equal to any value returned by the subquery.</td>
</tr>
<tr>
<td>ALL</td>
<td>a &lt;&gt; ALL or != ALL(subquery))</td>
<td>Value a is not equal to any value returned by the subquery.</td>
</tr>
<tr>
<td>IS TRUE</td>
<td>a IS TRUE</td>
<td>Value a is Boolean TRUE.</td>
</tr>
<tr>
<td>FALSE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UNKNOWN</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Usage notes

**= ANY | SOME**

The ANY and SOME keywords are synonymous with the IN condition, and return true if the comparison is true for at least one value returned by a subquery that returns one or more values. Amazon Redshift supports only the = (equals) condition for ANY and SOME. Inequality conditions are not supported.

**Note**

The ALL predicate is not supported.

**<> ALL**

The ALL keyword is synonymous with NOT IN (see IN condition (p. 255) condition) and returns true if the expression is not included in the results of the subquery. Amazon Redshift supports only the <> or != (not equals) condition for ALL. Other comparison conditions are not supported.

**IS TRUE/FALSE/UNKNOWN**

Non-zero values equate to TRUE, 0 equates to FALSE, and null equates to UNKNOWN. See the Boolean type (p. 231) data type.

#### Examples

Here are some simple examples of comparison conditions:

```sql
a = 5
a < b
min(x) >= 5
qtysold = any (select qtysold from sales where dateid = 1882
```

The following query returns venues with more than 10000 seats from the VENUE table:

```sql
select venueid, venuename, venueseats from venue
where venueseats > 10000
order by venueseats desc;
```

<table>
<thead>
<tr>
<th>venueid</th>
<th>venuename</th>
<th>venueseats</th>
</tr>
</thead>
<tbody>
<tr>
<td>83</td>
<td>FedExField</td>
<td>91704</td>
</tr>
<tr>
<td>6</td>
<td>New York Giants Stadium</td>
<td>80242</td>
</tr>
<tr>
<td>79</td>
<td>Arrowhead Stadium</td>
<td>79451</td>
</tr>
<tr>
<td>78</td>
<td>INVESCO Field</td>
<td>76125</td>
</tr>
<tr>
<td>69</td>
<td>Dolphin Stadium</td>
<td>74916</td>
</tr>
<tr>
<td>67</td>
<td>Ralph Wilson Stadium</td>
<td>73967</td>
</tr>
<tr>
<td>76</td>
<td>Jacksonville Municipal Stadium</td>
<td>73800</td>
</tr>
</tbody>
</table>
Logical condition

Logical conditions combine the result of two conditions to produce a single result. All logical conditions are binary operators with a Boolean return type.

Synopsis

expression
{ AND | OR }
expression
NOT expression
Logical conditions use a three-valued Boolean logic where the null value represents an unknown relationship. The following table describes the results for logical conditions, where \( E_1 \) and \( E_2 \) represent expressions:

<table>
<thead>
<tr>
<th>( E_1 )</th>
<th>( E_2 )</th>
<th>( E_1 \ \text{AND} \ E_2 )</th>
<th>( E_1 \ \text{OR} \ E_2 )</th>
<th>NOT ( E_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>TRUE</td>
<td>TRUE</td>
<td>TRUE</td>
<td>FALSE</td>
</tr>
<tr>
<td>TRUE</td>
<td>FALSE</td>
<td>FALSE</td>
<td>TRUE</td>
<td>TRUE</td>
</tr>
<tr>
<td>TRUE</td>
<td>UNKNOWN</td>
<td>UNKNOWN</td>
<td>TRUE</td>
<td>UNKNOWN</td>
</tr>
<tr>
<td>FALSE</td>
<td>TRUE</td>
<td>FALSE</td>
<td>TRUE</td>
<td>TRUE</td>
</tr>
<tr>
<td>FALSE</td>
<td>FALSE</td>
<td>FALSE</td>
<td>FALSE</td>
<td>TRUE</td>
</tr>
<tr>
<td>FALSE</td>
<td>UNKNOWN</td>
<td>FALSE</td>
<td>UNKNOWN</td>
<td>UNKNOWN</td>
</tr>
<tr>
<td>UNKNOWN</td>
<td>TRUE</td>
<td>UNKNOWN</td>
<td>TRUE</td>
<td>TRUE</td>
</tr>
<tr>
<td>UNKNOWN</td>
<td>FALSE</td>
<td>FALSE</td>
<td>UNKNOWN</td>
<td>UNKNOWN</td>
</tr>
<tr>
<td>UNKNOWN</td>
<td>UNKNOWN</td>
<td>UNKNOWN</td>
<td>UNKNOWN</td>
<td>UNKNOWN</td>
</tr>
</tbody>
</table>

The NOT operator is evaluated before AND, and the AND operator is evaluated before the OR operator. Any parentheses used may override this default order of evaluation.

**Examples**

The following example returns USERID and USERNAME from the USERS table where the user likes both Las Vegas and sports:

```sql
select userid, username from users
where likevegas = 1 and likesports = 1
order by userid;
```

<table>
<thead>
<tr>
<th>userid</th>
<th>username</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>JSG99FHE</td>
</tr>
<tr>
<td>67</td>
<td>TWU10MZT</td>
</tr>
<tr>
<td>87</td>
<td>DUF19VXU</td>
</tr>
<tr>
<td>92</td>
<td>HYP36WEQ</td>
</tr>
<tr>
<td>109</td>
<td>FPL38HZK</td>
</tr>
<tr>
<td>120</td>
<td>DMJ24GUZ</td>
</tr>
<tr>
<td>123</td>
<td>QZR22XGQ</td>
</tr>
<tr>
<td>130</td>
<td>2QC82ALK</td>
</tr>
<tr>
<td>133</td>
<td>LBN45WCH</td>
</tr>
<tr>
<td>144</td>
<td>UCX04JKN</td>
</tr>
<tr>
<td>165</td>
<td>TEY680EB</td>
</tr>
<tr>
<td>169</td>
<td>AYQ83HGO</td>
</tr>
<tr>
<td>184</td>
<td>TVX65AZX</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(2128 rows)</td>
</tr>
</tbody>
</table>

The next example returns the USERID and USERNAME from the USERS table where the user likes Las Vegas, or sports, or both. This query returns all of the output from the previous example plus the users who like only Las Vegas or sports.
The following query uses parentheses around the OR condition to find venues in New York or California where Macbeth was performed:

```
select distinct venuename, venuecity
from venue join event on venue.venueid=event.venueid
where (venuestate = 'NY' or venuestate = 'CA') and eventname='Macbeth'
order by 2,1;
```

```
venuename                |   venuecity
----------------------------------------+---------------
Geffen Playhouse                       | Los Angeles
Greek Theatre                          | Los Angeles
Royce Hall                             | Los Angeles
American Airlines Theatre              | New York City
August Wilson Theatre                  | New York City
Belasco Theatre                        | New York City
Bernard B. Jacobs Theatre              | New York City
...                                      
```

Removing the parentheses in this example changes the logic and results of the query.

The following example uses the NOT operator:

```
select * from category
where not catid=1
order by 1;
```

```
catid | catgroup |  catname  |                  catdesc
-------+----------+-----------+--------------------------------------------
2 | Sports   | NHL       | National Hockey League
3 | Sports   | NFL       | National Football League
4 | Sports   | NBA       | National Basketball Association
5 | Sports   | MLS       | Major League Soccer
...                                      
```
The following example uses a NOT condition followed by an AND condition:

```
select * from category
where (not catid=1) and catgroup='Sports'
order by catid;
```

<table>
<thead>
<tr>
<th>catid</th>
<th>catgroup</th>
<th>catname</th>
<th>catdesc</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Sports</td>
<td>NHL</td>
<td>National Hockey League</td>
</tr>
<tr>
<td>3</td>
<td>Sports</td>
<td>NFL</td>
<td>National Football League</td>
</tr>
<tr>
<td>4</td>
<td>Sports</td>
<td>NBA</td>
<td>National Basketball Association</td>
</tr>
<tr>
<td>5</td>
<td>Sports</td>
<td>MLS</td>
<td>Major League Soccer</td>
</tr>
</tbody>
</table>

(4 rows)

Pattern-matching conditions

Topics

- LIKE (p. 245)
- SIMILAR TO (p. 247)
- POSIX operators (p. 249)

A pattern-matching operator searches a string for a pattern specified in the conditional expression and returns true or false depend on whether it finds a match. Amazon Redshift uses three methods for pattern matching:

- LIKE expressions

  The LIKE operator compares a string expression, such as a column name, with a pattern that uses the wildcard characters % (percent) and _ (underscore). LIKE pattern matching always covers the entire string. LIKE performs a case-sensitive match and ILIKE performs a case-insensitive match.

- SIMILAR TO regular expressions

  The SIMILAR TO operator matches a string expression with a SQL standard regular expression pattern, which can include a set of pattern-matching metacharacters that includes the two supported by the LIKE operator. SIMILAR TO matches the entire string and performs a case-sensitive match.

- POSIX-style regular expressions

  POSIX regular expressions provide a more powerful means for pattern matching than the LIKE and SIMILAR TO operators. POSIX regular expression patterns can match any portion of the string and performs a case-sensitive match.

Regular expression matching, using SIMILAR TO or POSIX operators, is computationally expensive. We recommend using LIKE whenever possible, especially when processing a very large number of rows. For example, the following queries are functionally identical, but the query that uses LIKE executes several times faster than the query that uses a regular expression:

```
select count(*) from event where eventname SIMILAR TO '%(Ring|Die)%';
selct count(*) from event where eventname LIKE '%Ring%' OR eventname LIKE '%Die%';
```
LIKE

The LIKE operator compares a string expression, such as a column name, with a pattern that uses the wildcard characters % (percent) and _ (underscore). LIKE pattern matching always covers the entire string. To match a sequence anywhere within a string, the pattern must start and end with a percent sign.

LIKE is case-sensitive; ILIKE is case-insensitive.

Synopsis

```
expression [ NOT ] LIKE | ILIKE pattern [ ESCAPE 'escape_char' ]
```

Arguments

expression
A valid UTF-8 character expression, such as a column name.

LIKE | ILIKE
LIKE performs a case-sensitive pattern match. ILIKE performs a case-insensitive pattern match for single-byte characters. Both LIKE and ILIKE perform a case-insensitive pattern match for multi-byte characters.

pattern
A valid UTF-8 character expression with the pattern to be matched.

escape_char
A character expression that will escape metacharacters characters in the pattern. The default is two backslashes ('\').

If pattern does not contain metacharacters, then the pattern only represents the string itself; in that case LIKE acts the same as the equals operator.

Either of the character expressions can be CHAR or VARCHAR data types. If they differ, Amazon Redshift converts pattern to the data type of expression.

LIKE supports the following pattern-matching metacharacters:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>%</td>
<td>Matches any sequence of zero or more characters.</td>
</tr>
<tr>
<td>_</td>
<td>Matches any single character.</td>
</tr>
</tbody>
</table>

Examples

The following table shows examples of pattern matching using LIKE:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>'abc' LIKE 'abc'</td>
<td>True</td>
</tr>
<tr>
<td>'abc' LIKE 'a%'</td>
<td>True</td>
</tr>
<tr>
<td>'abc' LIKE '<em>B</em>'</td>
<td>False</td>
</tr>
<tr>
<td>'abc' ILIKE '<em>B</em>'</td>
<td>True</td>
</tr>
<tr>
<td>'abc' LIKE 'c%'</td>
<td>False</td>
</tr>
</tbody>
</table>
The following example finds all cities whose names start with "E":

```sql
select distinct city from users
where city like 'E%' order by city;

city
---------------
East Hartford
East Lansing
East Rutherford
East St. Louis
Easthampton
Easton
Eatontown
Eau Claire
...
```

The following example finds users whose last name contains "ten":

```sql
select distinct lastname from users
where lastname like '%ten%' order by lastname;

lastname
-------------
Christensen
Wooten
...
```

The following example finds cities whose third and fourth characters are "ea". The command uses ILIKE to demonstrate case insensitivity:

```sql
select distinct city from users where city ilike '__EA%' order by city;

city
-------------
Brea
Clearwater
Great Falls
Ocean City
Olean
Wheaton
(6 rows)
```

The following example uses the default escape string (\) to search for strings that include " _":

```sql
select tablename, "column" from pg_table_def
where "column" like '%start\_%'
limit 5;

tablename     |    column
-------------------+---------------
stl_s3client      | start_time
stl_tr_conflict   | xact_start_ts
stl_undone        | undo_start_ts
stl_unload_log    | start_time
stl_vacuum_detail | start_row
(5 rows)
```
The following example specifies `'\'` as the escape character, then uses the escape character to search for strings that include "_":

```sql
select tablename, "column" from pg_table_def
where "column" like '%start^_%' escape '^'
limit 5;
```

| tablename     | column           |
|---------------+-----------------|
| stl_s3client  | start_time      |
| stl_tr_conflict| xact_start_ts    |
| stl_undone    | undo_start_ts   |
| stl_unload_log| start_time      |
| stl_vacuum_detail | start_row       |

(5 rows)

**SIMILAR TO**

The SIMILAR TO operator matches a string expression, such as a column name, with a SQL standard regular expression pattern. A SQL regular expression pattern can include a set of pattern-matching metacharacters, including the two supported by the LIKE (p. 245) operator.

The SIMILAR TO operator returns true only if its pattern matches the entire string, unlike POSIX regular expression behavior, where the pattern can match any portion of the string.

SIMILAR TO performs a case-sensitive match.

**Note**

Regular expression matching using SIMILAR TO is computationally expensive. We recommend using LIKE whenever possible, especially when processing a very large number of rows. For example, the following queries are functionally identical, but the query that uses LIKE executes several times faster than the query that uses a regular expression:

```sql
select count(*) from event where eventname SIMILAR TO '%(Ring|Die)%';
select count(*) from event where eventname LIKE '%Ring%' OR eventname LIKE '%Die%';
```

**Synopsis**

```
expression [ NOT ] SIMILAR TO pattern [ ESCAPE 'escape_char' ]
```

**Arguments**

- **expression**
  - A valid UTF-8 character expression, such as a column name.
- **SIMILAR TO**
  - SIMILAR TO performs a case-sensitive pattern match for the entire string in expression.
- **pattern**
  - A valid UTF-8 character expression representing a SQL standard regular expression pattern.
- **escape_char**
  - A character expression that will escape metacharacters in the pattern. The default is two backslashes (``\``).

If pattern does not contain metacharacters, then the pattern only represents the string itself.
Either of the character expressions can be CHAR or VARCHAR data types. If they differ, Amazon Redshift converts *pattern* to the data type of *expression*.

SIMILAR TO supports the following pattern-matching metacharacters:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>%</td>
<td>Matches any sequence of zero or more characters.</td>
</tr>
<tr>
<td>_</td>
<td>Matches any single character.</td>
</tr>
<tr>
<td></td>
<td>Denotes alternation (either of two alternatives).</td>
</tr>
<tr>
<td>*</td>
<td>Repeat the previous item zero or more times.</td>
</tr>
<tr>
<td>+</td>
<td>Repeat the previous item one or more times.</td>
</tr>
<tr>
<td>?</td>
<td>Repeat the previous item zero or one time.</td>
</tr>
<tr>
<td>{m}</td>
<td>Repeat the previous item exactly <em>m</em> times.</td>
</tr>
<tr>
<td>{m,}</td>
<td>Repeat the previous item <em>m</em> or more times.</td>
</tr>
<tr>
<td>{m, n}</td>
<td>Repeat the previous item at least <em>m</em> and not more than <em>n</em> times.</td>
</tr>
<tr>
<td>(</td>
<td>Parentheses group items into a single logical item.</td>
</tr>
</tbody>
</table>

A bracket expression specifies a character class, just as in POSIX regular expressions.

**Examples**

The following table shows examples of pattern matching using SIMILAR TO:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>'abc' SIMILAR TO 'abc'</td>
<td>True</td>
</tr>
<tr>
<td>'abc' SIMILAR TO '<em>b</em>'</td>
<td>True</td>
</tr>
<tr>
<td>'abc' SIMILAR TO '<em>A</em>'</td>
<td>False</td>
</tr>
<tr>
<td>'abc' SIMILAR TO '%(b</td>
<td>d)%'</td>
</tr>
<tr>
<td>'abc' SIMILAR TO '(b</td>
<td>c)%'</td>
</tr>
<tr>
<td>'AbcAbcAbcdefgefgefg12efgef12' SIMILAR TO '((Ab)?c)+d((efg)+(12))+''</td>
<td>True</td>
</tr>
<tr>
<td>'aaaaab11111xy' SIMILAR TO 'a[6]_[0-9][5](x</td>
<td>y)[2]'</td>
</tr>
<tr>
<td>'$0.87' SIMILAR TO '$[0-9]+([0-9][0-9])?'</td>
<td>True</td>
</tr>
</tbody>
</table>

The following example finds all cities whose names contain "E" or "H":

```sql
select distinct city from users
where city similar to '%E%|%H%' order by city;
```

```sql
city
```
The following example uses the default escape string (\) to search for strings that include "\_":

```sql
select tablename, "column" from pg_table_def
where "column" similar to '%start\_\_%'
limit 5;
```

| tablename     |    column     |
|---------------+--------------|
| stl_s3client  | start_time   |
| stl_tr_conflict | xact_start_ts |
| stl undone    | undo_start_ts |
| stl unload log| start_time   |
| stl vacuum detail | start_row   |
(5 rows)

The following example specifies '^' as the escape string, then uses the escape string to search for strings that include "\_":

```sql
select tablename, "column" from pg_table_def
where "column" similar to '%start^\_\_' escape '^'
limit 5;
```

| tablename     |    column     |
|---------------+--------------|
| stl_s3client  | start_time   |
| stl_tr_conflict | xact_start_ts |
| stl undone    | undo_start_ts |
| stl unload log| start_time   |
| stl vacuum detail | start_row   |
(5 rows)

**POSIX operators**

POSIX regular expressions provide a more powerful means for pattern matching than the LIKE (p. 245) and SIMILAR TO (p. 247) operators. POSIX regular expression patterns can match any portion of a string, unlike the SIMILAR TO operator, which returns true only if its pattern matches the entire string.

**Note**

Regular expression matching using POSIX operators is computationally expensive. We recommend using LIKE whenever possible, especially when processing a very large number of rows. For example, the following queries are functionally identical, but the query that uses LIKE executes several times faster than the query that uses a regular expression:

```sql
select count(*) from event where eventname ~ '.*(Ring|Die).* ';  
select count(*) from event where eventname LIKE '%Ring%' OR eventname LIKE '%Die%';
```
Synopsis

expression [ ! ] ~ pattern

Arguments

expression

A valid UTF-8 character expression, such as a column name.

!  

Negation operator.

~  

Perform a case-sensitive match for any substring of expression.

pattern

A valid UTF-8 character expression representing a SQL standard regular expression pattern.

If pattern does not contain wildcard characters, then the pattern only represents the string itself.

To search for strings that include metacharacters, such as ‘. * | ? ’, and so on, escape the character using two backslashes (‘\’). Unlike SIMILAR TO and LIKE, POSIX regular expression syntax does not support a user-defined escape character.

Either of the character expressions can be CHAR or VARCHAR data types. If they differ, Amazon Redshift converts pattern to the data type of expression.

All of the character expressions can be CHAR or VARCHAR data types. If the expressions differ in data type, Amazon Redshift converts them to the data type of expression.

POSIX pattern matching supports the following metacharacters:

<table>
<thead>
<tr>
<th>POSIX</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td>Matches any single character.</td>
</tr>
<tr>
<td>*</td>
<td>Matches zero or more occurrences.</td>
</tr>
<tr>
<td>+</td>
<td>Matches one or more occurrences.</td>
</tr>
<tr>
<td>?</td>
<td>Matches zero or one occurrence.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>^</td>
<td>Matches the beginning-of-line character.</td>
</tr>
<tr>
<td>$</td>
<td>Matches the end-of-line character.</td>
</tr>
<tr>
<td>$</td>
<td>Matches the end of the string.</td>
</tr>
<tr>
<td>[ ]</td>
<td>Brackets specify a matching list, that should match one expression in the list. A caret (^) precedes a nonmatching list, which matches any character except for the expressions represented in the list.</td>
</tr>
<tr>
<td>( )</td>
<td>Parentheses group items into a single logical item.</td>
</tr>
<tr>
<td>{m}</td>
<td>Repeat the previous item exactly m times.</td>
</tr>
<tr>
<td>{m,}</td>
<td>Repeat the previous item m or more times.</td>
</tr>
<tr>
<td>{m,n}</td>
<td>Repeat the previous item at least m and not more than n times.</td>
</tr>
</tbody>
</table>
POSIX  | Description
--- | ---
[: :]  | Matches any character within a POSIX character class. In the following character classes, Amazon Redshift supports only ASCII characters: [:alnum:], [:alpha:], [:lower:], [:upper:]

Amazon Redshift supports the following POSIX character classes.

| Character Class | Description
| --- | ---
| [:alnum:] | All ASCII alphanumeric characters
| [:alpha:] | All ASCII alphabetic characters
| [:blank:] | All blank space characters.
| [:cntrl:] | All control characters (nonprinting)
| [:digit:] | All numeric digits
| [:lower:] | All lowercase ASCII alphabetic characters
| [:punct:] | All punctuation characters
| [:space:] | All space characters (nonprinting)
| [:upper:] | All uppercase ASCII alphabetic characters
| [:xdigit:] | All valid hexadecimal characters

Amazon Redshift supports the following Perl-influenced operators in regular expressions.

| Operator | Description
| --- | ---
| \d | A digit character
| \D | A nondigit character
| \w | A word character
| \W | A nonword character
| \s | A white space character
| \S | A non–white space character

**Examples**

The following table shows examples of pattern matching using POSIX operators:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>'abc' ~ 'abc'</td>
<td>True</td>
</tr>
<tr>
<td>'abc' ~ 'a'</td>
<td>True</td>
</tr>
<tr>
<td>'abc' ~ 'A'</td>
<td>False</td>
</tr>
</tbody>
</table>
The following example finds all cities whose names contain E or H:

```sql
select distinct city from users
where city ~ '.*E.*|.*H.*'
order by city;

city
-----------------------
Agoura Hills
Auburn Hills
Benton Harbor
Beverly Hills
Chicago Heights
Chino Hills
Citrus Heights
East Hartford
```

The following example uses the escape string (\) to search for strings that include a period.

```sql
select venuename from venue
where venuename ~ '.*\..*';

venuename
-----------------------------
Bernard B. Jacobs Theatre
E.J. Nutter Center
Hubert H. Humphrey Metrodome
Jobing.com Arena
St. James Theatre
St. Pete Times Forum
Superpages.com Center
U.S. Cellular Field
```

## Range condition

Range conditions test expressions for inclusion in a range of values, using the keywords BETWEEN and AND.

### Synopsis

```
expression [ NOT ] BETWEEN expression AND expression
```

Expressions can be numeric, character, or datetime data types, but they must be compatible.
Examples

The first example counts how many transactions registered sales of either 2, 3, or 4 tickets:

```sql
select count(*) from sales
where qtysold between 2 and 4;

<table>
<thead>
<tr>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>104021</td>
</tr>
</tbody>
</table>
```

The first expression in a range condition must be the lesser value and the second expression the greater value. The following example will always return zero rows due to the values of the expressions:

```sql
select count(*) from sales
where qtysold between 4 and 2;

<table>
<thead>
<tr>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>
```

However, applying the NOT modifier will invert the logic and produce a count of all rows:

```sql
select count(*) from sales
where qtysold not between 4 and 2;

<table>
<thead>
<tr>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>172456</td>
</tr>
</tbody>
</table>
```

The following query returns a list of venues with 20000 to 50000 seats:

```sql
select venueid, venuename, venueseats from venue
where venueseats between 20000 and 50000
order by venueseats desc;

<table>
<thead>
<tr>
<th>venueid</th>
<th>venuename</th>
<th>venueseats</th>
</tr>
</thead>
<tbody>
<tr>
<td>116</td>
<td>Busch Stadium</td>
<td>49660</td>
</tr>
<tr>
<td>106</td>
<td>Rangers BallPark in Arlington</td>
<td>49115</td>
</tr>
<tr>
<td>96</td>
<td>Oriole Park at Camden Yards</td>
<td>48876</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Null condition

The null condition tests for nulls, when a value is missing or unknown.
Synopsis

expression IS [ NOT ] NULL

Arguments

expression

Any expression such as a column.

IS NULL

Is true when the expression's value is null and false when it has a value.

IS NOT NULL

Is false when the expression's value is null and true when it has a value.

Example

This example indicates how many times the SALES table contains null in the QTYSOLD field:

```sql
select count(*) from sales
where qtysold is null;
```

```
count
-------
0
(1 row)
```

EXISTS condition

EXISTS conditions test for the existence of rows in a subquery, and return true if a subquery returns at least one row. If NOT is specified, the condition returns true if a subquery returns no rows.

Synopsis

[ NOT ] EXISTS (table_subquery)

Arguments

EXISTS

Is true when the table_subquery returns at least one row.

NOT EXISTS

Is true when the table_subquery returns no rows.

table_subquery

A subquery that evaluates to a table with one or more columns and one or more rows.

Example

This example returns all date identifiers, one time each, for each date that had a sale of any kind:

```sql
select dateid from date
where exists ( select 1 from sales
where date.dateid = sales.dateid )
```
order by dateid;

dateid
--------
1827
1828
1829
...

**IN condition**

An IN condition tests a value for membership in a set of values or in a subquery.

**Synopsis**

`expression [ NOT ] IN (expr_list | table_subquery)`

**Arguments**

- `expression`: A numeric, character, or datetime expression that is evaluated against the `expr_list` or `table_subquery` and must be compatible with the data type of that list or subquery.

- `expr_list`: One or more comma-delimited expressions, or one or more sets of comma-delimited expressions bounded by parentheses.

- `table_subquery`: A subquery that evaluates to a table with one or more rows, but is limited to only one column in its select list.

`IN | NOT IN`

IN returns true if the expression is a member of the expression list or query. NOT IN returns true if the expression is not a member. IN and NOT IN return NULL and no rows are returned in the following cases: If `expression` yields null; or if there are no matching `expr_list` or `table_subquery` values and at least one of these comparison rows yields null.

**Examples**

The following conditions are true only for those values listed:

```sql
qtysold in (2, 4, 5)
date.day in ('Mon', 'Tues')
date.month not in ('Oct', 'Nov', 'Dec')
```

**Optimization for large IN lists**

To optimize query performance, an IN list that includes more than 10 values is internally evaluated as a scalar array. IN lists with fewer than 10 values are evaluated as a series of OR predicates. This optimization is supported for all data types except DECIMAL.

Look at the EXPLAIN output for the query to see the effect of this optimization. For example:

```sql
explain select * from sales
where salesid in (1,2,3,4,5,6,7,8,9,10,11);
```
SQL Commands

Topics

- ABORT (p. 257)
- ALTER DATABASE (p. 258)
- ALTER GROUP (p. 259)
- ALTER SCHEMA (p. 260)
- ALTER TABLE (p. 261)
- ALTER USER (p. 266)
- ANALYZE (p. 268)
- ANALYZE COMPRESSION (p. 269)
- BEGIN (p. 270)
- CANCEL (p. 272)
- CLOSE (p. 273)
- COMMENT (p. 274)
- COMMIT (p. 275)
- COPY (p. 276)
- CREATE DATABASE (p. 308)
- CREATE GROUP (p. 308)
- CREATE SCHEMA (p. 309)
- CREATE TABLE (p. 310)
- CREATE TABLE AS (p. 320)
- CREATE USER (p. 325)
- CREATE VIEW (p. 327)
- DEALLOCATE (p. 327)
- DECLARE (p. 328)
- DELETE (p. 331)
- DROP DATABASE (p. 332)
- DROP GROUP (p. 333)
- DROP SCHEMA (p. 334)
- DROP TABLE (p. 334)
- DROP USER (p. 336)
- DROP VIEW (p. 337)
- END (p. 339)
- EXECUTE (p. 339)
- EXPLAIN (p. 340)
- FETCH (p. 344)
- GRANT (p. 346)
- INSERT (p. 349)
The SQL language consists of commands that you use to create and manipulate database objects, run queries, load tables, and modify the data in tables.

**Note**

Amazon Redshift is based on PostgreSQL 8.0.2. Amazon Redshift and PostgreSQL have a number of very important differences that you must be aware of as you design and develop your data warehouse applications. For more information about how Amazon Redshift SQL differs from PostgreSQL, see Amazon Redshift and PostgreSQL (p. 206).

**Note**

The maximum size for a single SQL statement is 16 MB.

### ABORT

Aborts the currently running transaction and discards all updates made by that transaction. ABORT has no effect on already completed transactions.

This command performs the same function as the ROLLBACK command. See ROLLBACK (p. 358) for more detailed documentation.

#### Synopsis

```
ABORT [ WORK | TRANSACTION ]
```

#### Parameters

**WORK**

Optional keyword.

**TRANSACTION**

Optional keyword; WORK and TRANSACTION are synonyms.
Example

The following example creates a table then starts a transaction where data is inserted into the table. The ABORT command then rolls back the data insertion to leave the table empty.

The following command creates an example table called MOVIE_GROSS:

```sql
create table movie_gross( name varchar(30), gross bigint );
```

The next set of commands starts a transaction that inserts two data rows into the table:

```sql
begin;
insert into movie_gross values ( 'Raiders of the Lost Ark', 23400000);
insert into movie_gross values ( 'Star Wars', 10000000 );
```

Next, the following command selects the data from the table to show that it was successfully inserted:

```sql
select * from movie_gross;
```

The command output shows that both rows are successfully inserted:

<table>
<thead>
<tr>
<th>name</th>
<th>gross</th>
</tr>
</thead>
<tbody>
<tr>
<td>Raiders of the Lost Ark</td>
<td>23400000</td>
</tr>
<tr>
<td>Star Wars</td>
<td>10000000</td>
</tr>
</tbody>
</table>

This command now rolls back the data changes to where the transaction began:

```sql
abort;
```

Selecting data from the table now shows an empty table:

```sql
select * from movie_gross;
```

<table>
<thead>
<tr>
<th>name</th>
<th>gross</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(0 rows)

ALTER DATABASE

Changes the attributes of a database.

Synopsis

```
ALTER DATABASE database_name
{
RENAME TO new_name |
```
Parameters

database_name
   Name of the database to alter. Typically, you alter a database that you are not currently connected to; in any case, the changes take effect only in subsequent sessions. You can change the owner of the current database, but you cannot rename it:

   alter database ticket rename to newticket;
   ERROR: current database may not be renamed

RENAME TO
   Renames the specified database. For more information about valid names, see Names and identifiers (p. 213). You cannot rename the current database. Only the database owner or a superuser can rename a database; non-superuser owners must also have the CREATEDB privilege.

   new_name
      New database name.

OWNER TO
   Changes the owner of the specified database. You can change the owner of the current database or some other database. Only a superuser can change the owner.

   new_owner
      New database owner. The new owner must be an existing database user with write privileges. See GRANT (p. 346) for more information about user privileges.

Usage notes

ALTER DATABASE commands apply to subsequent sessions not current sessions. You need to reconnect to the altered database to see the effect of the change.

Examples

The following example renames a database named TICKIT_SANDBOX to TICKIT_TEST:

   alter database ticket_sandbox rename to ticket_test;

The following example changes the owner of the TICKIT database (the current database) to DWUSER:

   alter database ticket owner to dwuser;

ALTER GROUP

Changes a user group. Use this command to add users to the group, drop users from the group, or rename the group.
**Synopsis**

```sql
ALTER GROUP group_name
{|ADD USER username [, ... ] |DROP USER username [, ... ] |
RENAME TO new_name
}
```

**Parameters**

- `group_name`  
  Name of the user group to modify.
- `ADD`  
  Adds a user to a user group.
- `DROP`  
  Removes a user from a user group.
- `username`  
  Name of the user to add to the group or drop from the group.
- `RENAME TO`  
  Renames the user group. For more information about valid names, see Names and identifiers (p. 213).
- `new_name`  
  New name of the user group.

**Examples**

The following example adds a user named DWUSER to the ADMIN_GROUP group:

```sql
alter group admin_group
add user dwuser;
```

The following example renames the group ADMIN_GROUP to ADMINISTRATORS:

```sql
alter group admin_group
rename to administrators;
```

**ALTER SCHEMA**

Changes the definition of an existing schema. Use this command to rename or change the owner of a schema.

For example, rename an existing schema to preserve a backup copy of that schema when you plan to create a newer version of that schema. See CREATE SCHEMA (p. 309) for more information about schemas.

**Synopsis**

```sql
ALTER SCHEMA schema_name
{|RENAME TO new_name |
```
Parameters

schema_name
   Name of the database schema to be altered.
RENAME TO
   Renames the schema.
new_name
   The new name of the schema. For more information about valid names, see Names and identifiers (p. 213).
OWNER TO
   Changes the owner of the schema.
new_owner
   The new owner of the schema.

Examples

The following example renames the SALES schema to US_SALES:

```sql
alter schema sales
rename to us_sales;
```

The following example gives ownership of the US_SALES schema to the user DWUSER:

```sql
alter schema us_sales
owner to dwuser;
```

ALTER TABLE

Topics
- Synopsis (p. 261)
- Parameters (p. 262)
- ALTER TABLE examples (p. 264)
- ALTER TABLE ADD and DROP COLUMN examples (p. 264)

Changes the definition of a database table. This command updates the values and properties set by CREATE TABLE.

Note
ALTER TABLE locks the table for reads and writes until the operation completes.

Synopsis

```sql
ALTER TABLE table_name
{
  ADD table_constraint | DROP table_constraint [ RESTRICT | CASCADE ] |
```
ALTER TABLE

OWNER TO new_owner
RENAME TO new_name
RENAME COLUMN column_name TO new_name
ADD [ COLUMN ] column_name column_type
[ DEFAULT default_expr ]
[ ENCODE encoding ]
[ NOT NULL | NULL ]
DROP [ COLUMN ] column_name [ RESTRICT | CASCADE ]

where table_constraint is:

[ CONSTRAINT constraint_name ]
{ UNIQUE ( column_name [, ... ] ) } | PRIMARY KEY ( column_name [, ... ] ) | FOREIGN KEY (column_name [, ... ] )
REFERENCES reftable [ ( refcolumn ) ]

Parameters

table_name
Name of the table to alter. Specify either just the name of the table, or use the format schema_name.table_name to use a specific schema. You can also specify a view name if you are using the ALTER TABLE statement to rename a view or change its owner.

ADD table_constraint
Adds the specified constraint to the table. See CREATE TABLE (p. 310) for descriptions of valid table_constraint values.

Note
You cannot add a primary-key constraint to a nullable column. If the column was originally created with the NOT NULL constraint, you can add the primary-key constraint.

DROP table_constraint
Drops a constraint from a table. Drops the specified constraint from the table. See CREATE TABLE (p. 310) for descriptions of valid table_constraint values.

RESTRICT
Removes only that constraint. Option for DROP CONSTRAINT. Cannot be used with CASCADE.

CASCADE
Removes constraint and anything dependent on that constraint. Option for DROP CONSTRAINT. Cannot be used with RESTRICT.

OWNER TO new_owner
Changes the owner of the table (or view) to the new_owner value.

RENAME TO new_name
Renames a table (or view) to the value specified in new_name. The maximum table name length is 127 characters; longer names are truncated to 127 characters.

RENAME COLUMN column_name TO new_name
Renames a column to the value specified in new_name. The maximum column name length is 127 characters; longer names are truncated to 127 characters. For more information about valid names, see Names and identifiers (p. 213).

ADD [ COLUMN ] column_name
Adds a column with the specified name to the table. You can add only one column in each ALTER TABLE statement.

You cannot add a column that is the distribution key (DISTKEY) or a sort key (SORTKEY) of the table.
You cannot use an ALTER TABLE ADD COLUMN command to modify the following table and column attributes:

- UNIQUE
- PRIMARY KEY
- REFERENCES (foreign key)
- IDENTITY

The maximum column name length is 127 characters; longer names are truncated to 127 characters. The maximum number of columns you can define in a single table is 1,600.

column_type
The data type of the column being added. For CHAR and VARCHAR columns, you can use the MAX keyword instead of declaring a maximum length. MAX sets the maximum length to 4096 bytes for CHAR or 65535 bytes for VARCHAR. Amazon Redshift supports the following Data types (p. 214)

- SMALLINT (INT2)
- INTEGER (INT, INT4)
- BIGINT (INT8)
- DECIMAL (NUMERIC)
- REAL (FLOAT4)
- DOUBLE PRECISION (FLOAT8)
- BOOLEAN (BOOL)
- CHAR (CHARACTER)
- VARCHAR (CHARACTER VARYING)
- DATE
- TIMESTAMP

DEFAULT default_expr
Assigns a default data value for the column. The data type of default_expr must match the data type of the column.

The default_expr is used in any INSERT operation that does not specify a value for the column. If no default value is specified, the default value for the column is null.

If a COPY operation encounters a null field on a column that has a DEFAULT value and a NOT NULL constraint, the COPY command inserts the value of the default_expr.

ENCODE encoding
Compression encoding for a column. RAW is the default, if no compression is selected. The following Compression encodings (p. 91) are supported:

- BYTEDICT
- DELTA
- DELTA32K
- MOSTLY8
- MOSTLY16
- MOSTLY32
- RAW (no compression, the default setting)
- RUNLENGTH
- TEXT255
- TEXT32K

NOT NULL | NULL
NOT NULL specifies that the column is not allowed to contain null values. NULL, the default, specifies that the column accepts null values.

DROP [ COLUMN ] column_name
Name of the column to delete from the table.
You cannot drop a column that is the distribution key (DISTKEY) or a sort key (SORTKEY) of the table. The default behavior for DROP COLUMN is RESTRICT if the column has any dependent objects, such as a view, primary key, foreign key, or UNIQUE restriction.

RESTRICT
When used with DROP COLUMN, RESTRICT means that if a defined view references the column that is being dropped, or if a foreign key references the column, or if the column takes part in a multi-part key, then the column will not be dropped. Cannot be used with CASCADE.

CASCADE
When used with DROP COLUMN, removes the specified column and anything dependent on that column. Cannot be used with RESTRICT.

**ALTER TABLE examples**

The follow examples demonstrate basic usage of the ALTER TABLE command.

**Rename a table**

The following command renames the USERS table to USERS_BKUP:

```
alter table users
rename to users_bkup;
```

You can also use this type of command to rename a view.

**Change the owner of a table or view**

The following command changes the VENUE table owner to the user DWUSER:

```
alter table venue
owner to dwuser;
```

The following commands create a view, then change its owner:

```
create view vdate as select * from date;
alter table vdate owner to vuser;
```

**Rename a column**

The following command renames the VENUESEATS column in the VENUE table to VENUESIZE:

```
alter table venue
rename column venueseats to venuesize;
```

**ALTER TABLE ADD and DROP COLUMN examples**

The following examples demonstrate how to use ALTER TABLE to add and then drop a basic table column and also how to drop a column with a dependent object.
ADD then DROP a basic column

The following example adds a standalone FEEDBACK_SCORE column to the USERS table. This column simply contains an integer, and the default value for this column is NULL (no feedback score).

First, query the PG_TABLE_DEF catalog table to view the USERS table:

```sql
select "column", type, encoding, distkey, sortkey
from pg_table_def where tablename = 'users';
```

<table>
<thead>
<tr>
<th>column</th>
<th>type</th>
<th>encoding</th>
<th>distkey</th>
<th>sortkey</th>
</tr>
</thead>
<tbody>
<tr>
<td>userid</td>
<td>integer</td>
<td>none</td>
<td>t</td>
<td>1</td>
</tr>
<tr>
<td>username</td>
<td>character(8)</td>
<td>none</td>
<td>f</td>
<td>0</td>
</tr>
<tr>
<td>firstname</td>
<td>character varying(30)</td>
<td>text32k</td>
<td>f</td>
<td>0</td>
</tr>
<tr>
<td>lastname</td>
<td>character varying(30)</td>
<td>text32k</td>
<td>f</td>
<td>0</td>
</tr>
<tr>
<td>city</td>
<td>character varying(30)</td>
<td>text32k</td>
<td>f</td>
<td>0</td>
</tr>
<tr>
<td>state</td>
<td>character(2)</td>
<td>none</td>
<td>f</td>
<td>0</td>
</tr>
<tr>
<td>email</td>
<td>character varying(100)</td>
<td>text255</td>
<td>f</td>
<td>0</td>
</tr>
<tr>
<td>phone</td>
<td>character(14)</td>
<td>none</td>
<td>f</td>
<td>0</td>
</tr>
<tr>
<td>likesports</td>
<td>boolean</td>
<td>none</td>
<td>f</td>
<td>0</td>
</tr>
<tr>
<td>liketheatre</td>
<td>boolean</td>
<td>none</td>
<td>f</td>
<td>0</td>
</tr>
<tr>
<td>likeconcerts</td>
<td>boolean</td>
<td>none</td>
<td>f</td>
<td>0</td>
</tr>
<tr>
<td>likejazz</td>
<td>boolean</td>
<td>none</td>
<td>f</td>
<td>0</td>
</tr>
<tr>
<td>likeclassical</td>
<td>boolean</td>
<td>none</td>
<td>f</td>
<td>0</td>
</tr>
<tr>
<td>likeopera</td>
<td>boolean</td>
<td>none</td>
<td>f</td>
<td>0</td>
</tr>
<tr>
<td>likerock</td>
<td>boolean</td>
<td>none</td>
<td>f</td>
<td>0</td>
</tr>
<tr>
<td>likevegas</td>
<td>boolean</td>
<td>none</td>
<td>f</td>
<td>0</td>
</tr>
<tr>
<td>likebroadway</td>
<td>boolean</td>
<td>none</td>
<td>f</td>
<td>0</td>
</tr>
<tr>
<td>likemusicals</td>
<td>boolean</td>
<td>none</td>
<td>f</td>
<td>0</td>
</tr>
</tbody>
</table>

(18 rows)

Now add the feedback_score column:

```sql
alter table users
add column feedback_score int
default NULL;
```

Select the FEEDBACK_SCORE column from USERS to verify that it was added:

```sql
select feedback_score from users limit 5;
```

feedback_score
--------------

(5 rows)

Drop the column to reinstate the original DDL:

```sql
alter table users drop column feedback_score;
```

DROPPING a column with a dependent object

This example drops a column that has a dependent object. As a result, the dependent object is also dropped.
To start, add the `FEEDBACK_SCORE` column to the `USERS` table again:

```sql
alter table users
add column feedback_score int
default NULL;
```

Next, create a view from the `USERS` table called `USERS_VIEW`:

```sql
create view users_view as select * from users;
```

Now, try to drop the `FEEDBACK_SCORE` column from the `USERS` table. This DROP statement uses the default behavior (RESTRICT):

```sql
alter table users drop column feedback_score;
```

Amazon Redshift displays an error message that the column cannot be dropped because another object depends on it.

Try dropping the `FEEDBACK_SCORE` column again, this time specifying CASCADE to drop all dependent objects:

```sql
alter table users
drop column feedback_score cascade;
```

### ALTER USER

Changes a database user account. You must be a database superuser to execute this command.

**Synopsis**

```sql
ALTER USER username [ WITH ] option [, ... ]
```

where `option` is

- `CREATEDB` | `NOCREATEDB` |
- `CREATEUSER` | `NOCREATEUSER` |
- `PASSWORD 'password'` [ `VALID UNTIL 'expiration_date'` ] |
- `RENAME TO new_name` |
- `SET parameter { TO | = } { value | DEFAULT }` |
- `RESET parameter` |

**Parameters**

- `username`
  - Name of the user account.
- `WITH`
  - Optional keyword.
- `CREATEDB` | `NOCREATEDB`
  - The `CREATEDB` option allows the user to create new databases. `NOCREATEDB` is the default.
CREATEUSER | NOCREATEUSER
The CREATEUSER option creates a superuser with all database privileges, including CREATE USER. The default is NOCREATEUSER. For more information, see Superusers (p. 84).

PASSWORD
Changes the password for a user.

`password`
Value of the new password.

Constraints:
• 8 to 64 characters in length.
• Must contain at least one uppercase letter, one lowercase letter, and one number.
• Can use any printable ASCII characters (ASCII code 33 to 126) except ' (single quote), " (double quote), \, /, @, or space.

VALID UNTIL 'expiration_date'
Specifies that the password has an expiration date. Use the value 'infinity' to avoid having an expiration date. The valid data type for this parameter is a timestamp without time zone.

RENAME TO
 Renames the user account.

`new_name`
New name of the user. For more information about valid names, see Names and identifiers (p. 213).

Important
When you rename a user, you must also change the user's password. The user name is used as part of the password encryption, so when a user is renamed, the password is cleared. The user will not be able to log in until the password is reset. For example:

```
ALTER USER
```

alter user newuser password 'EXAMPLENewPassword11';

SET
Sets a configuration parameter to a new default value for all sessions run by the specified user.

RESET
Resets a configuration parameter to the original default value for the specified user.

```
RESET
```

parameter
Name of the parameter to set or reset.

```
value
```
New value of the parameter.

DEFAULT
Sets the configuration parameter to the default value for all sessions run by the specified user.

Usage notes

When you set the search_path (p. 696) parameter with the ALTER USER command, the modification takes effect on the specified user's next login. If you want to change the search_path for the current user and session, use a SET command.

Examples

The following example gives the user ADMIN the privilege to create databases:

```
alter user admin createdb;
```
The following example sets the password of the user ADMIN to adminPass9 and sets an expiration date and time for the password:

```
alter user admin password 'adminPass9'
valid until '2013-12-31 23:59';
```

The following example renames the user ADMIN to SYSADMIN:

```
alter user admin rename to sysadmin;
```

**ANALYZE**

Updates table statistics for use by the query planner.

**Synopsis**

```
ANALYZE [ VERBOSE ]
[ [ table_name ]
[ { column_name [, ... ] } ] ]
```

**Parameters**

- **VERBOSE**
  Returns progress information messages about the ANALYZE operation. This option is useful when you do not specify a table.

- **table_name**
  You can analyze specific tables, including temporary tables. You can qualify the table with its schema name. You can optionally specify a table_name to analyze a single table. You cannot specify more than one table_name with a single ANALYZE table_name statement. If you do not specify a table_name, all of the tables in the currently connected database are analyzed, including the persistent tables in the system catalog. You do not need to analyze Amazon Redshift system tables (STL and STV tables).

- **column_name**
  If you specify a table_name, you can also specify one or more columns in the table (as a column-separated list within parentheses).

**Usage notes**

Amazon Redshift automatically analyzes tables that you create with the following commands:

- CREATE TABLE AS
- CREATE TEMP TABLE AS
- SELECT INTO

You do not need to run the ANALYZE command on these tables when they are first created. If you modify them, you should analyze them in the same way as other tables.

See also Analyzing tables (p. 158).
Examples

Analyze all of the tables in the TICKIT database and return progress information:

```
analyze verbose;
```

Analyze the LISTING table only:

```
analyze listing;
```

Analyze the VENUEID and VENUENAME columns in the VENUE table:

```
analyze venue(venueid, venuename);
```

ANALYZE COMPRESSION

Perform compression analysis and produce a report with the suggested column encoding schemes for the tables analyzed.

Synopsis

```
ANALYZE COMPRESSION
[ [ table_name ]
[ { column_name [, ...] } ] ]
[COMPROWS numrows]
```

Parameters

**table_name**

You can analyze compression for specific tables, including temporary tables. You can qualify the table with its schema name. You can optionally specify a `table_name` to analyze a single table. If you do not specify a `table_name`, all of the tables in the currently connected database are analyzed. You cannot specify more than one `table_name` with a single ANALYZE COMPRESSION statement.

**column_name**

If you specify a `table_name`, you can also specify one or more columns in the table (as a column-separated list within parentheses).

**COMPROWS**

Number of rows to be used as the sample size for compression analysis. The analysis is run on rows from each data slice. For example, if you specify COMPROWS 1000000 (1,000,000) and the system contains 4 total slices, no more than 250,000 rows per slice are read and analyzed. If COMPROWS is not specified, the sample size defaults to 100,000 per slice. Values of COMPROWS lower than the default of 100,000 rows per slice are automatically upgraded to the default value. However, compression analysis will not produce recommendations if the amount of data in the table is insufficient to produce a meaningful sample. If the COMPROWS number is greater than the number of rows in the table, the ANALYZE COMPRESSION command still proceeds and runs the compression analysis against all of the available rows.

**numrows**

Number of rows to be used as the sample size for compression analysis. The accepted range for `numrows` is a number between 1000 and 1000000000 (1,000,000,000).
Usage notes

Run ANALYZE COMPRESSION to get recommendations for column encoding schemes, based on a sample of the table's contents. ANALYZE COMPRESSION is an advisory tool and does not modify the column encodings of the table. The suggested encoding can be applied by recreating the table, or creating a new table with the same schema. Recreating an uncompressed table with appropriate encoding schemes can significantly reduce its on-disk footprint, saving disk space and improving query performance for IO-bound workloads.

ANALYZE COMPRESSION does not consider Runlength encoding (p. 96) encoding on any column that is designated as a SORTKEY because range-restricted scans might perform poorly when SORTKEY columns are compressed much more highly than other columns.

ANALYZE COMPRESSION acquires an exclusive table lock, which prevents concurrent reads and writes against the table. Only run the ANALYZE COMPRESSION command when the table is idle.

Examples

Analyze the LISTING table only:

```sql
analyze compression listing;
```

<table>
<thead>
<tr>
<th>Table</th>
<th>Column</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>listing</td>
<td>listid</td>
<td>delta</td>
</tr>
<tr>
<td>listing</td>
<td>sellerid</td>
<td>delta32k</td>
</tr>
<tr>
<td>listing</td>
<td>eventid</td>
<td>delta32k</td>
</tr>
<tr>
<td>listing</td>
<td>dateid</td>
<td>bytedict</td>
</tr>
<tr>
<td>listing</td>
<td>numtickets</td>
<td>bytedict</td>
</tr>
<tr>
<td>listing</td>
<td>priceperticket</td>
<td>delta32k</td>
</tr>
<tr>
<td>listing</td>
<td>totalprice</td>
<td>mostly32</td>
</tr>
<tr>
<td>listing</td>
<td>listtime</td>
<td>raw</td>
</tr>
</tbody>
</table>

Analyze the QTYSOLD, COMMISSION, and SALETIME columns in the SALES table:

```sql
analyze compression sales(qtysold, commission, saletime);
```

<table>
<thead>
<tr>
<th>Table</th>
<th>Column</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>sales</td>
<td>salesid</td>
<td>N/A</td>
</tr>
<tr>
<td>sales</td>
<td>listid</td>
<td>N/A</td>
</tr>
<tr>
<td>sales</td>
<td>sellerid</td>
<td>N/A</td>
</tr>
<tr>
<td>sales</td>
<td>buyerid</td>
<td>N/A</td>
</tr>
<tr>
<td>sales</td>
<td>eventid</td>
<td>N/A</td>
</tr>
<tr>
<td>sales</td>
<td>dateid</td>
<td>N/A</td>
</tr>
<tr>
<td>sales</td>
<td>qtysold</td>
<td>bytedict</td>
</tr>
<tr>
<td>sales</td>
<td>pricepaid</td>
<td>N/A</td>
</tr>
<tr>
<td>sales</td>
<td>commission</td>
<td>delta32k</td>
</tr>
<tr>
<td>sales</td>
<td>saletime</td>
<td>raw</td>
</tr>
</tbody>
</table>

BEGIN

Starts a transaction. Synonymous with START TRANSACTION.
A transaction is a single, logical unit of work, whether it consists of one command or multiple commands. In general, all commands in a transaction execute on a snapshot of the database whose starting time is determined by the value set for the transaction_snapshot_begin system configuration parameter.

By default, individual Amazon Redshift operations (queries, DDL statements, loads) are automatically committed to the database. If you want to suspend the commit for an operation until subsequent work is completed, you need to open a transaction with the BEGIN statement, then run the required commands, then close the transaction with a COMMIT statement. If necessary, you can use a ROLLBACK statement to abort a transaction that is in progress. An exception to this behavior is the TRUNCATE (p. 394) command, which commits the transaction in which it is run and cannot be rolled back.

**Synopsis**

```
BEGIN [ WORK | TRANSACTION ] [ ISOLATION LEVEL option ] [ READ WRITE | READ ONLY ]

START TRANSACTION [ ISOLATION LEVEL option ] [ READ WRITE | READ ONLY ]
```

Where `option` is

- **SERIALIZABLE**
- **READ UNCOMMITTED**
- **READ COMMITTED**
- **REPEATABLE READ**

**Note:** READ UNCOMMITTED, READ COMMITTED, and REPEATABLE READ have no operational impact and map to SERIALIZABLE in Amazon Redshift.

**Parameters**

- **WORK**
  Optional keyword.

- **TRANSACTION**
  Optional keyword; WORK and TRANSACTION are synonyms.

- **ISOLATION LEVEL SERIALIZABLE**
  Serializable isolation is supported by default, so the behavior of the transaction is the same whether or not this syntax is included in the statement. See Managing concurrent write operations (p. 169).

  No other isolation levels are supported.

  **Note**
  The SQL standard defines four levels of transaction isolation to prevent *dirty reads* (where a transaction reads data written by a concurrent uncommitted transaction), *nonrepeatable reads* (where a transaction re-reads data it read previously and finds that data was changed by another transaction that committed since the initial read), and *phantom reads* (where a transaction re-executes a query, returns a set of rows that satisfy a search condition, and then finds that the set of rows has changed because of another recently-committed transaction):

  - Read uncommitted: Dirty reads, nonrepeatable reads, and phantom reads are possible.
  - Read committed: Nonrepeatable reads and phantom reads are possible.
  - Repeatable read: Phantom reads are possible.
  - Serializable: Prevents dirty reads, nonrepeatable reads, and phantom reads.

  Though you can use any of the four transaction isolation levels, Amazon Redshift processes all isolation levels as serializable.

- **READ WRITE**
  Gives the transaction read and write permissions.
READ ONLY
  Gives the transaction read-only permissions.

Examples

The following example starts a serializable transaction block:

begin;

The following example starts the transaction block with a serializable isolation level and read and write permissions:

begin read write;

CANCEL

Cancels a database query that is currently running.

The CANCEL command requires the process ID of the running query and displays a confirmation message to verify that the query was cancelled.

Synopsis

CANCEL process_ID [ 'message' ]

Parameters

process_ID
  Process ID corresponding to the query that you want to cancel.

'message'
  An optional confirmation message that displays when the query cancellation completes. If you do not specify a message, Amazon Redshift displays the default message as verification. You must enclose the message in single quotes.

Usage notes

You cannot cancel a query by specifying a query ID; you must specify the query's process ID. You can only cancel queries currently being run by your user. Superusers can cancel all queries.

Examples

To cancel a currently running query, first retrieve the process ID for the query that you want to cancel. To determine the process IDs for all currently running queries, type the following command:

```sql
select pid, starttime, duration,
trim(user_name) as user,
trim(query) as querytxt
from stv_recent
where status = 'Running';
```
Check the query text to determine which process ID (PID) corresponds to the query that you want to cancel.

Type the following command to use PID 802 to cancel that query:

```sql
cancel 802;
```

The session where the query was running displays the following message:

```
ERROR: Query (168) cancelled on user's request
```

where 168 is the query ID (not the process ID used to cancel the query).

Alternatively, you can specify a custom confirmation message to display instead of the default message. To specify a custom message, include your message in quotes at the end of the CANCEL command:

```sql
cancel 802 'Long-running query';
```

The session where the query was running displays the following message:

```
ERROR: Long-running query
```

**CLOSE**

(Optional) Closes all of the free resources that are associated with an open cursor. COMMIT (p. 275), END (p. 339), and ROLLBACK (p. 358) automatically close the cursor, so it is not necessary to use the CLOSE command to explicitly close the cursor.

For more information, see DECLARE (p. 328), FETCH (p. 344).

**Synopsis**

```sql
CLOSE cursor
```

**Parameters**

- `cursor`
  - Name of the cursor to close.
CLOSE Example

The following commands close the cursor and perform a commit, which ends the transaction:

```sql
close movie_cursor;
commit;
```

COMMENT

Creates or changes a comment about a database object.

**Synopsis**

```sql
COMMENT ON
{ TABLE object_name |
  COLUMN object_name.column_name | 
  CONSTRAINT constraint_name ON table_name | 
  DATABASE object_name | 
  VIEW object_name
}
IS 'text'
```

**Parameters**

- `object_name` 
  Name of the database object being commented on. You can add a comment to the following objects:
  - TABLE
  - COLUMN (also takes a `column_name`).
  - CONSTRAINT (also takes a `constraint_name` and `table_name`).
  - DATABASE
  - VIEW

- `IS 'text'`
  The text of the comment that you want to apply to the specified object. Enclose the comment in single quotation marks.

- `column_name`
  Name of the column being commented on. Parameter of COLUMN. Follows a table specified in `object_name`.

- `constraint_name`
  Name of the constraint that is being commented on. Parameter of CONSTRAINT.

- `table_name`
  Name of a table containing the constraint. Parameter of CONSTRAINT.

- `arg1_type, arg2_type, ...`
  Data types of the arguments for a function. Parameter of FUNCTION.

**Usage notes**

Comments on databases may only be applied to the current database. A warning message is displayed if you attempt to comment on a different database. The same warning is displayed for comments on databases that do not exist.
Comments cannot be retrieved directly by using SQL commands. Comments can be viewed by using the PostresSQL psql too, using the \d family of commands.

**Example**

The following example adds a descriptive comment to the EVENT table:

```
comment on table event is 'Contains listings of individual events.';
```

The following example uses the psql \dd command to view the comments. Amazon Redshift does not support psql directly. You must execute psql commands from the PostgreSQL psql client. Result

```
\dd event

Object descriptions
schema | name | object | description
--------+-------+--------+-----------------------------------------
public | event | table | Contains listings of individual events.
(1 row)
```

**COMMIT**

Commits the current transaction to the database. This command makes the database updates from the transaction permanent.

**Synopsis**

```
COMMIT [ WORK | TRANSACTION ]
```

**Parameters**

- **WORK**
  Optional keyword.
- **TRANSACTION**
  Optional keyword; WORK and TRANSACTION are synonyms.

**Examples**

Each of the following examples commits the current transaction to the database:

```
commit;
commit work;
commit transaction;
```
COPY

Topics
• COPY from Amazon S3, Amazon EMR, Remote Hosts (SSH) (p. 276)
• COPY from Amazon DynamoDB Synopsis (p. 277)
• Parameters (p. 277)
• Usage notes (p. 290)
• COPY examples (p. 299)

Loads data into a table from flat files located in an Amazon S3 bucket, Amazon EMR cluster, Amazon DynamoDB table, or remote host using an SSH connection. The COPY command appends the new input data to any existing rows in the table.

**Note**
To use the COPY command, you must have INSERT privilege for the Amazon Redshift table.

**COPY from Amazon S3, Amazon EMR, Remote Hosts (SSH)**

The following synopsis gives the syntax for COPY from Amazon S3, COPY from Amazon EMR, and COPY from a remote host (COPY from SSH). A separate synopsis gives the syntax for a COPY from Amazon DynamoDB.

```sql
COPY table_name [ (column1 [,column2, ...]) ]
FROM
{ 's3://copy_from_s3_objectpath'
| 's3://copy_from_s3_manifest_file'
| 's3://copy_from_ssh_manifest_file'
| 'emr://emr_cluster_id/hdfs_filepath'
}
[ WITH ] CREDENTIALS [AS] 'aws_access_credentials'
[ option [ ... ] ]

where option is

{ FIXEDWIDTH 'fixedwidth_spec'
| DELIMITER [ AS ] 'delimiter_char'
| CSV [ QUOTE [ AS ] 'quote_character'
| JSON [AS] { 'auto' | 's3://jsonpaths_file' } }

| REGION [AS] 'aws_region'
| MANIFEST
| SSH
| ENCRYPTED
| GZIP
| LZOP
| REMOVEQUOTES
| EXPLICIT_IDS
| ACCEPTINVCCHARS [ AS ] ['replacement_char']
| MAXERROR [ AS ] error_count
| DATEFORMAT [ AS ] { 'dateformat_string' | 'auto' }
| TIMEFORMAT [ AS ] { 'timeformat_string' | 'auto' | 'epochsecs' | 'epochmilli
secs' }
| IGNOREHEADER [ AS ] number_rows
| ACCEPTANYDATE
| IGNOREBLANKLINES
```
COPY from Amazon DynamoDB Synopsis

```
COPY table_name [ (column1 [,column2, ...]) ]
FROM 'dynamodb://table_name'
[ WITH ] CREDENTIALS [AS] 'aws_access_credentials'
READRATIO ratio
[ option [ ... ] ]

where option is:

  | REGION [AS] 'aws_region'
  | EXPLICIT_IDS
  | MAXERROR [ AS ] error_count
  | DATEFORMAT [ AS ] { 'dateformat_string' | 'auto' }
  | TIMEFORMAT [ AS ] { 'timeformat_string' | 'auto' | 'epochsecs' | 'epochmillisecs' }
  | ACCEPTANYDATE
  | TRUNCATECOLUMNS
  | TRIMBLANKS
  | NOLOAD
  | EMPTYASNULL
  | BLANKSASNULL
  | COMPROWS numrows
  | COMPUPDATE [ { ON | TRUE } | { OFF | FALSE } ]
  | STATUPDATE [ { ON | TRUE } | { OFF | FALSE } ]
```

Parameters

table_name
Target table for the COPY command. The table must already exist in the database. The table can be temporary or persistent. The COPY command appends the new input data to any existing rows in the table.

(column1 [, column2, ...])
Specifies an optional column list to load data fields into specific columns. The columns can be in any order in the COPY statement, but when loading from flat files, such as in an Amazon S3 bucket, their order must match the order of the source data. Order does not matter when loading from an Amazon DynamoDB table. Any columns omitted from the column list are assigned either the defined DEFAULT expression or NULL if the omitted column is nullable and has no defined DEFAULT expression. If an omitted column is NOT NULL but has no defined DEFAULT expression, the COPY command fails.
If an IDENTITY column is included in the column list, then EXPLICIT_IDS must also be specified; if an IDENTITY column is omitted, then EXPLICIT_IDS cannot be specified. If no column list is specified, the command behaves as if a complete, in-order column list was specified (with IDENTITY columns omitted if EXPLICIT_IDS was also not specified).

FROM
Specifies the source of the data to be loaded. You can use the COPY command to load data to an Amazon Redshift table from data files located in an Amazon S3 bucket, Amazon EMR cluster, Amazon DynamoDB table, or remote host using an SSH connection. To export data from a table to a set of files, use the UNLOAD (p. 395) command.

Important
If the Amazon S3 bucket or Amazon DynamoDB table does not reside in the same region as your Amazon Redshift cluster, you must use the REGION (p. 285) option to specify the region in which the data is located. For COPY from Amazon EMR, the Amazon EMR cluster must reside in the same region as your Amazon Redshift cluster. For COPY from Amazon EMR, if the Amazon Redshift cluster is in a VPC, the Amazon EMR cluster must be in the same VPC group. If your Amazon Redshift cluster is in EC2-Classic (not in a VPC), the Amazon EMR cluster must also be in EC2-Classic.

's3://copy_from_s3_objectpath'
The path to the Amazon S3 objects that contain the data—for example, 's3://mybucket/cust.txt'. The s3://copy_from_s3_objectpath parameter can reference a single file or a set of objects or folders that have the same key prefix. For example, the name custdata.txt is a key prefix that refers to a number of physical files: custdata.txt.1, custdata.txt.2, and so on. The key prefix can also reference a number of folders. For example, 's3://mybucket/custfolder' refers to the folders custfolder_1, custfolder_2, and so on.

Important
If the Amazon S3 bucket that holds the data files does not reside in the same region as your cluster, you must use the REGION option to specify the region in which the data is located.

The maximum size of a single input row is 4 MB.

For more information, see Loading data from Amazon S3 (p. 119).

's3://copy_from_s3_manifest_file'
The Amazon S3 object key for a manifest file that lists the data files to be loaded. The 's3://copy_from_s3_manifest_file' parameter must explicitly reference a single file—for example, 's3://mybucket/manifest.txt'. It cannot be a key prefix.

The manifest is a text file in JSON format that lists the URL of each file that is to be loaded from Amazon S3. The URL includes the bucket name and full object path for the file. The files that are specified in the manifest can be in different buckets, but all the buckets must be in the same region as the Amazon Redshift cluster. The following example shows the JSON for a manifest that loads three files.

```json
{
    "entries": [
        {
            "url": "s3://mybucket-alpha/custdata.1", "mandatory": true
        },
        {
            "url": "s3://mybucket-alpha/custdata.2", "mandatory": true
        },
        {
            "url": "s3://mybucket-beta/custdata.1", "mandatory": false
        }
    ]
}
```

The double quotes are required. Each entry in the manifest can optionally include a mandatory flag. If mandatory is set to true, COPY terminates if it does not find the file for that entry; otherwise,
COPY will continue. Regardless of any mandatory settings, COPY will terminate if no files are found. The default value for mandatory is false.

The manifest file must not be encrypted or compressed, even if the ENCRYPTED, GZIP, or LZOP options are specified. COPY returns an error if the specified manifest file is not found or the manifest file is not properly formed.

If a manifest file is used, the MANIFEST option must be specified with the COPY command. If the MANIFEST option is not specified, COPY assumes that the file specified with FROM is a data file.

For more information, see Using a manifest to specify data files (p. 125).

's3://copy_from_ssh_manifest_file'

The COPY command can connect to multiple hosts using SSH, and can create multiple SSH connections to each host. COPY executes a command through each host connection, and then loads the output from the commands in parallel into the table. The s3://copy_from_ssh_manifest_file parameter specifies the Amazon S3 object key for the manifest file that provides the information COPY will use to open SSH connections and execute the remote commands.

The s3://copy_from_ssh_manifest_file parameter must explicitly reference a single file; it cannot be a key prefix. For example:

's3://mybucket/ssh_manifest.txt'

The manifest file is a text file in JSON format that Amazon Redshift uses to connect to the host. The manifest file specifies the SSH host endpoints and the commands that will be executed on the hosts to return data to Amazon Redshift. Optionally, you can include the host public key, the login user name, and a mandatory flag for each entry. The following example shows a manifest file that creates two SSH connections:

```json
{
    "entries": [
        {
            "endpoint": "<ssh_endpoint_or_IP>",
            "command": "<remote_command>",
            "mandatory": true,
            "publickey": "<public_key>",
            "username": "<host_user_name>"},
        {
            "endpoint": "<ssh_endpoint_or_IP>",
            "command": "<remote_command>",
            "mandatory": true,
            "publickey": "<public_key>",
            "username": "<host_user_name>"}
    ]
}
```

The manifest file contains one "entries" construct for each SSH connection. You can have multiple connections to a single host or multiple connections to multiple hosts. The double quotes are required as shown, both for the field names and the values. The only value that does not need double quotes is the Boolean value true or false for the "mandatory" field.

The following list describes the fields in the manifest file.

**endpoint**

The URL address or IP address of the host—for example, "ec2-111-222-333.compute-1.amazonaws.com", or "198.51.100.0".

**command**

The command that will be executed by the host to generate text or binary (gzip or lzop) output. The command can be any command that the user "host_user_name" has permission to run.
The command can be as simple as printing a file, or it could query a database or launch a script. The output (text file, gzip binary file, or lzop binary file) must be in a form the Amazon Redshift COPY command can ingest. For more information, see Preparing your input data (p. 118).

**publickey**
(Optional) The public key of the host. If provided, Amazon Redshift will use the public key to identify the host. If the public key is not provided, Amazon Redshift will not attempt host identification. For example, if the remote host's public key is `ssh-rsa AbcCbaxxx...Example root@amazon.com`, enter the following text in the publickey field: "AbcCbaxxx...Example"

**mandatory**
(Optional) Indicates whether the COPY command should fail if the connection fails. The default is `false`. If Amazon Redshift does not successfully make at least one connection, the COPY command fails.

**username**
(Optional) The user name that will be used to log in to the host system and execute the remote command. The user login name must be the same as the login that was used to add the Amazon Redshift cluster’s public key to the host’s authorized keys file. The default username is `redshift`.

For more information about creating a manifest file, see Loading data process (p. 135).

To COPY from a remote host, the SSH option must be specified with the COPY command. If the SSH option is not specified, COPY assumes that the file specified with FROM is a data file and will fail.

The following options cannot be used with COPY from SSH:
- MANIFEST
- ENCRYPTED

If you use automatic compression, the COPY command performs two data read operations, which means it will execute the remote command twice. The first read operation is to provide a data sample for compression analysis, then the second read operation actually loads the data. If executing the remote command twice might cause a problem, you should disable automatic compression. To disable automatic compression, run the COPY command with the COMPUPDATE option set to OFF. For more information, see Loading tables with automatic compression (p. 143).

For detailed procedures for using COPY from SSH, see Loading data from remote hosts (p. 134)

`dynamodb://table_name`
The name of the Amazon DynamoDB table that contains the data. For example, `dynamodb://ProductCatalog`. For details about how Amazon DynamoDB attributes are mapped to Amazon Redshift columns, see Loading data from an Amazon DynamoDB table (p. 140).

An Amazon DynamoDB table name is unique to an AWS account, which is identified by the AWS access credentials.

`emr://emr_cluster_id/hdfs_file_path`
The unique identifier for the Amazon EMR cluster and the Hadoop Distributed File System (HDFS) file path that references the data files for the COPY command. The HDFS data file names must not contain the wildcard characters asterisk (*) and question mark (?).

**Note**
The Amazon EMR cluster must continue running until the COPY operation completes. If any of the HDFS data files are changed or deleted before the COPY operation completes, you might have unexpected results, or the COPY operation might fail.

You can use the wildcard characters asterisk (*) and question mark (?) as part of the `hdfs_file_path` argument to specify multiple files to be loaded. For example, `emr://j-SAMPLE2B500FC/myoutput/part*` identifies the files `part-0000`, `part-0001`, and so on. If the file path does not contain wildcard characters, it is treated as a string literal. If you specify only a folder name, COPY attempts to load all files in the folder.
Important
If you use wildcard characters or use only the folder name, verify that no unwanted files will be loaded. For example, some processes might write a log file to the output folder.

For more information, see Loading data from Amazon EMR (p. 127).

WITH
This keyword is optional.

CREDENTIALS [AS] 'aws_access_credentials'
The AWS account access credentials for the Amazon S3 bucket or Amazon DynamoDB table, or Amazon EMR cluster that contains the data or manifest file. The access key and secret access key are required. If your data is encrypted, credentials must include a master symmetric key. If you are using temporary access credentials, you must include the temporary session token in the credentials string. For more information, see Temporary security credentials (p. 290) in the COPY usage notes.

The aws_access_credentials string must not contain spaces.

The access credentials must belong to an AWS account user or an IAM user with the following permissions:
• For COPY from Amazon S3, permission to LIST and GET the Amazon S3 objects that are being loaded, and the manifest file, if one is used.
• For COPY from Amazon S3, Amazon EMR, and remote hosts (SSH) with JSON-formatted data, permission to LIST and GET the jsonpaths file on Amazon S3, if one is used.
• For COPY from Amazon DynamoDB, permission to SCAN and DESCRIBE the Amazon DynamoDB table that is being loaded.
• For COPY from an Amazon EMR cluster, permission for the ListInstances action on the Amazon EMR cluster.

If only an access key and secret access key are required, the aws_access_credentials string is in the following format:

'aws_access_key_id=<access-key-id>;aws_secret_access_key=<secret-access-key>'

You must replace the <access-key-id> and <secret-access-key> with valid AWS account credentials or IAM user credentials.

To use temporary token credentials, you must provide the temporary access key ID, the temporary secret access key, and the temporary token. The aws_access_credentials string is in the following format:

'aws_access_key_id=<temporary-access-key-id>;aws_secret_access_key=<temporary-secret-access-key>;token=<temporary-token>'

If the ENCRYPTED option is used, the aws_access_credentials string is in the following format:

'aws_access_key_id=<access-key-id>;aws_secret_access_key=<secret-access-key>;master_symmetric_key=<master_key>

where <master_key> is the value of the master key that was used to encrypt the files.

READRATIO [AS] ratio
Specify the percentage of the Amazon DynamoDB table's provisioned throughput to use for the data load. READRATIO is required for a COPY from Amazon DynamoDB. It cannot be used with a COPY from Amazon S3. We highly recommend setting the ratio to a value less than the average unused provisioned throughput. Valid values are integers 1-200.
Caution
Setting READRATIO to 100 or higher will enable Amazon Redshift to consume the entirety of the Amazon DynamoDB table's provisioned throughput, which will seriously degrade the performance of concurrent read operations against the same table during the COPY session. Write traffic will be unaffected. Values higher than 100 are allowed to troubleshoot rare scenarios when Amazon Redshift fails to fulfill the provisioned throughput of the table. If you load data from Amazon DynamoDB to Amazon Redshift on an ongoing basis, consider organizing your Amazon DynamoDB tables as a time series to separate live traffic from the COPY operation.

'option'
Optional list of load options.

FIXEDWIDTH 'fixedwidth_spec'
Loads the data from a file where each column width is a fixed length, rather than separated by a delimiter. The fixedwidth_spec is a string that specifies a user-defined column label and column width. The column label can be either a text string or an integer, depending on what the user chooses. The column label has no relation to the column name. The order of the label/width pairs must match the order of the table columns exactly. FIXEDWIDTH cannot be used with DELIMITER. In Amazon Redshift, the length of CHAR and VARCHAR columns is expressed in bytes, so be sure that the column width that you specify accommodates the binary length of multibyte characters when preparing the file to be loaded. For more information, see Character types (p. 223).

The format for fixedwidth_spec is shown below:

'colLabel1:colWidth1,colLabel:colWidth2, ...'

DELIMITER ['delimiter_char']
Single ASCII character that is used to separate fields in the input file, such as a pipe character ( | ), a comma ( , ), or a tab ( \t ). Non-printing ASCII characters are supported. ASCII characters can also be represented in octal, using the format 'idd', where 'd' is an octal digit (0-7). The default delimiter is a pipe character ( | ), unless the CSV option is used, in which case the default delimiter is a comma ( , ). The AS keyword is optional. DELIMITER cannot be used with FIXEDWIDTH.

CSV
Enables use of CSV format in the input data. To automatically escape delimiters, newline characters, and carriage returns, enclose the field in the character specified by the QUOTE option. The default quote character is a double quote ( " ). When the quote character is used within the field, escape the character with an additional quote character. For example, if the quote character is a double quote, to insert the string " quoted " word, the input file would include the string "A ""quoted"
word". When the CSV option is used, the default delimiter is a comma ( , ). You can specify a different delimiter by using the DELIMITER option.

When a field is enclosed in quotes, white space between the delimiters and the quote characters is ignored. If the delimiter is a whitespace character, such as a tab, the delimiter is not treated as whitespace.

CSV cannot be used with FIXEDWIDTH, REMOVEQUOTES, or ESCAPE.

QUOTE [AS] 'quote_character'
Specifies the character to be used as the quote character when using the CSV option. The default is a double quote (" ). If you use the QUOTE option to define a quote character other than double quote, you don’t need to escape the double quotes within the field. The QUOTE option can be used only with the CSV option. The AS keyword is optional.

JSON [AS]
Specifies that the source data is in JSON format.

JSON format is supported for COPY from Amazon S3, COPY from Amazon EMR, and COPY from SSH. JSON is not supported for COPY from Amazon DynamoDB.
The JSON data file contains a set of either objects or arrays. COPY loads each JSON object or array into one row in the target table. Each object or array corresponding to a row must be a standalone, root-level structure; that is, it must not be a member of another JSON structure.

A JSON object begins and ends with braces, and contains an unordered collection of name/value pairs. Each paired name and value are separated by a colon, and the pairs are separated by commas. If you use the 'auto' option instead of specifying a jsonpaths file, the name in the name/value pairs must match the name of the corresponding column in the table. Order in a JSON object does not matter. Any names that don't match a column name are ignored. The following shows the structure of a simple JSON object.

```
{
    "column1": "value1",
    "column2": value2,
    "notacolumn" : "ignore this value"
}
```

A JSON array begins and ends with brackets, and contains an ordered collection of values separated by commas. If your data files use arrays, you must specify a jsonpaths file to match the values to columns. The following shows the structure of a simple JSON array.

```
["value1", value2]
```

The JSON must be well-formed. For example, the objects or arrays cannot be separated by commas or any other characters except white space.

The maximum size of a single JSON object or array, including braces or brackets, is 4 MB. If the maximum size is exceeded, even if the row size is less than 4 MB, the COPY command fails.

COPY loads \n as a newline character and loads \t as a tab character. To load a backslash, escape it with a backslash (\). COPY searches the specified JSON source for a well-formed, valid JSON object or array. If COPY encounters any non-white space characters before locating a usable JSON structure, or between valid JSON objects or arrays, COPY returns an error for each instance, which counts toward the MAXERROR error count. When the error count equals or exceeds MAXERROR, COPY fails.

For each error, Amazon Redshift records a row in the STL_LOAD_ERRORS system table. The LINE_NUMBER column records the last line of the JSON file that caused the error.

If IGNOREHEADER is specified, COPY ignores the specified number of lines in the JSON data. Newline characters in the JSON data are always counted for IGNOREHEADER calculations.

COPY loads empty strings as empty fields by default. If EMPTYASNULL is specified, COPY loads empty strings for CHAR and VARCHAR fields as NULL. Empty strings for other data types, such as INT, are always loaded with NULL.

The following options are not supported with JSON: FIXEDWIDTH, DELIMITER, CSV, REMOVEQUOTES, FILLRECORD, ESCAPE, NULL AS, IGNOREBLANKLINES, READRATIO.

For more information, see COPY from JSON format (p. 291) in the following usage notes. For more information about JSON data structures, go to www.json.org.

```
{ 'auto' | 's3://jsonpaths_file' }
```

If 'auto' is specified, COPY maps the data elements in the JSON source data to the columns in the target table by matching names in the source name/value pairs to the names of columns in the target table. If the JSON data consists of a set of arrays, you cannot use the 'auto' option; you must specify a jsonpaths file.
If the `s3://jsonpaths_file` parameter is specified, COPY uses the named `jsonpaths_file` to map the data elements in the JSON source data to the columns in the target table. The `s3://jsonpaths_file` parameter must be an Amazon S3 object key that explicitly references a single file, such as `s3://mybucket/jsonpaths.txt`; it cannot be a key prefix.

**Note**

If the file specified by `jsonpaths_file` has the same prefix as the path specified by `copy_from_s3_objectpath` for the data files, COPY reads the `jsonpaths_file` as a data file and returns errors. For example, if your data files use the object path `s3://mybucket/my_data.json` and your `jsonpaths_file` is `s3://mybucket/my_data.jsonpaths`, COPY attempts to load `my_data.jsonpaths` as a data file.

The `jsonpaths_file` must contain a single JSON object (not an array) with the key name "`jsonpaths`" paired with an array of JSONPath expressions that reference data elements in the JSON source data structure. A JSONPath expression refers to a JSON structure similarly to how an XPath expression refers to elements in an XML document. For more information, perform an Internet search for "JSONPath."

**Note**

If the key name is any string other than "`jsonpaths`", the COPY command does not return an error, but it ignores `jsonpaths_file` and uses the 'auto' option instead.

In the Amazon Redshift COPY syntax, a JSONPath expression must specify the explicit path to a single name element in a JSON hierarchical data structure. Amazon Redshift does not support any JSONPath elements, such as wildcard characters or filter expressions, that might resolve to an ambiguous path or multiple name elements.

Each JSONPath expression corresponds to one column in the Amazon Redshift target table. The order of the `jsonpaths` array elements must match the order of the columns in the target table or the column list, if a column list is used. If an element referenced by a JSONPath expression is not found in the JSON data, COPY attempts to load a NULL value.

The JSONPath expressions can use either bracket notation or dot notation, but you cannot mix notations. The following example shows JSONPath expressions using bracket notation.

```json
{
  "jsonpaths": [
    "$['venuename']",
    "$['venuecity']",
    "$['venuestate']",
    "$['venueseats']"
  ]
}
```

The following example shows JSONPath expressions using dot notation.

```json
{
  "jsonpaths": [
    ".venuename",
    ".venuecity",
    ".venuestate",
    ".venueseats"
  ]
}
```
The jsonpaths file must contain only a single JSON object (not an array). If the JSON is malformed, if there is more than one object, if there are any characters except white space outside the object, or if any of the array elements is an empty string or is not a string, the COPY command fails. MAXERROR does not apply to the jsonpaths file.

If the ENCRYPTED option is used with COPY, the jsonpaths file is not encrypted.

For more information, see COPY from JSON format (p. 291) in the following usage notes.

REGION [AS] 'aws_region'
Specifies the AWS region where the source data is located. REGION is required for COPY from an Amazon S3 bucket or an Amazon DynamoDB table when the AWS resource that contains the data is not in the same region as the Amazon Redshift cluster. The value for aws_region must match a region listed in the tables on the Regions and Endpoints page. COPY does not support REGION for COPY from Amazon EMR.

If the REGION option is specified, all resources, including a manifest file or multiple Amazon S3 buckets must be located in the specified region.

Note
Transferring data across regions incurs additional charges against the Amazon S3 bucket or the Amazon DynamoDB table that contains the data. For more information about pricing, go to Data Transfer OUT From Amazon S3 To Another AWS Region on the Amazon S3 Pricing page and Data Transfer OUT on the Amazon DynamoDB Pricing page.

By default, COPY assumes that the data is located in the same region as the Amazon Redshift cluster.

MANIFEST
Specifies that a manifest is used to identify the data files to be loaded from Amazon S3. If the MANIFEST option is used, COPY loads data from the files listed in the manifest referenced by 's3://copy_from_s3_manifest_file'. If the manifest file is not found, or is not properly formed, COPY fails.

ENCRIPTED
Specifies that the input files on Amazon S3 are encrypted. See Loading encrypted data files from Amazon S3 (p. 127). If the encrypted files are in gzip format, add the GZIP option.

GZIP
Specifies that the input file or files are in compressed gzip format (.gz files). The COPY operation reads the compressed file and uncompresses the data as it loads.

LZOP
Specifies that the input file or files are in compressed lzop format (.lzo files). The COPY operation reads the compressed file and uncompresses the data as it loads.

Note
COPY does not support files that are compressed using the lzop '--filter' option.

REMOVEQUOTES
Surrounding quotation marks are removed from strings in the incoming data. All characters within the quotes, including delimiters, are retained. If a string has a beginning single or double quotation mark but no corresponding ending mark, the COPY command fails to load that row and returns an error. The following table shows some simple examples of strings that contain quotes and the resulting loaded values.

<table>
<thead>
<tr>
<th>Input string</th>
<th>Loaded value with REMOVEQUOTES option</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;The delimiter is a pipe (</td>
<td>) character&quot;</td>
</tr>
<tr>
<td>'Black'</td>
<td>Black</td>
</tr>
<tr>
<td>&quot;White&quot;</td>
<td>White</td>
</tr>
</tbody>
</table>
### Explicit IDs

Use `EXPLICIT_IDS` with tables that have IDENTITY columns if you want to override the auto-generated values with explicit values from the source data files for the tables. If the command includes a column list, that list must include the IDENTITY columns to use this option. The data format for `EXPLICIT_IDS` values must match the IDENTITY format specified by the `CREATE TABLE` definition.

### ACCEPTINVCHARS [AS] `['replacement_char']`

Enables loading of data into VARCHAR columns even if the data contains invalid UTF-8 characters. When `ACCEPTINVCHARS` is specified, COPY replaces each invalid UTF-8 character with a string of equal length consisting of the character specified by `replacement_char`. For example, if the replacement character is `''`, an invalid three-byte character will be replaced with `''``'.

The replacement character can be any ASCII character except NULL. The default is a question mark (`?`). For information about invalid UTF-8 characters, see Multi-byte character load errors (p. 148).

COPY returns the number of rows that contained invalid UTF-8 characters, and it adds an entry to the `STL_REPLACEMENTS` (p. 623) system table for each affected row, up to a maximum of 100 rows per node slice. Additional invalid UTF-8 characters are also replaced, but those replacement events are not recorded.

If `ACCEPTINVCHARS` is not specified, COPY returns an error whenever it encounters an invalid UTF-8 character.

`ACCEPTINVCHARS` is valid only for VARCHAR columns.

### MAXERROR [AS] `error_count`

If the load returns the `error_count` number of errors or greater, the load fails. If the load returns fewer errors, it continues and returns an INFO message that states the number of rows that could not be loaded. Use this option to allow loads to continue when certain rows fail to load into the table because of formatting errors or other inconsistencies in the data. Set this value to 0 or 1 if you want the load to fail as soon as the first error occurs. The AS keyword is optional.

The actual number of errors reported might be greater than the specified MAXERROR because of the parallel nature of Amazon Redshift. If any node in the Amazon Redshift cluster detects that MAXERROR has been exceeded, each node reports all of the errors it has encountered.

### DATEFORMAT [AS] `{'dateformat_string' | 'auto'}`

If no DATEFORMAT is specified, the default format is `YYYY-MM-DD`. For example, an alternative valid format is `MM-DD-YYYY`.

If you want Amazon Redshift to automatically recognize and convert the date format in your source data, specify `auto`. The `auto` keyword is case sensitive. If the COPY command does not recognize the format of your date or time values, or if your date or time values use different formats, use the `auto` option with the DATEFORMAT or TIMEFORMAT parameter. The `auto` option recognizes several formats that are not supported when using a DATEFORMAT and TIMEFORMAT string. For more information, see Using Automatic Recognition with DATEFORMAT and TIMEFORMAT (p. 298).
The date format can include time information (hour, minutes, seconds), but this information is ignored. The AS keyword is optional. For more information, see DATEFORMAT and TIMEFORMAT strings (p. 297).

**TIMEFORMAT [AS] { 'timeformat_string' | 'auto' | 'epochsecs' | 'epochmillisecs' }**

If no TIMEFORMAT is specified, the default format is **YYYY-MM-DD HH:MI:SS**. For more information about `timeformat_string`, see DATEFORMAT and TIMEFORMAT strings (p. 297).

If you want Amazon Redshift to automatically recognize and convert the time format in your source data, specify 'auto'. If the COPY command does not recognize the format of your date or time values, or if your date or time values use different formats, use the 'auto' option with the DATEFORMAT or TIMEFORMAT parameter. The 'auto' option recognizes several formats that are not supported when using a DATEFORMAT and TIMEFORMAT string. For more information, see Using Automatic Recognition with DATEFORMAT and TIMEFORMAT (p. 298).

If your source data is represented as epoch time, the number of seconds or milliseconds since Jan 1, 1970 00:00:00 UTC, specify 'epochsecs' or 'epochmillisecs'.

The 'auto', 'epochsecs', and 'epochmillisecs' keywords are case sensitive.

The AS keyword is optional.

**IGNOREHEADER [ AS ] number_rows**

Treats the specified `number_rows` as a file header and does not load them. Use IGNOREHEADER to skip file headers in all files in a parallel load.

**ACCEPTANYDATE**

Allows any date format, including invalid formats such as **00/00/00 00:00:00**, to be loaded without generating an error. Applies only to TIMESTAMP and DATE columns. Always use ACCEPTANYDATE with the DATEFORMAT option. If the date format for the data does not match the DATEFORMAT specification, Amazon Redshift inserts a NULL value into that field.

**IGNOREBLANKLINES**

Ignores blank lines that only contain a line feed in a data file and does not try to load them.

**TRUNCATECOLUMNS**

Truncates data in columns to the appropriate number of characters so that it fits the column specification. Applies only to columns with a VARCHAR or CHAR data type, and rows 4 MB or less in size.

**FILLRECORD**

Allows data files to be loaded when contiguous columns are missing at the end of some of the records. The missing columns are filled with either zero-length strings or NULLs, as appropriate for the data types of the columns in question. If the EMPTYASNULL option is present in the COPY command and the missing column is a VARCHAR column, NULLs are loaded; if EMPTYASNULL is not present and the column is a VARCHAR, zero-length strings are loaded. NULL substitution only works if the column definition allows NULLs.

For example, if the table definition contains four nullable CHAR columns, and a record contains the values **apple, orange, banana, mango**, the COPY command could load and fill in a record that contains only the values **apple, orange**. The missing CHAR values would be loaded as NULL values.

**TRIMBLANKS**

Removes the trailing whitespace characters from a VARCHAR string. Only applicable to columns with a VARCHAR data type.

**NOLOAD**

Checks the validity of the data file without actually loading the data. Use the NOLOAD option to make sure that your data file will load without any errors before running the actual data load. Running COPY with the NOLOAD option is much faster than loading the data since it only parses the files.

**NULL AS 'null_string'**

Loads fields that match `null_string` as NULL, where `null_string` can be any string. This option cannot be used with numeric columns. To load NULL into numeric columns, such as INT, use an empty field.
If your data includes a null terminator, also referred to as NUL (UTF-8 0000) or binary zero (0x000), COPY treats it as an end of record (EOR) and terminates the record. If a field contains only NUL, you can use NULL AS to replace the null terminator with NULL by specifying '\0' or '"\000"'. For example, NULL AS '"\0' or NULL AS '"\000' . If a field contains a string that ends with NUL and NULL AS is specified, the string is inserted with NUL at the end. Do not use '"n' (newline) for the null_string value. Amazon Redshift reserves '"n' for use as a line delimiter. The default null_string is '"n'.

**Note**

If you attempt to load nulls into a column defined as NOT NULL, the COPY command will fail.

**EMPTYASNULL**

Indicates that Amazon Redshift should load empty CHAR and VARCHAR fields as NULL. Empty fields for other data types, such as INT, are always loaded with NULL. Empty fields occur when data contains two delimiters in succession with no characters between the delimiters. EMPTYASNULL and NULL AS '"' (empty string) produce the same behavior.

**BLANKSASNULL**

Loads blank fields, which consist of only white space characters, as NULL. This option applies only to CHAR and VARCHAR columns. Blank fields for other data types, such as INT, are always loaded with NULL. For example, a string that contains three space characters in succession (and no other characters) is loaded as a NULL. The default behavior, without this option, is to load the space characters as is.

**COMPROWS numrows**

Number of rows to be used as the sample size for compression analysis. The analysis is run on rows from each data slice. For example, if you specify `COMPROWS 1000000` (1,000,000) and the system contains 4 total slices, no more than 250,000 rows per slice are read and analyzed.

If COMPROWS is not specified, the sample size defaults to 100,000 per slice. Values of COMPROWS lower than the default of 100,000 rows per slice are automatically upgraded to the default value. However, automatic compression will not take place if the amount of data being loaded is insufficient to produce a meaningful sample.

If the COMPROWS number is greater than the number of rows in the input file, the COPY command still proceeds and runs the compression analysis against all of the available rows. The accepted range for this option is a number between 1000 and 1000000000 (1,000,000,000).

**COMPUPDATE [ { ON | TRUE } | { OFF | FALSE } ]**

Controls whether compression encodings are automatically applied during a COPY.

The COPY command will automatically choose the optimal compression encodings for each column in the target table based on a sample of the input data. For more information, see Loading tables with automatic compression (p. 143).

If COMPUPDATE is omitted, COPY applies automatic compression only if the target table is empty and all the table columns either have RAW encoding or no encoding. This is the default behavior.

With COMPUPDATE ON (or TRUE), COPY applies automatic compression if the table is empty, even if the table columns already have encodings other than RAW. Existing encodings are replaced. If COMPUPDATE is specified, this is the default.

With COMPUPDATE OFF (or FALSE), automatic compression is disabled.

**STATUPDATE [ { ON | TRUE } | { OFF | FALSE } ]**

Governs automatic computation and refresh of optimizer statistics at the end of a successful COPY command. By default, if the STATUPDATE option is not used, statistics are updated automatically if the table is initially empty. See also Analyzing tables (p. 158).

Whenever ingesting data into a nonempty table significantly changes the size of the table, we recommend updating statistics either by running an ANALYZE (p. 268) command or by using the STATUPDATE ON option.
With STATUPDATE ON (or TRUE), statistics are updated automatically regardless of whether the table is initially empty. If STATUPDATE is used, the current user must be either the table owner or a superuser. If STATUPDATE is not specified, only INSERT permission is required.

With STATUPDATE OFF (or FALSE), statistics are never updated.

ESCAPE
When this option is specified, the backslash character (\) in input data is treated as an escape character. The character that immediately follows the backslash character is loaded into the table as part of the current column value, even if it is a character that normally serves a special purpose. For example, you can use this option to escape the delimiter character, a quote, an embedded newline, or the escape character itself when any of these characters is a legitimate part of a column value.

If you specify the ESCAPE option in combination with the REMOVEQUOTES option, you can escape and retain quotes (‘ or ”) that might otherwise be removed. The default null string, \N, works as is, but can also be escaped in the input data as \\N. As long as you do not specify an alternative null string with the NULL AS option, \N and \\N produce the same results.

Note
The control character 0x00 (NUL) cannot be escaped and should be removed from the input data or converted. This character is treated as an end of record (EOR) marker, causing the remainder of the record to be truncated.

You cannot use the ESCAPE option for FIXEDWIDTH loads, and you cannot specify the escape character itself; the escape character is always the backslash character. Also, you must ensure that the input data contains the escape character in the appropriate places.

Here are some examples of input data and the resulting loaded data when the ESCAPE option is specified. The result for row 4 assumes that the REMOVEQUOTES option is also specified. The input data consists of two pipe-delimited fields:

| 1 | The quick brown fox\[newline] jumped over the lazy dog. |
| 2 | A\B\C |
| 3 | A | B | C |
| 4 | 'A Midsummer Night\'s Dream' |

The data loaded into column 2 looks like this:

The quick brown fox
jumped over the lazy dog.
A\B\C
A|B|C
A Midsummer Night's Dream

Note
Applying the escape character to the input data for a load is the responsibility of the user. One exception to this requirement is when you reload data that was previously unloaded with the ESCAPE option. In this case, the data will already contain the necessary escape characters.

The ESCAPE option does not interpret octal, hex, unicode, or other escape sequence notation. For example, if your source data contains the octal linefeed value (\012) and you try to load this data with the ESCAPE option, Amazon Redshift loads the value 012 into the table and does not interpret this value as a linefeed that is being escaped.
In order to escape newlines in data that originates from Windows platforms, you might need to use two escape characters: one for the carriage return and one for the linefeed. Alternatively, you can remove the carriage returns before loading the file (for example, by using the dos2unix utility).

**ROUNDEC**

Rounds up numeric values when the scale of the input value is greater than the scale of the column. By default, COPY truncates values when necessary to fit the scale of the column. For example, if a value of 20.259 is loaded into a DECIMAL(8,2) column, COPY truncates the value to 20.25 by default. If ROUNDEC is specified, COPY rounds the value to 20.26. The INSERT command always rounds values when necessary to match the column's scale, so a COPY command with the ROUNDEC option behaves the same as an INSERT command.

**Usage notes**

**Topics**

- Loading multi-byte data from Amazon S3 (p. 290)
- Errors when reading multiple files (p. 290)
- Temporary security credentials (p. 290)
- COPY from JSON format (p. 291)
- DATEFORMAT and TIMEFORMAT strings (p. 297)
- Using Automatic Recognition with DATEFORMAT and TIMEFORMAT (p. 298)

**Loading multi-byte data from Amazon S3**

If your data includes non-ASCII multi-byte characters (such as Chinese or Cyrillic characters), you must load the data to VARCHAR columns. The VARCHAR data type supports four-byte UTF-8 characters, but the CHAR data type only accepts single-byte ASCII characters. You cannot load five-byte or longer characters into Amazon Redshift tables. For more information, see Multi-byte characters (p. 215).

**Errors when reading multiple files**

The COPY command is atomic and transactional. Even when the COPY command reads data from multiple files, the entire process is treated as a single transaction. If COPY encounters an error reading a file, it automatically retries until the process times out (see statement_timeout (p. 697)) or if data cannot be download from Amazon S3 for a prolonged period of time (between 15 and 30 minutes), ensuring that each file is loaded only once. If the COPY command fails, the entire transaction is aborted and all changes are rolled back. For more information about handling load errors, see Troubleshooting data loads (p. 146).

**Temporary security credentials**

You can limit the access users have to your data by using temporary security credentials. Temporary security credentials provide enhanced security because they have short life spans and cannot be reused after they expire. The access key id and secret access key generated with the token cannot be used without the token, and a user who has these temporary security credentials can access your resources only until the credentials expire.

To grant users temporary access to your resources, you call the AWS Security Token Service (STS) APIs. The AWS STS APIs return temporary security credentials consisting of a security token, an access key id, and a secret access key. You issue the temporary security credentials to the users who need temporary access to your resources. These users can be existing IAM users, or they can be non-AWS users. For more information about creating temporary security credentials, see Using Temporary Security Credentials in the AWS Identity and Access Management (IAM) documentation.

To use temporary security credentials with a COPY command, include the `token=option` in the credentials string. You must supply the access key id and secret access key that were provided with the token.
Note
These examples contain line breaks for readability. Do not include line breaks or spaces in your
aws_access_credentials string.

The syntax for a COPY command with temporary security credentials is as follows:

```
copy table_name
from 's3://objectpath'
credentials 'aws_access_key_id=<temporary-access-key-id>;aws_secret_access_key=<temporary-secret-access-key>;token=<temporary-token>''
```

The following example loads the LISTING table using temporary credentials and file encryption:

```
copy listing
from 's3://mybucket/data/listings_pipe.txt'
credentials 'aws_access_key_id=<temporary-access-key-id>aws_secret_access_key=<temporary-secret-access-key>;token=<temporary-token>;master_symmetric_key=<master_key>'
encrypted;
```

Important
The temporary security credentials must be valid for the entire duration of the COPY statement. If the temporary security credentials expire during the load process, the COPY will fail and the transaction will be rolled back. For example, if temporary security credentials expire after 15 minutes and the COPY requires one hour, the COPY will fail before it completes.

COPY from JSON format
The JSON data structure is made up of a set of objects or arrays. An object begins and ends with braces, and contains an unordered collection of name/value pairs. Each name and value are separated by a colon, and the pairs are separated by commas. The name is a string in double quotation marks. A JSON array begins and ends with brackets, and contains an ordered collection of values separated by commas. A value can be a string in double quotation marks, a number, a Boolean true or false, null, a JSON object, or an array. JSON objects and arrays can be nested, enabling a hierarchical data structure. The following example shows a JSON data structure with two valid objects.

```
{
    "id": 1006410,
    "title": "Amazon Redshift Database Developer Guide"
}
{
    "id": 100540,
    "name": "Amazon Simple Storage Service Developer Guide"
}
```

The following shows the same data as two JSON arrays.

```
[
    1006410,
    "Amazon Redshift Database Developer Guide"
]
[
    100540,
    "Amazon Simple Storage Service Developer Guide"
]
COPY uses a jsonpaths file to parse the JSON source data. A jsonpaths file is a text file that contains a single JSON object with the name "jsonpaths" paired with an array of JSONPath expressions. If the name is any string other than "jsonpaths", COPY uses the 'auto' option instead of using the jsonpaths file.

In the Amazon Redshift COPY syntax, a JSONPath expression specifies the explicit path to a single name element in a JSON hierarchical data structure, using either bracket notation or dot notation. Amazon Redshift does not support any JSONPath elements, such as wildcard characters or filter expressions, that might resolve to an ambiguous path or multiple name elements.

The following is an example of a jsonpaths file with JSONPath expressions using bracket notation. The dollar sign ($) represents the root-level structure.

```
{
  "jsonpaths": [
    "$['id']",
    "$['store']['book']['title']",
    "$['location'][0]"
  ]
}
```

In the previous example, "$['location'][0]" references the first element in an array. JSON uses zero-based array indexing. Array indices must be positive integers (greater than or equal to zero).

The following example shows the previous jsonpaths file using dot notation.

```
{
  "jsonpaths": [
    ".id",
    ".store.book.title",
    ".location[0]"
  ]
}
```

You cannot mix bracket notation and dot notation in the jsonpaths array. Brackets can be used in both bracket notation and dot notation to reference an array element.

When using dot notation, the jsonpath expressions must not contain the following characters:

- Single straight quotation mark (')
- Period, or dot (.)
- Brackets ([]) unless used to reference an array element

If the value in the name/value pair referenced by a JSONPath expression is an object or an array, the entire object or array is loaded as a string, including the braces or brackets. For example, suppose your JSON data contains the following object.

```
{
  "id": 0,
  "guid": "84512477-fa49-456b-b407-581d0d851c3c",
  "isActive": true,
  "tags": [
    "nisi",
    "culpa",
    "ad"
  ]
}
```
The JSONPath expression $['tags'] would return the following value.

"["nisi","culpa","ad","amet","voluptate","reprehenderit","veniam"]"

The JSONPath expression $['friends'][1] would return the following value.

{"id": 1,"name": "Renaldo"}

Each JSONPath expression in the jsonpaths array corresponds to one column in the Amazon Redshift target table. The order of the jsonpaths array elements must match the order of the columns in the target table or the column list, if a column list is used.

The following examples show how to load the same data into the CATEGORY table using either the 'auto' option or a jsonpaths file, and using either JSON objects or arrays.

<table>
<thead>
<tr>
<th>column</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>catid</td>
<td>smallint</td>
</tr>
<tr>
<td>catgroup</td>
<td>character varying(10)</td>
</tr>
<tr>
<td>catname</td>
<td>character varying(10)</td>
</tr>
<tr>
<td>catdesc</td>
<td>character varying(50)</td>
</tr>
</tbody>
</table>
| (4 rows) 

**Copy from JSON Examples**

In the following examples, you will load the CATEGORY table with the following data.

<table>
<thead>
<tr>
<th>CATID</th>
<th>CATGROUP</th>
<th>CATNAME</th>
<th>CATDESC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Sports</td>
<td>MLB</td>
<td>Major League Baseball</td>
</tr>
<tr>
<td>2</td>
<td>Sports</td>
<td>NHL</td>
<td>National Hockey League</td>
</tr>
<tr>
<td>3</td>
<td>Sports</td>
<td>NFL</td>
<td>National Football League</td>
</tr>
<tr>
<td>4</td>
<td>Sports</td>
<td>NBA</td>
<td>National Basketball Association</td>
</tr>
</tbody>
</table>
### Load from JSON Data Using the 'auto' Option

To load from JSON data using the 'auto' option, the JSON data must consist of a set of objects. The key names must match the column names, but, in this case, order does not matter. The following shows the contents of a file named `category_object_auto.json`.

```json
{
    "catdesc": "Major League Baseball",
    "catid": 1,
    "catgroup": "Sports",
    "catname": "MLB"
}
{
    "catgroup": "Sports",
    "catid": 2,
    "catname": "NHL",
    "catdesc": "National Hockey League"
}
{
    "catid": 3,
    "catname": "NFL",
    "catgroup": "Sports",
    "catdesc": "National Football League"
}
{
    "bogus": "Bogus Sports LLC",
    "catid": 4,
    "catgroup": "Sports",
    "catname": "NBA",
    "catdesc": "National Basketball Association"
}
{
    "catid": 5,
    "catgroup": "Shows",
    "catname": "Musicals",
    "catdesc": "All symphony, concerto, and choir concerts"
}
```

To load from the JSON data file in the previous example, execute the following COPY command.

```
copy category
from 's3://mybucket/category_object_auto.json'
credentials 'aws_access_key_id=<access-key-id>;aws_secret_access_key=<secret-access-key>'
json 'auto';
```

### Load from JSON Data Using a jsonpaths File

If the JSON data objects don't correspond directly to column names, you can use a jsonpaths file to map the JSON elements to columns. Again, the order does not matter in the JSON source data, but the order of the jsonpaths file expressions must match the column order. Suppose you have the following data file, named `category_object_paths.json`.

```json
{
    "catdesc": "Major League Baseball",
    "catid": 1,
    "catgroup": "Sports",
    "catname": "MLB"
}
{
    "catgroup": "Sports",
    "catid": 2,
    "catname": "NHL",
    "catdesc": "National Hockey League"
}
{
    "catid": 3,
    "catname": "NFL",
    "catgroup": "Sports",
    "catdesc": "National Football League"
}
{
    "bogus": "Bogus Sports LLC",
    "catid": 4,
    "catgroup": "Sports",
    "catname": "NBA",
    "catdesc": "National Basketball Association"
}
{
    "catid": 5,
    "catgroup": "Shows",
    "catname": "Musicals",
    "catdesc": "All symphony, concerto, and choir concerts"
}
```

To load from the JSON data file in the previous example, execute the following COPY command.

```
copy category
json 'auto';
```
The following jsonpath file, named category_object_jsonpath.json, maps the source data to the table columns.

```
{
  "jsonpaths": [
    "$['one']",
    "$['two']",
    "$['three']",
    "$['four']"
  ]
}
```

To load from the JSON data file in the previous example, execute the following COPY command.

```
copy category
from 's3://mybucket/category_object_paths.json'
credentials 'aws_access_key_id=<access-key-id>;aws_secret_access_key=<secret-access-key>'
json 's3://mybucket/category_object_auto.json';
```

### Load from JSON Arrays Using a jsonpaths File

To load from JSON data that consists of a set of arrays, you must use a jsonpaths file to map the array elements to columns. Suppose you have the following data file, named category_array_data.json.

```
[{
  "one": 1,
  "two": "Sports",
  "three": "MLB",
  "four": "Major League Baseball"
}
{
  "three": "NHL",
  "four": "National Hockey League",
  "one": 2,
  "two": "Sports",
}
{
  "two": "Sports",
  "three": "NFL",
  "one": 3,
  "four": "National Football League"
}
{
  "one": 4,
  "two": "Sports",
  "three": "NBA",
  "four": "National Basketball Association"
}
{
  "one": 6,
  "two": "Shows",
  "three": "Musicals",
  "four": "All symphony, concerto, and choir concerts"
}
```
The following jsonpaths file, named `category_array_jsonpath.json`, maps the source data to the table columns.

```json
{
  "jsonpaths": [  
      "$[0]",
      "$[1]",
      "$[2]",
      "$[3]"
  ]
}
```

To load from the JSON data file in the previous example, execute the following COPY command.

```sql
copy category
from 's3://mybucket/category_array_data.json'
credentials 'aws_access_key_id=<access-key-id>;aws_secret_access_key=<secret-access-key>'
json 's3://mybucket/category_array_jsonpath.json';
```

**Escape Characters in JSON**

COPY loads \n as a newline character and loads \t as a tab character. To load a backslash, escape it with a backslash (\\).

For example, suppose you have the following JSON in a file named `escape.json` in the bucket `s3://dw-tickit/json/`.

```json
{
  "backslash": "This is a backslash: \\",
  "newline": "This sentence\n is on two lines."
  "tab": "This sentence \t contains a tab."
}
```

Execute the following commands to create the ESCAPES table and load the JSON.

```sql
create table escapes (backslash varchar(25), newline varchar(35), tab varchar(35));

copy escapes from 's3://mybucket/json/escape.json' with
credentials 'aws_access_key_id=<access-key-id>;aws_secret_access_key=<secret-access-key>'
json 'auto';
```

Query the ESCAPES table to view the results.
select * from escapes;

<table>
<thead>
<tr>
<th>backslash</th>
<th>newline</th>
<th>tab</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

This is a backslash: \ | This sentence | This sentence contains a tab.
: is on two lines.

(1 row)

**DATEFORMAT and TIMEFORMAT strings**

The **DATEFORMAT** and **TIMEFORMAT** options in the COPY command take format strings. These strings can contain datetime separators (such as ‘-’, ‘/’, or ‘:’) and the following "dateparts" and "timeparts":

**Note**

If the COPY command does not recognize the format of your date or time values, or if your date or time values use different formats, use the ‘auto’ option with the **TIMEFORMAT** parameter. The ‘auto’ option recognizes several formats that are not supported when using a **DATEFORMAT** and **TIMEFORMAT** string.

<table>
<thead>
<tr>
<th>Datepart/timepart</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>YY</td>
<td>Year without century</td>
</tr>
<tr>
<td>YYYY</td>
<td>Year with century</td>
</tr>
<tr>
<td>MM</td>
<td>Month as a number</td>
</tr>
<tr>
<td>MON</td>
<td>Month as a name (abbreviated name or full name)</td>
</tr>
<tr>
<td>DD</td>
<td>Day of month as a number</td>
</tr>
<tr>
<td>HH or HH24</td>
<td>Hour (24-hour clock)</td>
</tr>
<tr>
<td>HH12</td>
<td>Hour (12-hour clock)</td>
</tr>
<tr>
<td>MI</td>
<td>Minutes</td>
</tr>
<tr>
<td>SS</td>
<td>Seconds</td>
</tr>
<tr>
<td>AM or PM</td>
<td>Meridian indicator (for 12-hour clock)</td>
</tr>
</tbody>
</table>

The default timestamp format is **YYYY-MM-DD HH:MI:SS**, and the default date format is **YYYY-MM-DD**. The seconds (**SS**) field also supports fractional seconds up to microsecond granularity. You must specify a space character between the date and time sections of the **TIMEFORMAT** string, as shown in the example below.

For example, the following **DATEFORMAT** and **TIMEFORMAT** strings are valid:

<table>
<thead>
<tr>
<th>COPY syntax</th>
<th>Example of valid input string</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DATEFORMAT AS 'MM/DD/YYYY'</strong></td>
<td>03/31/2003</td>
</tr>
<tr>
<td><strong>DATEFORMAT AS 'MON DD, YYYY'</strong></td>
<td>March 31, 2003</td>
</tr>
</tbody>
</table>
Using Automatic Recognition with DATEFORMAT and TIMEFORMAT

If you specify 'auto' as the parameter for the DATEFORMAT or TIMEFORMAT option, Amazon Redshift will automatically recognize and convert the date format or time format in your source data. For example:

```
copy favoritemovies from 'dynamodb://ProductCatalog'
credentials 'aws_access_key_id=<access-key-id>; aws_secret_access_key=<secret-access-key>'
dateformat 'auto';
```

When used with the 'auto' option for DATEFORMAT and TIMEFORMAT, COPY recognizes and converts the date and time formats listed in the table in DATEFORMAT and TIMEFORMAT strings (p. 297). In addition, the 'auto' option recognizes the following formats that are not supported when using a DATEFORMAT and TIMEFORMAT string:

<table>
<thead>
<tr>
<th>Format</th>
<th>Example of valid input string</th>
</tr>
</thead>
<tbody>
<tr>
<td>Julian</td>
<td>J2451187</td>
</tr>
<tr>
<td>BC</td>
<td>Jan-08-95 BC</td>
</tr>
<tr>
<td>YYYYMMDD HHMISS</td>
<td>19960108 040809</td>
</tr>
<tr>
<td>YYMMDD HHMISS</td>
<td>960108 040809</td>
</tr>
<tr>
<td>YYYY.DDD</td>
<td>1996.008</td>
</tr>
<tr>
<td>YYYY-MM-DD HH:MI:SSSS</td>
<td>1996-01-08 04:05:06.789</td>
</tr>
</tbody>
</table>

To test whether a date or timestamp value will be automatically converted, use a CAST function to attempt to convert the string to a date or timestamp value. For example, the following commands test the timestamp value 'J2345678 04:05:06.789':

```
create table formattest (test char(16));
insert into formattest values('J2345678 04:05:06.789');
select test, cast(test as timestamp) as timestamp, cast(test as date) as date
from formattest;
```

```
<table>
<thead>
<tr>
<th>test</th>
<th>timestamp</th>
<th>date</th>
</tr>
</thead>
<tbody>
<tr>
<td>J2345678 04:05:06.789</td>
<td>1710-02-23</td>
<td>1710-02-23</td>
</tr>
</tbody>
</table>
```

If the source data for a DATE column includes time information, the time component is truncated. If the source data for a TIMESTAMP column omits time information, 00:00:00 is used for the time component.
COPY examples

Note
These examples contain line breaks for readability. Do not include line breaks or spaces in your `aws_access_credentials` string.

Load FAVORITEMOVIES from an Amazon DynamoDB table

The AWS SDKs include a simple example of creating an Amazon DynamoDB table called `my-favorite-movies-table`. (See AWS SDK for Java.) This example loads the Amazon Redshift FAVORITEMOVIES table with data from the Amazon DynamoDB table. The Amazon Redshift table must already exist in the database.

```sql
COPY favoritemovies from 'dynamodb://ProductCatalog'
credentials 'aws_access_key_id=<access-key-id>;aws_secret_access_key=<secret-access-key>'
readratio 50;
```

Load LISTING from an Amazon S3 bucket

The following example loads LISTING from an Amazon S3 bucket. The COPY command loads all of the files in the `/data/listing/` folder.

```sql
COPY listing
from 's3://mybucket/data/listing/
credentials 'aws_access_key_id=<access-key-id>;aws_secret_access_key=<secret-access-key>'
```

Load LISTING from an Amazon EMR cluster

The following example loads the SALES table with tab-delimited data from lzop-compressed files on an Amazon EMR cluster. COPY will load every file in the `myoutput/` folder that begins with `part-`.

```sql
COPY sales
from 'emr://j-SAMPLE2B500FC/myoutput/part-*'
credentials 'aws_access_key_id=<access-key-id>;aws_secret_access_key=<secret-access-key>'
delimiter '\t' lzop;
```

The following example loads the SALES table with JSON formatted data on an Amazon EMR cluster. COPY will load every file in the `myoutput/json/` folder.

```sql
COPY sales
from 'emr://j-SAMPLE2B500FC/myoutput/json/
credentials 'aws_access_key_id=<access-key-id>;aws_secret_access_key=<secret-access-key>'
JSON 's3://mybucket/jsonpaths.txt';
```

Using a manifest to specify data files

You can use a manifest to ensure that your COPY command loads all of the required files, and only the required files, from Amazon S3. You can also use a manifest when you need to load multiple files from different buckets or files that do not share the same prefix.
For example, suppose you need to load the following three files: `custdata1.txt`, `custdata2.txt`, and `custdata3.txt`. You could use the following command to load all of the files in `mybucket` that begin with `custdata` by specifying a prefix:

```
copy category
to 's3://mybucket/custdata'
from 's3://mybucket/custdata/custdata1.txt'
credentials 'aws_access_key_id=<access-key-id>;aws_secret_access_key=<secret-access-key>'';
```

If only two of the files exist because of an error or because of eventual consistency, COPY will load only those two files and finish successfully, resulting in an incomplete data load. If the bucket also contains a file named `custdata.backup`, COPY will load that file as well, resulting in unwanted data being loaded.

To ensure that all of the required files are loaded and to prevent unwanted files from being loaded, you can use a manifest file. The manifest is a JSON-formatted text file that lists the files to be processed by the COPY command. For example, the following manifest loads the three files in the previous example.

```
{
  "entries": [        
    {"url":"s3://mybucket/custdata.1","mandatory":true},
    {"url":"s3://mybucket/custdata.2","mandatory":true},
    {"url":"s3://mybucket/custdata.3","mandatory":true}
  ]
}
```

The optional `mandatory` flag indicates whether COPY should terminate if the file does not exist. The default is `false`. In this example, COPY will return an error if any of the files is not found. Unwanted files that might have been picked up if you specified only a key prefix, such as `custdata.backup`, are ignored, because they are not on the manifest. The following example uses the manifest in the previous example, which is named `cust.manifest`.

```
copy customer
to 's3://mybucket/cust.manifest'
from 's3://mybucket/cust.manifest'
credentials 'aws_access_key_id=<access-key-id>;aws_secret_access_key=<secret-access-key>''
manifest;
```

You can use a manifest to load files from different buckets or files that do not share the same prefix. The following example shows the JSON to load data with files whose names begin with a date stamp.

```
{
  "entries": [        
    {"url":"s3://mybucket/2013-10-04-custdata.txt","mandatory":true},
    {"url":"s3://mybucket/2013-10-05-custdata.txt","mandatory":true},
    {"url":"s3://mybucket/2013-10-06-custdata.txt","mandatory":true},
    {"url":"s3://mybucket/2013-10-07-custdata.txt","mandatory":true}
  ]
}
```

The manifest can list files that are in different buckets, as long as the buckets are in the same region as the cluster.

```
{
  "entries": [        
```

API Version 2012-12-01
300
Load LISTING from a pipe-delimited file (default delimiter)

The following example is a very simple case in which no options are specified and the input file contains the default delimiter, a pipe character ('|').

```sql
COPY listing
FROM 's3://mybucket/data/listings_pipe.txt'
credentials 'aws_access_key_id=<access-key-id>;aws_secret_access_key=<secret-access-key>';
```

Load LISTING using temporary credentials

The following example uses the TOKEN option to specify temporary session credentials:

```sql
COPY listing
FROM 's3://mybucket/data/listings_pipe.txt'
credentials 'aws_access_key_id=<temporary-access-key-id>;aws_secret_access_key=<temporary-secret-access-key>;token=<temporary-token>';
```

Load EVENT with options

This example loads pipe-delimited data into the EVENT table and applies the following rules:

- If pairs of quotes are used to surround any character strings, they are removed.
- Both empty strings and strings that contain blanks are loaded as NULL values.
- The load will fail if more than 5 errors are returned.
- Timestamp values must comply with the specified format; for example, a valid timestamp is 2008-09-26 05:43:12.

```sql
COPY event
FROM 's3://mybucket/data/allevents_pipe.txt'
credentials 'aws_access_key_id=<access-key-id>;aws_secret_access_key=<secret-access-key>'
removequotes
emptyasnull
blanksasnull
maxerror 5
delimiter '|'
timeformat 'YYYY-MM-DD HH:MI:SS';
```

Load VENUE from a fixed-width data file

```sql
COPY venue
FROM 's3://mybucket/data/venue_fw.txt'
credentials 'aws_access_key_id=<access-key-id>;aws_secret_access_key=<secret-access-key>,'
```
This example assumes a data file formatted in the same way as the sample data below. In the sample below, spaces act as placeholders so that all of the columns are the same width as noted in the specification:

<table>
<thead>
<tr>
<th></th>
<th>Venue Name</th>
<th>City</th>
<th>State</th>
<th>Seats</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Toyota Park</td>
<td>Bridgeview</td>
<td>IL</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>Columbus Crew Stadium</td>
<td>Columbus</td>
<td>OH</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>RFK Stadium</td>
<td>Washington</td>
<td>DC</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>CommunityAmerica Ballpark</td>
<td>Kansas City</td>
<td>KS</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>Gillette Stadium</td>
<td>Foxborough</td>
<td>MA</td>
<td>68756</td>
</tr>
</tbody>
</table>

Load CATEGORY from a CSV file

Suppose the file category_csv.txt contains the following text:

<table>
<thead>
<tr>
<th></th>
<th>Category</th>
<th>Subcategory</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>Shows, Musicals</td>
<td>Musical theatre</td>
<td>All musical theatre</td>
</tr>
<tr>
<td>13</td>
<td>Shows, Plays</td>
<td>All &quot;non-musical&quot; theatre</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>Shows, Opera</td>
<td>All opera, light, and &quot;rock&quot; opera</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>Concerts</td>
<td>Classical, All symphony, concerto, and choir concerts</td>
<td></td>
</tr>
</tbody>
</table>

If you load the category_csv.txt file using the DELIMITER option to specify comma-delimited input, the COPY will fail because some input fields contain commas. You can avoid that problem by using the CSV option and enclosing the fields that contain commas in quotes. If the quote character appears within a quoted string, you need to escape it by doubling the quote character. The default quote character is a double quote, so you will need to escape double quotes with an additional double quote. Your new input file will look something like this.

<table>
<thead>
<tr>
<th></th>
<th>Category</th>
<th>Subcategory</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>Shows, Musicals</td>
<td>Musical theatre</td>
<td>All musical theatre</td>
</tr>
<tr>
<td>13</td>
<td>Shows, Plays</td>
<td>All &quot;&quot;non-musical&quot;&quot; theatre</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>Shows, Opera</td>
<td>All opera, light, and &quot;&quot;rock&quot;&quot; opera</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>Concerts</td>
<td>Classical, All symphony, concerto, and choir concerts</td>
<td></td>
</tr>
</tbody>
</table>

You could load category_csv.txt by using the following COPY command:

```
copy category
from 's3://mybucket/data/category_csv.txt'
credentials 'aws_access_key_id=<access-key-id>;aws_secret_access_key=<secret-access-key>'
csv;
```

To avoid the need to escape the double quotes in your input, you can specify a different quote character by using the QUOTE AS option. For example, the following version of category_csv.txt uses `%' as the quote character:

<table>
<thead>
<tr>
<th></th>
<th>Category</th>
<th>Subcategory</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>Shows, Musicals</td>
<td>Musical theatre</td>
<td>All musical theatre</td>
</tr>
<tr>
<td>13</td>
<td>Shows, Plays</td>
<td>All &quot;%non-musical%&quot; theatre</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>Shows, Opera</td>
<td>All opera, light, and &quot;%rock%&quot; opera</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>Concerts</td>
<td>Classical, All symphony, concerto, and choir concerts</td>
<td></td>
</tr>
</tbody>
</table>

The following COPY command uses QUOTE AS to load category_csv.txt:
Load VENUE with explicit values for an IDENTITY column

This example assumes that when the VENUE table was created that at least one column (such as the venueid column) was specified to be an IDENTITY column. This command overrides the default IDENTITY behavior of auto-generating values for an IDENTITY column and instead loads the explicit values from the venue.txt file:

```sql
COPY venue
FROM 's3://mybucket/data/venue.txt'
CREDENTIALS 'aws_access_key_id=<access-key-id>;aws_secret_access_key=<secret-access-key>'
explicit_ids;
```

Load TIME from a pipe-delimited GZIP file

This example loads the TIME table from a pipe-delimited GZIP file:

```sql
COPY time
FROM 's3://mybucket/data/timerows.gz'
CREDENTIALS 'aws_access_key_id=<access-key-id>;aws_secret_access_key=<secret-access-key>'
gzip
delimiter '|';
```

Load a time/datestamp

This example loads data with a formatted timestamp.

**Note**
The TIMEFORMAT of `HH:MI:SS` can also support fractional seconds beyond the `SS` to microsecond granularity. The file `time.txt` used in this example contains one row, `2009-01-12 14:15:57.119568`.

```sql
COPY timestamp1
FROM 's3://mybucket/data/time.txt'
CREDENTIALS 'aws_access_key_id=<access-key-id>;aws_secret_access_key=<secret-access-key>'
timeformat 'YYYY-MM-DD HH:MI:SS';
```

The result of this copy is as follows:

```sql
SELECT * FROM timestamp1;
c1----------------------------
2009-01-12 14:15:57.119568
(1 row)
```
Load data from a file with default values

This example uses a variation of the VENUE table in the TICKIT database. Consider a VENUE_NEW table defined with the following statement:

```sql
create table venue_new(
    venueid smallint not null,
    venuename varchar(100) not null,
    venuecity varchar(30),
    venuestate char(2),
    venueseats integer not null default '1000');
```

Consider a venue_noseats.txt data file that contains no values for the VENUESEATS column, as shown in the following example:

<table>
<thead>
<tr>
<th></th>
<th>Toyota Park</th>
<th>Bridgeview</th>
<th>IL</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Columbus Crew Stadium</td>
<td>Columbus</td>
<td>OH</td>
</tr>
<tr>
<td>3</td>
<td>RFK Stadium</td>
<td>Washington</td>
<td>DC</td>
</tr>
<tr>
<td>4</td>
<td>CommunityAmerica Ballpark</td>
<td>Kansas City</td>
<td>KS</td>
</tr>
<tr>
<td>5</td>
<td>Gillette Stadium</td>
<td>Foxborough</td>
<td>MA</td>
</tr>
<tr>
<td>6</td>
<td>New York Giants Stadium</td>
<td>East Rutherford</td>
<td>NJ</td>
</tr>
<tr>
<td>7</td>
<td>BMO Field</td>
<td>Toronto</td>
<td>ON</td>
</tr>
<tr>
<td>8</td>
<td>The Home Depot Center</td>
<td>Carson</td>
<td>CA</td>
</tr>
<tr>
<td>9</td>
<td>Dick's Sporting Goods Park</td>
<td>Commerce City</td>
<td>CO</td>
</tr>
<tr>
<td>10</td>
<td>Pizza Hut Park</td>
<td>Frisco</td>
<td>TX</td>
</tr>
</tbody>
</table>

The following COPY statement will successfully load the table from the file and apply the DEFAULT value ('1000') to the omitted column:

```sql
copy venue_new(venueid, venuename, venuecity, venuestate)
from 's3://mybucket/data/venue_noseats.txt'
credentials 'aws_access_key_id=<access-key-id>;aws_secret_access_key=<secret-access-key>'
delimiter '|';
```

Now view the loaded table:

```sql
select * from venue_new order by venueid;
```

<table>
<thead>
<tr>
<th>venueid</th>
<th>venuename</th>
<th>venuecity</th>
<th>venuestate</th>
<th>venueseats</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Toyota Park</td>
<td>Bridgeview</td>
<td>IL</td>
<td>1000</td>
</tr>
<tr>
<td>2</td>
<td>Columbus Crew Stadium</td>
<td>Columbus</td>
<td>OH</td>
<td>1000</td>
</tr>
<tr>
<td>3</td>
<td>RFK Stadium</td>
<td>Washington</td>
<td>DC</td>
<td>1000</td>
</tr>
<tr>
<td>4</td>
<td>CommunityAmerica Ballpark</td>
<td>Kansas City</td>
<td>KS</td>
<td>1000</td>
</tr>
<tr>
<td>5</td>
<td>Gillette Stadium</td>
<td>Foxborough</td>
<td>MA</td>
<td>1000</td>
</tr>
<tr>
<td>6</td>
<td>New York Giants Stadium</td>
<td>East Rutherford</td>
<td>NJ</td>
<td>1000</td>
</tr>
<tr>
<td>7</td>
<td>BMO Field</td>
<td>Toronto</td>
<td>ON</td>
<td>1000</td>
</tr>
<tr>
<td>8</td>
<td>The Home Depot Center</td>
<td>Carson</td>
<td>CA</td>
<td>1000</td>
</tr>
<tr>
<td>9</td>
<td>Dick's Sporting Goods Park</td>
<td>Commerce City</td>
<td>CO</td>
<td>1000</td>
</tr>
<tr>
<td>10</td>
<td>Pizza Hut Park</td>
<td>Frisco</td>
<td>TX</td>
<td>1000</td>
</tr>
</tbody>
</table>

For the following example, in addition to assuming that no VENUESEATS data is included in the file, also assume that no VENUENAME data is included:
Using the same table definition, the following COPY statement will fail because no DEFAULT value was specified for VENUENAME, and VENUENAME is a NOT NULL column:

```sql
copy venue(venueid, venuename, venuecity, venuestate) from 's3://mybucket/data/venue_pipe.txt' credentials 'aws_access_key_id=<access-key-id>;aws_secret_access_key=<secret-access-key>' delimiter '|';
```

Now consider a variation of the VENUE table that uses an IDENTITY column:

```sql
create table venue_identity(
    venueid int identity(1,1),
    venuename varchar(100) not null,
    venuecity varchar(30),
    venuestate char(2),
    venueseats integer not null default '1000'
);
```

As with the previous example, assume that the VENUESEATS column has no corresponding values in the source file. The following COPY statement will successfully load the table, including the pre-defined IDENTITY data values instead of auto-generating those values:

```sql
copy venue(venueid, venuename, venuecity, venuestate) from 's3://mybucket/data/venue_pipe.txt' credentials 'aws_access_key_id=<access-key-id>;aws_secret_access_key=<secret-access-key>' delimiter '|' explicit_ids;
```

This statement fails because it does not include the IDENTITY column (VENUEID is missing from the column list) yet includes an EXPLICIT_IDS option:

```sql
copy venue(venuename, venuecity, venuestate) from 's3://mybucket/data/venue_pipe.txt' credentials 'aws_access_key_id=<access-key-id>;aws_secret_access_key=<secret-access-key>' delimiter '|' explicit_ids;
```

This statement fails because it does not include an EXPLICIT_IDS option:
COPY data with the ESCAPE option

This example shows how to load characters that match the delimiter character (in this case, the pipe character). In the input file, make sure that all of the pipe characters (|) that you want to load are escaped with the backslash character (\). Then load the file with the ESCAPE option.

```
$ more redshiftinfo.txt
1|public\|event\|dwuser
2|public\|sales\|dwuser
```

```
create table redshiftinfo(infoid int, tableinfo varchar(50));
copy redshiftinfo from 's3://mybucket/data/redshiftinfo.txt'
credentials 'aws_access_key_id=<access-key-id>; aws_secret_access_key=<secret-access-key>'
delimiter '|' escape;
select * from redshiftinfo order by 1;
```

```
infoid | tableinfo
-------+-------------------
1      | public|event|dwuser
2      | public|sales|dwuser
```

Without the ESCAPE option, this COPY command would fail with an Extra column(s) found error.

**Important**

If you load your data using a COPY with the ESCAPE option, you must also specify the ESCAPE option with your UNLOAD command to generate the reciprocal output file. Similarly, if you UNLOAD using the ESCAPE option, you will need to use ESCAPE when you COPY the same data.

Preparing files for COPY with the ESCAPE option

This example describes how you might prepare data to "escape" newline characters before importing the data into an Amazon Redshift table using the COPY command with the ESCAPE option. Without preparing the data to delimit the newline characters, Amazon Redshift will return load errors when you run the COPY command, since the newline character is normally used as a record separator.

For example, consider a file or a column in an external table that you want to copy into an Amazon Redshift table. If the file or column contains XML-formatted content or similar data, you will need to make sure that all of the newline characters (\n) that are part of the content are escaped with the backslash character (\).

A good thing about a file or table containing embedded newlines characters is that it provides a relatively easy pattern to match. Each embedded newline character most likely always follows a > character with potentially some whitespace characters (\s \t or tab) in between, as you can see in the following example of a text file named nlTest1.txt.

```
$ cat nlTest1.txt
<xml start>
<newline characters provide>
<line breaks at the end of each>
```
Amazon Redshift Database Developer Guide
COPY

<line in content>
</xml>|1000
<xml>
</xml>|2000

With this example, you can run a text-processing utility to pre-process the source file and insert escape
characters where needed. (The | character is intended to be used as delimiter to separate column data
when copied into an Amazon Redshift table.)
$ sed -e ':a;N;$!ba;s/>[[:space:]]*\n/>\\\n/g'
nlTest1.txt > nlTest2.txt

Similarly, you could use Perl to perform a similar operation:
cat n1Test1.txt | perl -p -e 's/\\\n//g' > n1Test2.txt

To accommodate loading the data from the nlTest2.txt file into Amazon Redshift, we created a
two-column table in Amazon Redshift. The first column c1, is a character column that will hold
XML-formatted content from the nlTest2.txt file. The second column c2 holds integer values loaded
from the same file.
After running the sed command, you can correctly load data from the nlTest2.txt file into an Amazon
Redshift table using the ESCAPE option.

Note
When you include the ESCAPE option with the COPY command, it escapes a number of special
characters that include the backslash character (including newline).
copy t2 from 's3://mybucket/data/nlTest2.txt'
credentials 'aws_access_key_id=<access-key-id>;aws_secret_access_key=<secretaccess-key>'
escape
delimiter as '|';

select * from t2 order by 2;
c1
| c2
-------------+-----<xml start>
<newline characters provide>
<line breaks at the end of each>
<line in content>
</xml>
| 1000
<xml>
</xml>
| 2000
(2 rows)

You can prepare data files exported from external databases in a similar way. For example, with an Oracle
database, you could use the REPLACE function on each affected column in a table that you want to copy
into Amazon Redshift.
SELECT c1, REPLACE(c2, \n',\\n' ) as c2 from my_table_with_xml

API Version 2012-12-01
307


In addition, many database export and ETL tools that routinely process large amounts of data provide options to specify escape and delimiter characters.

CREATE DATABASE

Creates a new database.

Synopsis

```
CREATE DATABASE database_name [ WITH ]
[ OWNER [=] db_owner ]
```

Parameters

database_name
Name of the new database. For more information about valid names, see Names and identifiers (p. 213).

WITH
Optional keyword.

OWNER
Specifies a database owner.

=
Optional character.

db_owner
Username for the database owner.

CREATE DATABASE limits

Amazon Redshift enforces these limits for databases.

- Maximum of 60 user-defined databases per cluster.
- Maximum of 127 characters for a database name.
- Cannot be a reserved word.

Examples

The following example creates a database named TICKIT_TEST and gives ownership to the user DWUSER:

```
create database tickit_test
with owner dwuser;
```

CREATE GROUP

Defines a new user group.

Synopsis

```
CREATE GROUP group_name
[ [ WITH ] [ USER username ] [, ... ] ]
```
Parameters

group_name
    Name of the new user group. For more information about valid names, see Names and identifiers (p. 213).

WITH
    Optional syntax to indicate additional parameters for CREATE GROUP.

USER
    Add one or more users to the group.

username
    Name of the user to add to the group.

Examples

The following example creates a user group named ADMIN_GROUP with a single user ADMIN:

```sql
create group admin_group with user admin;
```

CREATE SCHEMA

Defines a new schema for the current database.

Synopsis

```
CREATE SCHEMA schema_name [ AUTHORIZATION username ] [ schema_element [ ... ] ]
```

```
CREATE SCHEMA AUTHORIZATION username [ schema_element [ ... ] ]
```

Parameters

schema_name
    Name of the new schema. For more information about valid names, see Names and identifiers (p. 213).

Note
    The list of schemas in the search_path (p. 696) configuration parameter determines the precedence of identically named objects when they are referenced without schema names.

AUTHORIZATION
    Gives ownership to a specified user.

username
    Name of the schema owner.

schema_element
    Defines one or more objects to be created within the schema.

CREATE SCHEMA limits

Amazon Redshift enforces these limits for schemas.

- Maximum of 256 schemas per database.
- Cannot be a reserved word.
Examples

The following example creates a schema named US_SALES and gives ownership to the user DWUSER:

```sql
create schema us_sales authorization dwuser;
```

To view the new schema, query the PG_NAMESPACE catalog table.

```sql
select nspname as schema, usename as owner
from pg_namespace, pg_user
where pg_namespace.nspowner = pg_user.usesysid
and pg_user.usename = 'dwuser';
```

<table>
<thead>
<tr>
<th>name</th>
<th>owner</th>
</tr>
</thead>
<tbody>
<tr>
<td>us_sales</td>
<td>dwuser</td>
</tr>
</tbody>
</table>

(1 row)

CREATE TABLE

Topics

- Synopsis (p. 310)
- Parameters (p. 311)
- CREATE TABLE usage notes (p. 315)
- CREATE TABLE examples (p. 316)

Creates a new table in the current database. The owner of this table is the issuer of the CREATE TABLE command.

Synopsis

```sql
CREATE [ [LOCAL ] { TEMPORARY | TEMP } ] TABLE table_name
( { column_name data_type [column_attributes] [ column_constraints ]
  | table_constraints
  | LIKE parent_table [ { INCLUDING | EXCLUDING } DEFAULTS ] }
|[, ... ]
)[table_attribute]
```

where column_attributes are:

- [ DEFAULT default_expr ]
- [ IDENTITY ( seed, step ) ]
- [ ENCODE encoding ]
- [ DISTKEY ]
- [ SORTKEY ]

and column_constraints are:

- [ { NOT NULL | NULL } ]
- [ { UNIQUE | PRIMARY KEY } ]
- [ REFERENCES ref_table [ { refcolumn } ] ]

and table_constraints are:
Parameters

LOCAL
Although this optional keyword is accepted in the statement, it has no effect in Amazon Redshift.

TEMPORARY | TEMP
Creates a temporary table that is visible only within the current session. The table is automatically dropped at the end of the session in which it is created. The temporary table can have the same name as a permanent table. The temporary table is created in a separate, session-specific schema. (You cannot specify a schema name.) This temporary schema becomes the first schema in the search path, so the temporary table will take precedence over the permanent table unless you qualify the table name with the schema name to access the permanent table. For more information about schemas and precedence, see See search_path (p. 696).

Note
By default, users have permission to create temporary tables by their automatic membership in the PUBLIC group. To deny this privilege to a user, revoke the TEMP privilege from the PUBLIC group, and then explicitly grant the TEMP privilege only to specific users or groups of users.

table_name
The name of the table to be created.

Important
If you specify a table name that begins with '# ', the table will be created as a temporary table. For example:

```sql
create table #newtable (id int);
```

The maximum length for the table name is 127 characters; longer names are truncated to 127 characters. Amazon Redshift enforces a maximum limit of 9,900 permanent tables per cluster. The table name may be qualified with the database and or schema name. For example:

```sql
create table tickit.public.test (c1 int);
```

In this example, tickit is the database name, public is the schema name, and test is the table name. If the database or schema does not exist, table is not created, and the statement returns an error. You cannot create tables or views in the system databases template0, template1, and padb_harvest.

If a schema name is given, the new table is created in that schema (assuming the creator has access to the schema). The table name must be a unique name for that schema. If no schema is specified, the table is created by using the current database schema. If you are creating a temporary table, you cannot specify a schema name, because temporary tables exist in a special schema.
Multiple temporary tables with the same name can exist at the same time in the same database if they are created in separate sessions because the tables are assigned to different schemas. For more information about valid names, see Names and identifiers (p. 213).

**column_name**
The name of a column to be created in the new table. The maximum length for the column name is 127 characters; longer names are truncated to 127 characters. The maximum number of columns you can define in a single table is 1,600. For more information about valid names, see Names and identifiers (p. 213).

**Note**
If you are creating a "wide table," take care that your list of columns does not exceed row-width boundaries for intermediate results during loads and query processing. For more information, see CREATE TABLE usage notes (p. 315).

**data_type**
The data type of the column being created. For CHAR and VARCHAR columns, you can use the MAX keyword instead of declaring a maximum length. MAX sets the maximum length to 4096 bytes for CHAR or 65535 bytes for VARCHAR. The following Data types (p. 214) are supported:

- SMALLINT (INT2)
- INTEGER (INT, INT4)
- BIGINT (INT8)
- DECIMAL (NUMERIC)
- REAL (FLOAT4)
- DOUBLE PRECISION (FLOAT8)
- BOOLEAN (BOOL)
- CHAR (CHARACTER)
- VARCHAR (CHARACTER VARYING)
- DATE
- TIMESTAMP

**DEFAULT default_expr**
Assigns a default data value for the column. The data type of default_expr must match the data type of the column. The DEFAULT value must be a variable-free expression. Subqueries, cross-references to other columns in the current table, and non-system defined functions are not allowed.

The default_expr is used in any INSERT operation that does not specify a value for the column. If no default value is specified, the default value for the column is null.

If a COPY operation with a defined column list omits a column that has a DEFAULT value and a NOT NULL constraint, the COPY command inserts the value of the default_expr.

If a COPY operation with a defined column list omits a column that has a DEFAULT value and is nullable, the COPY command inserts the value of the default_expr, not the NULL value.

**IDENTITY(seed, step)**
Specifies that the column is an IDENTITY column. An IDENTITY column contains unique auto-generated values. These values start with the value specified as the seed and increment by the number specified as the step. The data type for an IDENTITY column must be either INT or BIGINT. IDENTITY columns are declared NOT NULL by default and do not accept NULLs.

**ENCODE encoding**
Compression encoding for a column. RAW is the default, if no compression is selected. The following Compression encodings (p. 91) are supported:

- BYTEDICT
- DELTA
- DELTA32K
- LZO
DISTKEY
Specifies that the column is the distribution key for the table. Only one column in a table can be the
distribution key. You can use the DISTKEY keyword after a column name or as part of the table
definition by using the DISTKEY (column_name) syntax. Either method has the same effect. For
more information, see the DISTSTYLE keyword definition.

SORTKEY
Specifies that the column is the sort key for the table. When data is loaded into the table, the data is
sorted by one or more columns that are designated as sort keys. You can use the SORTKEY keyword
after a column name to specify a single-column sort key, or you can specify one or more columns
as sort key columns for the table by using the SORTKEY (column_name [, ...]) syntax.

If you do not specify any sort keys, the table is not sorted. You can define a maximum of 400
SORTKEY columns per table.

NOT NULL | NULL
NOT NULL specifies that the column is not allowed to contain null values. NULL, the default, specifies
that the column accepts null values. IDENTITY columns are declared NOT NULL by default.

UNIQUE
Specifies that the column may contain only unique values. The behavior of the unique table constraint
is the same as that for column constraints, with the additional capability to span multiple columns.
To define a unique table constraint, use the UNIQUE (column_name [, ...] ) syntax.

Important
Unique constraints are informational and are not enforced by the system.

PRIMARY KEY
Specifies that the column is the primary key for the table. Only one column can be defined as the
primary key by using a column definition. To define a table constraint with a multiple-column primary
key, use the PRIMARY KEY (column_name [, ...] ) syntax.

Identifying a column as the primary key provides metadata about the design of the schema. A primary
key implies that other tables can rely on this set of columns as a unique identifier for rows. One
primary key can be specified for a table, whether as a column constraint or a table constraint. The
primary key constraint should name a set of columns that is different from other sets of columns
named by any unique constraint defined for the same table.

Important
Primary key constraints are informational only. They are not enforced by the system, but
they are used by the planner.

References ref_table [ (refcolumn) ]
Specifies a foreign key constraint, which implies that the column must contain only values that match
values in the referenced column of some row of the referenced table. The referenced columns should
be the columns of a unique or primary key constraint in the referenced table.

Important
Foreign key constraints are informational only. They are not enforced by the system, but
they are used by the planner.

LIKE parent_table [ (INCLUDING | EXCLUDING ) DEFAULTS ]
Specifies an existing table from which the new table automatically copies column names, data types,
and NOT NULL constraints. The new table and the parent table are decoupled, and any changes
made to the parent table are not applied to the new table. Default expressions for the copied column
definitions are copied only if INCLUDING DEFAULTS is specified. The default behavior is to exclude default expressions, so that all columns of the new table have null defaults.

Tables created with the LIKE option do not inherit primary and foreign key constraints. Distribution style, sort keys, and NULL properties are inherited by LIKE tables but they cannot be explicitly set in the CREATE TABLE statement.

**DISTSTYLE (EVEN | KEY | ALL)**

Defines the data distribution style for the whole table. Amazon Redshift distributes the rows of a table to the compute nodes according the distribution style specified for the table.

The distribution style that you select for tables affects the overall performance of your database. For more information, see Choosing a data distribution style (p. 101).

- **EVEN**: The data in the table is spread evenly across the nodes in a cluster in a round-robin distribution. Row IDs are used to determine the distribution, and roughly the same number of rows are distributed to each node. This is the default distribution method.

- **KEY**: The data is distributed by the values in the DISTKEY column. When you set the joining columns of joining tables as distribution keys, the joining rows from both tables are collocated on the compute nodes. When data is collocated, the optimizer can perform joins more efficiently. If you specify DISTSTYLE KEY, you must name a DISTKEY column, either for the table or as part of the column definition. For more information, see the DISTKEY keyword definition.

- **ALL**: A copy of the entire table is distributed to every node. This distribution style ensures that all the rows required for any join are available on every node, but it multiplies storage requirements and increases the load and maintenance times for the table. ALL distribution can improve execution time when used with certain dimension tables where KEY distribution is not appropriate, but performance improvements must be weighed against maintenance costs.

**DISTKEY (column_name)**

Specifies the column to be used as the distribution key for the table. You can use the DISTKEY keyword after a column name or as part of the table definition, by using the DISTKEY (column_name) syntax. Either method has the same effect. For more information, see the DISTSTYLE keyword definition.

**SORTKEY (column_name [...] )**

Specifies one or more sort keys for the table. When data is loaded into the table, the data is sorted by one or more columns that are designated as sort keys. You can use the SORTKEY keyword after a column name to specify a single-column sort key, or you can specify one or more columns as sort key columns for the table by using the SORTKEY (column_name [...]) syntax.

If you do not specify any sort keys, the table is not sorted by default. You can define a maximum of 400 SORTKEY columns per table.

**UNIQUE (column_name [...] )**

The UNIQUE constraint specifies that a group of one or more columns of a table may contain only unique values. The behavior of the unique table constraint is the same as that for column constraints, with the additional capability to span multiple columns. In the context of unique constraints, null values are not considered equal. Each unique table constraint must name a set of columns that is different from the set of columns named by any other unique or primary key constraint defined for the table.

**Important**

Unique constraints are informational and are not enforced by the system.

**PRIMARY KEY (column_name [...] )**

The primary key constraint specifies that a column or a number of columns of a table may contain only unique (non-duplicate) non-null values. Identifying a set of columns as the primary key also provides metadata about the design of the schema. A primary key implies that other tables may rely on this set of columns as a unique identifier for rows. One primary key can be specified for a table, whether as a single column constraint or a table constraint. The primary key constraint should name a set of columns that is different from other sets of columns named by any unique constraint defined for the same table.
Important
Primary key constraints are informational only. They are not enforced by the system, but they are used by the planner.

FOREIGN KEY ( column_name [, ... ] ) REFERENCES reftable [ ( refcolumn ) ]

Specifies a foreign key constraint, which requires that a group of one or more columns of the new table must only contain values that match values in the referenced column(s) of some row of the referenced table. If refcolumn is omitted, the primary key of the reftable is used. The referenced columns must be the columns of a unique or primary key constraint in the referenced table.

Important
Foreign key constraints are informational only. They are not enforced by the system, but they are used by the planner.

CREATE TABLE usage notes

Limits

Amazon Redshift enforces a maximum limit of 9,900 permanent tables per cluster.

The maximum number of characters for a table name is 127.

The maximum number of columns you can define in a single table is 1,600.

Summary of column-level settings and table-level settings

Several attributes and settings can be set at the column level or at the table level. In some cases, setting an attribute or constraint at the column level or at the table level has the same effect. In other cases, they produce different results.

The following list summarizes column-level and table-level settings:

DISTKEY
There is no difference in effect whether set at the column level or at the table level.

If DISTKEY is set, either at the column level or at the table level, DISTSTYLE must be set to KEY or not set at all. DISTSTYLE can be set only at the table level.

SORTKEY
If set at the column level, SORTKEY must be a single column. If set at the table level, one or more columns may make up a composite sort key.

UNIQUE
At the column level, one or more keys can be set to UNIQUE; the UNIQUE constraint applies to each column individually. If set at the table level, one or more columns may make up a composite UNIQUE constraint.

PRIMARY KEY
If set at the column level, PRIMARY KEY must be a single column. If set at the table level, one or more columns may make up a composite primary key.

FOREIGN KEY
There is no difference in effect. At the column level, the syntax is simply REFERENCES reftable [ ( refcolumn ) ].

Distribution of incoming data

When the hash distribution scheme of the incoming data matches that of the target table, no physical distribution of the data is actually necessary when the data is loaded. For example, if a distribution key is set for the new table and the data is being inserted from another table that is distributed on the same
key column, the data is loaded in place, using the same nodes and slices. However, if the source and target tables are both set to EVEN distribution, data is redistributed into the target table.

Wide tables

You might be able to create a very wide table but be unable to insert data into it or select from it. These restrictions derive from offsets required for null handling during query processing. The nullability of the columns in a table affects the behavior of loads and queries against that table:

• For query-processing purposes, a table with any nullable columns in it cannot exceed a total width of 64KB - 1 (or 65535 bytes). For example:

```
create table t (c1 varchar(30000), c2 varchar(30000), c3 varchar(10000));
insert into t values (1,1,1);
select * from t;
ERROR:  8001
DETAIL:  The combined length of columns processed in the SQL statement exceeded the query-processing limit of 65535 characters (pid:7627)
```

• In a table with only NOT NULL columns, the starting position of the last column cannot be greater than 64KB - 1 (or 65535 bytes). For example, the following table can be loaded and queried:

```
create table t (c1 varchar(30000) not null, c2 varchar(30000) not null, c3 varchar (10000) not null);
insert into t values(1,1,1);
select trim(c1), trim(c2), trim(c3) from t;
```

```
+-------+-------+-------+
| 1     | 1     | 1     |
+-------+-------+-------+
(1 row)
```

However, the following table, in which the starting position of the third column is greater than 64KB - 1, returns an error if you attempt to insert a row:

```
create table t (c1 varchar(35000) not null, c2 varchar(35000) not null, c3 varchar (10000) not null);
insert into t values (1,1,1);
ERROR:  8001
DETAIL:  The combined length of columns processed in the SQL statement exceeded the query-processing limit of 65535 characters (pid:7627)
```

CREATE TABLE examples

The following examples demonstrate various column and table attributes in Amazon Redshift CREATE TABLE statements.
**Create a table with a distribution key, a multi-column sort key, and compression**

This example creates SALES table in the TICKIT database with compression defined for several columns. LISTID is declared as the distribution key, and LISTID and SELLERID are declared as a multicolumn sort key. Primary key and foreign key constraints are also defined for the table.

```sql
create table sales(
salesid integer not null,
listid integer not null,
sellerid integer not null,
buyerid integer not null,
eventid integer not null encode mostly16,
dateid smallint not null,
qtysold smallint not null encode mostly8,
pricepaid decimal(8,2) encode delta32k,
commission decimal(8,2) encode delta32k,
saletime timestamp,
primary key(salesid),
foreign key(listid) references listing(listid),
foreign key(sellerid) references users(userid),
foreign key(buyerid) references users(userid),
foreign key(dateid) references date(dateid))
distkey(listid)
sortkey(listid,sellerid);
```

**Result**

```sql
select "column", type, encoding, distkey, sortkey
from pg_table_def where tablename = 'sales';
```

<table>
<thead>
<tr>
<th>column</th>
<th>type</th>
<th>encoding</th>
<th>distkey</th>
<th>sortkey</th>
</tr>
</thead>
<tbody>
<tr>
<td>salesid</td>
<td>integer</td>
<td>none</td>
<td>f</td>
<td>0</td>
</tr>
<tr>
<td>listid</td>
<td>integer</td>
<td>none</td>
<td>t</td>
<td>1</td>
</tr>
<tr>
<td>sellerid</td>
<td>integer</td>
<td>none</td>
<td>f</td>
<td>2</td>
</tr>
<tr>
<td>buyerid</td>
<td>integer</td>
<td>none</td>
<td>f</td>
<td>0</td>
</tr>
<tr>
<td>eventid</td>
<td>integer</td>
<td>mostly16</td>
<td>f</td>
<td>0</td>
</tr>
<tr>
<td>dateid</td>
<td>smallint</td>
<td>none</td>
<td>f</td>
<td>0</td>
</tr>
<tr>
<td>qtysold</td>
<td>smallint</td>
<td>mostly8</td>
<td>f</td>
<td>0</td>
</tr>
<tr>
<td>pricepaid</td>
<td>numeric(8,2)</td>
<td>delta32k</td>
<td>f</td>
<td>0</td>
</tr>
<tr>
<td>commission</td>
<td>numeric(8,2)</td>
<td>delta32k</td>
<td>f</td>
<td>0</td>
</tr>
<tr>
<td>saletime</td>
<td>timestamp</td>
<td>none</td>
<td>f</td>
<td>0</td>
</tr>
</tbody>
</table>

(10 rows)

**Create a table with ALL distribution**

The following example creates the VENUE table with ALL distribution.

```sql
create table venue(
venueid smallint not null,
venuename varchar(100),
venuecity varchar(30),
venuestate char(2),
venueseats integer,

```
primary key(venueid))
diststyle all;

Create a table with default EVEN distribution

This example creates a table called MYEVENT with three columns.

```sql
create table myevent(
    eventid int,
    eventname varchar(200),
    eventcity varchar(30));
```

By default, the table is distributed evenly and is not sorted. The table has no declared DISTKEY or SORTKEY columns.

```sql
select "column", type, encoding, distkey, sortkey
from pg_table_def where tablename = 'myevent';
```

<table>
<thead>
<tr>
<th>column</th>
<th>type</th>
<th>encoding</th>
<th>distkey</th>
<th>sortkey</th>
</tr>
</thead>
<tbody>
<tr>
<td>eventid</td>
<td>integer</td>
<td>none</td>
<td>f</td>
<td>0</td>
</tr>
<tr>
<td>eventname</td>
<td>character varying(200)</td>
<td>none</td>
<td>f</td>
<td>0</td>
</tr>
<tr>
<td>eventcity</td>
<td>character varying(30)</td>
<td>none</td>
<td>f</td>
<td>0</td>
</tr>
</tbody>
</table>

(3 rows)

Create a temporary table that is LIKE another table

This example creates a temporary table called TEMPEVENT, which inherits its columns from the EVENT table.

```sql
create temp table tempevent(like event);
```

This table also inherits the DISTKEY and SORTKEY attributes of its parent table:

```sql
select "column", type, encoding, distkey, sortkey
from pg_table_def where tablename = 'tempevent';
```

<table>
<thead>
<tr>
<th>column</th>
<th>type</th>
<th>encoding</th>
<th>distkey</th>
<th>sortkey</th>
</tr>
</thead>
<tbody>
<tr>
<td>eventid</td>
<td>integer</td>
<td>none</td>
<td>t</td>
<td>1</td>
</tr>
<tr>
<td>venueid</td>
<td>smallint</td>
<td>none</td>
<td>f</td>
<td>0</td>
</tr>
<tr>
<td>catid</td>
<td>smallint</td>
<td>none</td>
<td>f</td>
<td>0</td>
</tr>
<tr>
<td>dateid</td>
<td>smallint</td>
<td>none</td>
<td>f</td>
<td>0</td>
</tr>
<tr>
<td>eventname</td>
<td>character varying(200)</td>
<td>none</td>
<td>f</td>
<td>0</td>
</tr>
<tr>
<td>starttime</td>
<td>timestamp without time zone</td>
<td>none</td>
<td>f</td>
<td>0</td>
</tr>
</tbody>
</table>

(6 rows)

Create a table with an IDENTITY column

This example creates a table named VENUE_IDENT, which has an IDENTITY column named VENUEID. This column starts with 0 and increments by 1 for each record. VENUEID is also declared as the primary key of the table.
create table venue_ident(venueid bigint identity(0, 1),
  venuename varchar(100),
  venuecity varchar(30),
  venuestate char(2),
  venueseats integer,
  primary key(venueid));

Create a table with DEFAULT column values

This example creates a CATEGORYDEF table that declares default values for each column:

create table categorydef(
  catid smallint not null default 0,
  catgroup varchar(10) default 'Special',
  catname varchar(10) default 'Other',
  catdesc varchar(50) default 'Special events',
  primary key(catid));

insert into categorydef values(default,default,default,default);

select * from categorydef;

catid | catgroup | catname |    catdesc
-------+----------+---------+----------------
0 | Special  | Other   | Special events
(1 row)

DISTSTYLE, DISTKEY, and SORTKEY options

The following statement shows how the DISTKEY, SORTKEY, and DISTSTYLE options work. In this example, COL1 is the distribution key; therefore, the distribution style must be either set to KEY or not set. By default, the table has no sort key and so is not sorted:

create table t1(col1 int distkey, col2 int) diststyle key;

Result

select "column", type, encoding, distkey, sortkey
from pg_table_def where tablename = 't1';

| column |  type   | encoding | distkey | sortkey |
|--------+---------+----------+---------+---------|
| col1   | integer | none     | t       | 0       |
| col2   | integer | none     | f       | 0       |

In this example, the same column is defined as the distribution key and the sort key. Again, the distribution style must be either set to KEY or not set.

create table t2(col1 int distkey sortkey, col2 int);

Result
In this example, no column is set as the distribution key, COL2 is set as the sort key, and the distribution style is set to ALL:

```sql
create table t3(col1 int, col2 int sortkey) diststyle all;
```

Result

```
<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Encoding</th>
<th>DistKey</th>
<th>SortKey</th>
</tr>
</thead>
<tbody>
<tr>
<td>col1</td>
<td>integer</td>
<td>none</td>
<td>t</td>
<td>1</td>
</tr>
<tr>
<td>col2</td>
<td>integer</td>
<td>none</td>
<td>f</td>
<td>0</td>
</tr>
</tbody>
</table>
```

In this example, the distribution style is set to EVEN and no sort key is defined explicitly; therefore the table is distributed evenly but is not sorted.

```sql
create table t4(col1 int, col2 int) diststyle even;
```

Result

```
<table>
<thead>
<tr>
<th>column</th>
<th>type</th>
<th>encoding</th>
<th>distkey</th>
<th>sortkey</th>
</tr>
</thead>
<tbody>
<tr>
<td>col1</td>
<td>integer</td>
<td>none</td>
<td>f</td>
<td>0</td>
</tr>
<tr>
<td>col2</td>
<td>integer</td>
<td>none</td>
<td>f</td>
<td>0</td>
</tr>
</tbody>
</table>
```

**CREATE TABLE AS**

**Topics**
- Synopsis (p. 321)
- Parameters (p. 321)
- CTAS usage notes (p. 323)
- CTAS examples (p. 323)

Creates a new table based on a query. The owner of this table is the user that issues the command.

The new table is loaded with data defined by the query in the command. The table columns have names and data types associated with the output columns of the query. The CREATE TABLE AS (CTAS) command creates a new table and evaluates the query to load the new table.
Synopsis

CREATE [ [LOCAL ] { TEMPORARY | TEMP } ]
TABLE table_name [ ( column_name [, ... ] ) ]
| DISTSTYLE { EVEN | ALL | KEY }
| DISTKEY ( distkey_identifier )
| SORTKEY ( sortkey_identifier [, ... ] )
AS query

Parameters

LOCAL
Although this optional keyword is accepted in the statement, it has no effect in Amazon Redshift.

TEMPORARY or TEMP
Creates a temporary table. A temporary table is automatically dropped at the end of the session in
which it was created.

table_name
The name of the table to be created.

Important
If you specify a table name that begins with '# ', the table will be created as a temporary
table. For example:

create table #newtable (id int);

The maximum table name length is 127 characters; longer names are truncated to 127 characters.
Amazon Redshift enforces a maximum limit of 9,900 permanent tables per cluster. The table name
may be qualified with the database and or schema name. For example:

create table tickit.public.test (c1 int);

In this example, tickit is the database name and public is the schema name. If the database or
schema does not exist, the statement returns an error.

If a schema name is given, the new table is created in that schema (assuming the creator has access
to the schema). The table name must be a unique name for that schema. If no schema is specified,
the table is created using the current database schema. If you are creating a temporary table, you
cannot specify a schema name, since temporary tables exist in a special schema.

Multiple temporary tables with the same name are allowed to exist at the same time in the same
database if they are created in separate sessions. These tables are assigned to different schemas.

column_name
The name of a column in the new table. If no column names are provided, the column names are
taken from the output column names of the query. Default column names are used for expressions.

DISTSTYLE { EVEN | KEY | ALL }
Defines the data distribution style for the whole table. Amazon Redshift distributes the rows of a table
to the compute nodes according the distribution style specified for the table.

The distribution style that you select for tables affects the overall performance of your database. For
more information, see Choosing a data distribution style (p. 101).

• EVEN: The data in the table is spread evenly across the nodes in a cluster in a round-robin
distribution. Row IDs are used to determine the distribution, and roughly the same number of rows
are distributed to each node. This is the default distribution method.
• KEY: The data is distributed by the values in the DISTKEY column. When you set the joining columns of joining tables as distribution keys, the joining rows from both tables are collocated on the compute nodes. When data is collocated, the optimizer can perform joins more efficiently. If you specify DISTSTYLE KEY, you must name a DISTKEY column.

• ALL: A copy of the entire table is distributed to every node. This distribution style ensures that all the rows required for any join are available on every node, but it multiplies storage requirements and increases the load and maintenance times for the table. ALL distribution can improve execution time when used with certain dimension tables where KEY distribution is not appropriate, but performance improvements must be weighed against maintenance costs.

DISTKEY

Only one column in a table can be the distribution key:

• If you declare a column as the DISTKEY column, DISTSTYLE must be set to KEY or not set at all.

• If you do not declare a DISTKEY column, you can set DISTSTYLE to EVEN.

• If you declare neither a DISTKEY column nor a DISTSTYLE option, the default behavior is inherited from the query in the CTAS statement, if possible. For example, if the query is:

```
select * from date
```

and the DATE table is distributed on the DATEID column, that column is the inherited distribution key for the target table. If the distribution style cannot be inherited from the query, the table is evenly distributed (as if DISTSTYLE EVEN had been specified).

You can define the same column as the distribution key and the sort key; this approach tends to accelerate joins when the column in question is a joining column in the query.

distkey_identifier

A column name or positional number for the distribution key. Use the name specified in either the optional column list for the table or the select list of the query. Alternatively, use a positional number, where the first column selected is 1, the second is 2, and so on.

SORTKEY

Amazon Redshift supports multi-column sort keys. You can define a maximum of 400 SORTKEY columns per table.

When data is loaded into the table, the data is sorted by these columns. If you do not specify any keys, the default behavior is inherited from the properties of the incoming data defined in the CTAS statement, if possible. For example, if the statement is:

```
create table copydate as select * from date;
```

and the DATE table is sorted on the DATEID column, that column is the inherited sort key for the target table.

If the sort key cannot be inherited from the incoming data, the table is not sorted. For example, if the CTAS statement has no DISTSTYLE or DISTKEY setting or defines a DISTSTYLE or DISTKEY setting that requires redistribution of the incoming data, the new table is not sorted.

sortkey_identifier

One or more column names or positional numbers. Use the names specified in either the optional column list for the table or the select list of the query. Alternatively, use positional numbers, where the first column selected is 1, the second is 2, and so on.

query

Any query (SELECT statement) that Amazon Redshift supports.
CTAS usage notes

Limits

Amazon Redshift enforces a maximum limit of 9,900 permanent tables.

The maximum number of characters for a table name is 127.

The maximum number of columns you can define in a single table is 1,600.

Inheritance of column and table attributes

CTAS tables do not inherit compression encodings, constraints, identity columns, default column values, or the primary key from the table that they were created from (assuming that the original table has any of these characteristics). Distribution and sort keys are inherited where possible if the CTAS statement does not define its own keys.

Distribution of incoming data

When the hash distribution scheme of the incoming data matches that of the target table, no physical distribution of the data is actually necessary when the data is loaded. For example, if a distribution key is set for the new table and the data is being inserted from another table that is distributed on the same key column, the data is loaded in place, using the same nodes and slices. However, if the source and target tables are both set to EVEN distribution, data is redistributed into the target table.

Automatic ANALYZE operations

Amazon Redshift automatically analyzes tables that you create with CTAS commands. You do not need to run the ANALYZE command on these tables when they are first created. If you modify them, you should analyze them in the same way as other tables.

CTAS examples

The following example creates a table called EVENT_BACKUP for the EVENT table:

```
create table event_backup as select * from event;
```

The resulting table inherits the distribution and sort key from the EVENT table (EVENTID).

```
select "column", type, encoding, distkey, sortkey
from pg_table_def where tablename = 'event_backup';
```

<table>
<thead>
<tr>
<th>column</th>
<th>type</th>
<th>encoding</th>
<th>distkey</th>
<th>sortkey</th>
</tr>
</thead>
<tbody>
<tr>
<td>eventid</td>
<td>integer</td>
<td>none</td>
<td>t</td>
<td>1</td>
</tr>
<tr>
<td>venueid</td>
<td>smallint</td>
<td>none</td>
<td>f</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The following command creates a new table called EVENTDISTSORT by selecting four columns from the EVENT table. The new table is distributed by EVENTID and sorted by EVENTID and DATEID:

```
create table eventdistsort
distkey (1)
sortkey (1,3)
as
```
select eventid, venueid, dateid, eventname
from event;

Result

select "column", type, encoding, distkey, sortkey
from pg_table_def where tablename = 'eventdistsort';

<table>
<thead>
<tr>
<th>column</th>
<th>type</th>
<th>encoding</th>
<th>distkey</th>
<th>sortkey</th>
</tr>
</thead>
<tbody>
<tr>
<td>eventid</td>
<td>integer</td>
<td>none</td>
<td>t</td>
<td>1</td>
</tr>
<tr>
<td>venueid</td>
<td>smallint</td>
<td>none</td>
<td>f</td>
<td>0</td>
</tr>
<tr>
<td>dateid</td>
<td>smallint</td>
<td>none</td>
<td>f</td>
<td>2</td>
</tr>
<tr>
<td>eventname</td>
<td>character varying(200)</td>
<td>none</td>
<td>f</td>
<td>0</td>
</tr>
</tbody>
</table>

You could create exactly the same table by using column names for the distribution and sort keys. For example:

create table eventdistsort1
distkey (eventid)
sortkey (eventid, dateid)
as
select eventid, venueid, dateid, eventname
from event;

The following statement applies even distribution to the table but does not define an explicit sort key:

create table eventdisteven
disstyle even
as
select eventid, venueid, dateid, eventname
from event;

The table does not inherit the sort key from the EVENT table (EVENTID) because EVEN distribution is specified for the new table. The new table has no sort key and no distribution key.

select "column", type, encoding, distkey, sortkey
from pg_table_def where tablename = 'eventdisteven';

<table>
<thead>
<tr>
<th>column</th>
<th>type</th>
<th>encoding</th>
<th>distkey</th>
<th>sortkey</th>
</tr>
</thead>
<tbody>
<tr>
<td>eventid</td>
<td>integer</td>
<td>none</td>
<td>f</td>
<td>0</td>
</tr>
<tr>
<td>venueid</td>
<td>smallint</td>
<td>none</td>
<td>f</td>
<td>0</td>
</tr>
<tr>
<td>dateid</td>
<td>smallint</td>
<td>none</td>
<td>f</td>
<td>0</td>
</tr>
<tr>
<td>eventname</td>
<td>character varying(200)</td>
<td>none</td>
<td>f</td>
<td>0</td>
</tr>
</tbody>
</table>

The following statement applies even distribution and defines a sort key:

create table eventdistevensort
disstyle even sortkey (venueid)
as
select eventid, venueid, dateid, eventname
from event;

The resulting table has a sort key but no distribution key.
The following statement redistributes the EVENT table on a different key column from the incoming data, which is sorted on the EVENTID column, and defines no SORTKEY column; therefore the table is not sorted.

```
create table venuedistevent distkey(venueid)
as select * from event;
```

Result

```
select "column", type, encoding, distkey, sortkey
from pg_table_def where tablename = 'venuedistevent';
```

| column    | type               | encoding | distkey | sortkey |
|-----------+-------------------+----------+---------+--------|
| eventid   | integer           | none     | f       | 0      |
| venueid   | smallint          | none     | f       | 1      |
| catid     | smallint          | none     | f       | 0      |
| dateid    | smallint          | none     | f       | 0      |
| eventname | character varying | none     | f       | 0      |
| startdate | timestamp without |          |         |        |

**CREATE USER**

Creates a new database user account. You must be a database superuser to execute this command.

**Synopsis**

```
CREATE USER name
[ [ WITH] option [ ... ] ]
```

where option can be:

CREATEDB | NOCREATEDB
| CREATEUSER | NOCREATEUSER
| IN GROUP groupname [, ... ]
| PASSWORD 'password'
| VALID UNTIL 'abstime'
Parameters

name
The name of the user account to create. For more information about valid names, see Names and identifiers (p. 213).

CREATEDB | NOCREATEDB
The CREATEDB option allows the new user account to create databases. The default is NOCREATEDB.

CREATEUSER | NOCREATEUSER
The CREATEUSER option creates a superuser with all database privileges, including CREATE USER. The default is NOCREATEUSER. For more information, see Superusers (p. 84).

IN GROUP groupname
Specifies the name of an existing group that the user belongs to. Multiple group names may be listed.

PASSWORD password
Sets the user's password. The user account password can be changed with the ALTER USER command.

Constraints:
• 8 to 64 characters in length.
• Must contain at least one uppercase letter, one lowercase letter, and one number.
• Can use any printable ASCII characters (ASCII code 33 to 126) except ' (single quote), " (double quote), \, /, @, or space.

VALID UNTIL abstime
The VALID UNTIL option sets an absolute time after which the user account password is no longer valid. By default the password has no time limit.

Usage Notes
By default, all users have CREATE and USAGE privileges on the PUBLIC schema. To disallow users from creating objects in the PUBLIC schema of a database, use the REVOKE command to remove that privilege.

Examples
The following command creates a user account named danny, with the password "abcD1234":

create user danny with password 'abcD1234';

The following command creates a user named danny who can create databases:

create user danny with password 'abcD1234' createdb;

In this example, the account password is valid until June 10, 2014:

create user danny with password 'abcD1234' valid until '2014-06-10';

The following example creates a user with a case-sensitive password that contains special characters:

create user newman with password '@AbC4321!';
CREATE VIEW

Creates a view in a database. The view is not physically materialized; the query that defines the view is run every time the view is referenced in a query.

Synopsis

```
CREATE [ OR REPLACE ] VIEW name [ ( column_name [, ...] ) ] AS query
```

Parameters

**OR REPLACE**

If a view of the same name already exists, the view is replaced. You can only replace a view with a new query that generates the identical set of columns, using the same column names and data types.

**name**

The name of the view. If a schema name is given (such as `myschema.myview`) the view is created using the specified schema. Otherwise, the view is created in the current schema. The view name must be different from the name of any other view or table in the same schema. For more information about valid names, see Names and identifiers (p. 213). You cannot create tables or views in the system databases `template0`, `template1`, and `padb_harvest`.

**column_name**

Optional list of names to be used for the columns in the view. If no column names are given, the column names are derived from the query.

**query**

A query (in the form of a SELECT statement) that evaluates to a table. This table defines the columns and rows in the view.

Note

You cannot update, insert into, or delete from a view.

Usage notes

Having ownership of a view, or having privileges granted on a view, does not imply access to the underlying tables. You need to grant access to the underlying tables explicitly.

Examples

The following command creates a view called `myevent` from a table called `EVENT`:

```
create view myevent as select eventname from event
where eventname = 'LeAnn Rimes';
```

The following command creates a view called `myuser` from a table called `USERS`:

```
create view myuser as select lastname from users;
```

DEALLOCATE

Deallocates a prepared statement.
Synopsis

DEALLOCATE [PREPARE] plan_name

Parameters

PREPARE
  This keyword is optional and is ignored.

plan_name
  The name of the prepared statement to deallocate.

Usage Notes

DEALLOCATE is used to deallocate a previously prepared SQL statement. If you do not explicitly deallocate a prepared statement, it is deallocated when the current session ends. For more information on prepared statements, see PREPARE (p. 354).

See Also

EXECUTE (p. 339), PREPARE (p. 354)

DECLARE

Defines a new cursor. Use a cursor to retrieve a few rows at a time from the result set of a larger query.

When the first row of a cursor is fetched, the entire result set is materialized on the leader node, in memory or on disk, if needed. Because of the potential negative performance impact of using cursors with large result sets, we recommend using alternative approaches whenever possible. For more information, see Performance considerations when using cursors (p. 330).

You must declare a cursor within a transaction block. Only one cursor at a time can be open per session.

For more information, see FETCH (p. 344), CLOSE (p. 273).

Synopsis

DECLARE cursor_name CURSOR FOR query

Parameters

cursor_name
  Name of the new cursor.

query
  A SELECT statement that populates the cursor.

DECLARE CURSOR Usage Notes

If your client application uses an ODBC connection and your query creates a result set that is too large to fit in memory, you can stream the result set to your client application by using a cursor. When you use a cursor, the entire result set is materialized on the leader node, and then your client can fetch the results incrementally.
Note
To enable cursors in ODBC for Microsoft Windows, enable the **Use Declare/Fetch** option in the ODBC DSN you use for Amazon Redshift. We recommend setting the ODBC cache size, using the **Cache Size** field in the ODBC DSN options dialog, to 4,000 or greater on multi-node clusters to minimize round trips. On a single-node cluster, set Cache Size to 1,000.

Because of the potential negative performance impact of using cursors, we recommend using alternative approaches whenever possible. For more information, see [Performance considerations when using cursors](p. 330).

Amazon Redshift cursors are supported with the following limitations:

- Only one cursor at a time can be open per session.
- Cursors must be used within a transaction (BEGIN … END).
- The maximum cumulative result set size for all cursors is constrained based on the cluster node type. If you need more cursors, you can reduce the maximum cursor size. If you need larger result sets, you can resize to an XL or 8XL node configuration.

For more information, see [Cursor constraints](p. 329).

**Cursor constraints**

When the first row of a cursor is fetched, the entire result set is materialized on the leader node. If the result set does not fit in memory, it is written to disk as needed. To protect the integrity of the leader node, Amazon Redshift enforces constraints on the size of all cursor result sets, based on the cluster's node type:

The maximum cursor size is user-configurable by setting the `max_cursor_result_set_size` parameter. The maximum cluster result set size is based on the cluster's node type and is not configurable.

To calculate the maximum number of concurrent cursors available, divide the maximum cumulative cluster result size by the maximum cursor size.

The following table shows the maximum total result set size for each cluster node type, the default maximum cursor result set, and the number of concurrent cursors based on the default maximum cursor size. Result set sizes are in megabytes.

<table>
<thead>
<tr>
<th>Node type</th>
<th>Maximum result set per cluster</th>
<th>Maximum result set per cursor (default)</th>
<th>Concurrent cursors</th>
</tr>
</thead>
<tbody>
<tr>
<td>DW1 XL single node</td>
<td>64000</td>
<td>32000</td>
<td>2</td>
</tr>
<tr>
<td>DW1 XL multiple nodes</td>
<td>1800000</td>
<td>450000</td>
<td>4</td>
</tr>
<tr>
<td>DW1 8XL multiple nodes</td>
<td>14400000</td>
<td>960000</td>
<td>15</td>
</tr>
<tr>
<td>DW2 Large single node</td>
<td>16000</td>
<td>16000</td>
<td>1</td>
</tr>
<tr>
<td>DW2 Large multiple nodes</td>
<td>384000</td>
<td>192000</td>
<td>2</td>
</tr>
<tr>
<td>DW2 8XL multiple nodes</td>
<td>3000000</td>
<td>750000</td>
<td>4</td>
</tr>
</tbody>
</table>

To enable more concurrent cursors, reduce the maximum result set per cursor by setting the `max_cursor_result_set_size` parameter. For example, if the maximum cumulative result set size is 64000 (64 GB) and `max_cursor_result_set_size` is set to 8000 (8 GB), the maximum number of...
concurrent cursors is 8 (64 GB / 8 GB). For more information, see Configure Maximum Size of a Cursor Result Set in the Amazon Redshift Cluster Management Guide.

You cannot have more concurrent cursors than concurrent queries, so Amazon Redshift limits the total number of concurrent cursors to 50, which is the maximum query concurrency level. Amazon Redshift adjusts the value of `max_cursor_result_set_size` as needed to maintain a limit of 50 concurrent cursors.

To view the active cursor configuration for a cluster, query the `STV_CURSOR_CONFIGURATION` system table as a superuser. To view the state of active cursors, query the `STV_ACTIVE_CURSORS` system table. Only the rows for a user's own cursors are visible to the user, but a superuser can view all cursors.

**Performance considerations when using cursors**

Because cursors materialize the entire result set on the leader node before beginning to return results to the client, using cursors with very large result sets can have a negative impact on performance. We strongly recommend against using cursors with very large result sets. In some cases, such as when your application uses an ODBC connection, cursors might be the only feasible solution. If possible, we recommend using these alternatives:

- Use `UNLOAD` to export a large table. When you use `UNLOAD`, the compute nodes work in parallel to transfer the data directly to data files on Amazon S3. For more information, see Unloading Data (p. 173).
- Set the JDBC fetch size parameter in your client application. If you use a JDBC connection and you are encountering client-side out-of-memory errors, you can enable your client to retrieve result sets in smaller batches by setting the JDBC fetch size parameter. For more information, see Setting the JDBC fetch size parameter (p. 196).

**DECLARE CURSOR Example**

The following example declares a cursor named LOLLAPALOOZA to select sales information for the Lollapalooza event, and then fetches rows from the result set using the cursor:

```sql
-- Begin a transaction
begin;

-- Declare a cursor
declare lollapalooza cursor for
select eventname, starttime, pricepaid/qtysold as costperticket, qtysold from sales, event
where sales.eventid = event.eventid
and eventname='Lollapalooza';

-- Fetch the first 5 rows in the cursor lollapalooza:
fetch forward 5 from lollapalooza;

eventname | starttime | costperticket | qtysold
-------------+-----------+---------------+---------
Lollapalooza | 2008-05-01 19:00:00 | 92.000000000 | 3
Lollapalooza | 2008-11-15 15:00:00 | 222.000000000 | 2
Lollapalooza | 2008-04-17 15:00:00 | 239.000000000 | 3
Lollapalooza | 2008-04-17 15:00:00 | 239.000000000 | 4
```
DELETE

Deletes rows from tables.

Note
The maximum size for a single SQL statement is 16 MB.

Synopsis

```
DELETE [ FROM ] table_name
[ {USING } table_name, ... ]
[ WHERE condition ]
```

Parameters

FROM
The FROM keyword is optional, except when the USING clause is specified. The statements `delete from event;` and `delete event;` are equivalent operations that remove all of the rows from the EVENT table.

Note
To delete all the rows from a table, TRUNCATE (p. 394) the table. TRUNCATE is much more efficient than DELETE and does not require a VACUUM and ANALYZE. However, be aware that TRUNCATE commits the transaction in which it is run.

table_name
A temporary or persistent table. Only the owner of the table or a user with DELETE privilege on the table may delete rows from the table.

Consider using the TRUNCATE command for fast unqualified delete operations on large tables; see TRUNCATE (p. 394).

Note
After deleting a large number of rows from a table:

- Vacuum the table to reclaim storage space and resort rows.
- Analyze the table to update statistics for the query planner.

USING table_name, ...
The USING keyword is used to introduce a table list when additional tables are referenced in the WHERE clause condition. For example, the following statement deletes all of the rows from the
EVENT table that satisfy the join condition over the EVENT and SALES tables. The SALES table must be explicitly named in the FROM list:

```sql
delete from event using sales where event.eventid=sales.eventid;
```

If you repeat the target table name in the USING clause, the DELETE operation runs a self-join. You can use a subquery in the WHERE clause instead of the USING syntax as an alternative way to write the same query.

**WHERE condition**
Optional clause that limits the deletion of rows to those that match the condition. For example, the condition can be a restriction on a column, a join condition, or a condition based on the result of a query. The query can reference tables other than the target of the DELETE command. For example:

```sql
delete from t1
    where col1 in(select col2 from t2);
```

If no condition is specified, all of the rows in the table are deleted.

## Examples

Delete all of the rows from the CATEGORY table:

```sql
delete from category;
```

Delete rows with CATID values between 0 and 9 from the CATEGORY table:

```sql
delete from category
    where catid between 0 and 9;
```

Delete rows from the LISTING table whose SELLERID values do not exist in the SALES table:

```sql
delete from listing
    where listing.sellerid not in(select sales.sellerid from sales);
```

The following two queries both delete one row from the CATEGORY table, based on a join to the EVENT table and an additional restriction on the CATID column:

```sql
delete from category
    using event
    where event.catid=category.catid and category.catid=9;
```

```sql
delete from category
    where catid in
        (select category.catid from category, event
         where category.catid=event.catid and category.catid=9);
```

## DROP DATABASE

Drops a database.
Synopsis

DROP DATABASE database_name [ FORCE ]

Parameters

database_name
  Name of the database to be dropped. You cannot drop a database if you are currently connected to it.
FORCE
  Option used to drop the DEV database in order to allow a restore of that database. You cannot drop the DEV database without using the FORCE option.

Caution
  Do not use this option unless you intend to restore the DEV database from a backup. In general, Amazon Redshift requires the DEV database to be in place, and certain operations depend on its existence.

Examples

The following example drops a database named TICKIT_TEST:

drop database tickit_test;

DROP GROUP

Deletes a user group. This command is not reversible. This command does not delete the individual users in a group.

See DROP USER to delete an individual user.

Synopsis

DROP GROUP name

Parameter

name
  Name of the user group to delete.

Example

The following example deletes the GUEST user group:

drop group guests;
DROP SCHEMA

Deletes a schema. This command is not reversible.

Synopsis

```
DROP SCHEMA name [, ...] [ CASCADE | RESTRICT ]
```

Parameters

- **name**
  
  Name of the schema to drop.

- **CASCADE**
  
  Automatically drops all objects in the schema, such as tables and functions.

- **RESTRICT**
  
  Do not drop the schema if it contains any objects. Default.

Example

The following example deletes a schema named S_SALES. This example has a safety mechanism so that the schema will not be deleted if it contains any objects. In this case, delete the schema objects before deleting the schema:

```
drop schema s_sales restrict;
```

The following example deletes a schema named S_SALES and all objects that are dependent on that schema:

```
drop schema s_sales cascade;
```

DROP TABLE

Removes a table from a database. Only the owner of a table can remove a table.

If you are trying to empty a table of rows, without removing the table, use the DELETE or TRUNCATE command.

DROP TABLE removes constraints that exist on the target table. Multiple tables can be removed with a single DROP TABLE command.

Synopsis

```
DROP TABLE name [, ...] [ CASCADE | RESTRICT ]
```

Parameters

- **name**
  
  The name of the table to drop.
CASCADE
Automatically drops objects that depend on the table, such as views.

RESTRICT
A table is not dropped if any objects depend on it. This is the default action.

Examples

Dropping a table with no dependencies
The following command set creates and drops a FEEDBACK table that has no dependencies:

```
create table feedback(a int);
drop table feedback;
```

If this table contained any columns that were references to other tables, Amazon Redshift would display a message advising you to use the CASCADE option to also drop dependent objects:

```
ERROR:  cannot drop table category because other objects depend on it
HINT:  Use DROP ... CASCADE to drop the dependent objects too.
```

Dropping two tables simultaneously
The following command set creates a FEEDBACK and BUYERS table and then drops both tables with a single command:

```
create table feedback(a int);
create table buyers(a int);
drop table feedback, buyers;
```

The following steps show how to drop a table called FEEDBACK using the CASCADE switch.

First, create a simple table called FEEDBACK using the CREATE TABLE command:

```
create table feedback(a int);
```

Dropping a table with a dependency

Next, create a view called FEEDBACK_VIEW using the CREATE VIEW command that relies on the table FEEDBACK:

```
create view feedback_view as select * from feedback;
```

The following command drops the table FEEDBACK and also drops the view FEEDBACK_VIEW, since FEEDBACK_VIEW is dependent on the table FEEDBACK:

```
drop table feedback cascade;
```

Viewing the dependencies for a table
You can create a view that holds the dependency information for all of the tables in a database. Query this view to determine if a given table has dependencies before dropping that table.

Type the following command to create a FIND_DEPEND view, which joins dependencies with object references:

```
select distinct c_p.oid as tbloid,
    n_p.nspname as schemaname, c_p.relname as name,
    n_c.nspname as refbyschemaname, c_c.relname as refbyname,
    c_c.oid as viewoid
from pg_catalog.pg_class c_p
join pg_catalog.pg_depend d_p
    on c_p.relfilenode = d_p.refobjid
join pg_catalog.pg_depend d_c
    on d_p.objid = d_c.objid
join pg_catalog.pg_class c_c
    on d_c.refobjid = c_c.relfilenode
left outer join pg_namespace n_p
    on c_p.relnamespace = n_p.oid
left outer join pg_namespace n_c
    on c_c.relnamespace = n_c.oid
where d_c.deptype = 'i'::"char"
    and c_c.relkind = 'v'::"char";
```

For this example, now create a SALES_VIEW from the SALES table:

```
create view sales_view as select * from sales;
```

Now query the FIND_DEPEND view to view dependencies in the database. Limit the scope of the query to the PUBLIC schema:

```
select * from find_depend
where refbyschemaname='public'
order by name;
```

This query returns the following dependencies, which shows that the SALES_VIEW would be also be dropped by using the CASCADE option when dropping the SALES table:

```
tbold | schemaname | name     | viewoid | refbyschemaname | refbyname
------|------------|----------|---------|----------------|-------------
100241 | public     | find_depend | 100241 | public         | find_depend
100203 | public     | sales     | 100245 | public         | sales_view
100245 | public     | sales_view | 100245 | public         | sales_view

(3 rows)
```

**DROP USER**

Drops a user from a database. Multiple users can be dropped with a single DROP USER command. You must be a database superuser to execute this command.
Synopsis

DROP USER name [, ... ]

Parameters

name

The name of the user account to remove. Multiple user accounts can be specified, separated by a comma between each user account name.

Notes

You cannot drop a user if the user owns any database object, such as a schema, database, table, or view, or if the user has any privileges on a table, database, or group. If you attempt to drop such a user, you will receive one of the following errors.

ERROR: user "username" cannot be dropped because the user owns some object [SQL State=55006]
ERROR: user "username" cannot be dropped because the user has a privilege on some object [SQL State=55006]

If a user owns an object, first drop the object or change its ownership to another user. If the user has privileges on an object, first revoke the privileges. The following example shows dropping an object, changing ownership, and revoking privileges before dropping the user.

drop database dwdatabase;
alter schema owner to dwadmin;
revoke all on table dwtable from dwuser;
drop user dwuser;

If a schema contains objects or if a table or view has dependencies, such as views, use the CASCADE keyword to drop the dependent objects as well.

drop table dwtable cascade;

Examples

To drop a user account called danny:

drop user danny;

To drop two users, called danny and billybob:

drop user danny, billybob;

DROP VIEW

Removes a view from the database. Multiple views can be dropped with a single DROP VIEW command. This command is not reversible.
Synopsis

DROP VIEW name [, ... ] [ CASCADE | RESTRICT ]

Parameters

name
The name of the view to be removed.

CASCADE
Automatically drops objects that depend on the view, such as other views.

RESTRICT
If any objects depend on the view, the view is not dropped. This is the default action.

Examples

This command drops the view called event:

drop view event;

To remove a view that has dependencies, use the CASCADE option. For example, say we have a table called EVENT and we create a view of the EVENT table called eventview.

We create another view based on the view eventview and call it myeventview.

The DROP VIEW command drops the myeventview view.

First, create the eventview view of the EVENT table, using the CREATE VIEW command:

```
create view eventview as
  select dateid, eventname, catid
  from event where catid = 1;
```

Now, a second view is created called myeventview, that is based on the first view eventview:

```
create view myeventview as
  select eventname, catid
  from eventview where eventname <> ' ';
```

At this point, two views have been created: eventview and myeventview.

The myeventview view is a child view with eventview as its parent.

For example, to delete the eventview view, the obvious command to use would be:

```
drop view eventview;
```

Notice that if you run this command, you will get the following error:
drop view eventview;
ERROR: cannot drop view eventview because other objects depend on it
HINT: Use DROP ... CASCADE to drop the dependent objects too.

To remedy this, enter the following command (as suggested in the error message):

drop view eventview cascade;

The command executes properly:

drop view myeventview cascade;

Both views have now been dropped successfully.

END

Commits the current transaction. Performs exactly the same function as the COMMIT command.

See COMMIT (p. 275) for more detailed documentation.

Synopsis

END [ WORK | TRANSACTION ]

Parameters

WORK
Optional keyword.

TRANSACTION
Optional keyword; WORK and TRANSACTION are synonyms.

Examples

The following examples all end the transaction block and commit the transaction:

end;

end work;

end transaction;

After any of these commands, Amazon Redshift ends the transaction block and commits the changes.

EXECUTE

Executes a previously prepared statement.
Synopsis

EXECUTE plan_name [ {parameter [, ...]} ]

Parameters

plan_name
   Name of the prepared statement to be executed.

parameter
   The actual value of a parameter to the prepared statement. This must be an expression yielding a value of a type compatible with the data type specified for this parameter position in the PREPARE command that created the prepared statement.

Usage Notes

EXECUTE is used to execute a previously prepared statement. Since prepared statements only exist for the duration of a session, the prepared statement must have been created by a PREPARE statement executed earlier in the current session.

If the previous PREPARE statement specified some parameters, a compatible set of parameters must be passed to the EXECUTE statement, or else Amazon Redshift will return an error. Unlike functions, prepared statements are not overloaded based on the type or number of specified parameters; the name of a prepared statement must be unique within a database session.

When an EXECUTE command is issued for the prepared statement, Amazon Redshift may optionally revise the query execution plan (to improve performance based on the specified parameter values) before executing the prepared statement. Also, for each new execution of a prepared statement, Amazon Redshift may revise the query execution plan again based on the different parameter values specified with the EXECUTE statement. To examine the query execution plan that Amazon Redshift has chosen for any given EXECUTE statements, use the EXPLAIN (p. 340) command.

For examples and more information on the creation and usage of prepared statements, see PREPARE (p. 354).

See Also

DEALLOCATE (p. 327), PREPARE (p. 354)

EXPLAIN

Displays the execution plan for a query statement without running the query.

Synopsis

EXPLAIN [ VERBOSE ] query

Parameters

VERBOSE
   Displays the full query plan instead of just a summary.
query

Query statement to explain. The query can be a SELECT, INSERT, CREATE TABLE AS, UPDATE, or DELETE statement.

Usage notes

EXPLAIN performance is sometimes influenced by the time it takes to create temporary tables. For example, a query that uses the common subexpression optimization requires temporary tables to be created and analyzed in order to return the EXPLAIN output. The query plan depends on the schema and statistics of the temporary tables. Therefore, the EXPLAIN command for this type of query might take longer to run than expected.

Query planning and execution steps

The execution plan for a specific Amazon Redshift query statement breaks down execution and calculation of a query into a discrete sequence of steps and table operations that will eventually produce a final result set for the query. The following table provides a summary of steps that Amazon Redshift can use in developing an execution plan for any query a user submits for execution.

<table>
<thead>
<tr>
<th>EXPLAIN Operators</th>
<th>Query Execution Steps</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCAN:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sequential Scan</td>
<td>scan</td>
<td>Amazon Redshift relation scan or table scan operator or step. Scans whole table sequentially from beginning to end; also evaluates query constraints for every row (Filter) if specified with WHERE clause. Also used to run INSERT, UPDATE, and DELETE statements.</td>
</tr>
<tr>
<td>JOINS:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nested Loop</td>
<td>nloop</td>
<td>Least optimal join; mainly used for cross-joins (Cartesian products; without a join condition) and some inequality joins.</td>
</tr>
<tr>
<td>Hash Join</td>
<td>hjoin</td>
<td>Also used for inner joins and left and right outer joins and typically faster than a nested loop join. Hash Join reads the outer table, hashes the joining column, and finds matches in the inner hash table. Step can spill to disk. (Inner input of hjoin is hash step which can be disk-based.)</td>
</tr>
<tr>
<td>Merge Join</td>
<td>mjoin</td>
<td>Also used for inner joins and outer joins (for join tables that are both distributed and sorted on the joining columns). Typically the fastest Amazon Redshift join algorithm, not including other cost considerations.</td>
</tr>
<tr>
<td>AGGREGATION:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Aggregate</td>
<td>aggr</td>
<td>Operator/step for scalar aggregate functions.</td>
</tr>
<tr>
<td>EXPLAIN Operators</td>
<td>Query Execution Steps</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-----------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>HashAggregate</td>
<td>aggr</td>
<td>Operator/step for grouped aggregate functions. Can operate from disk by virtue of hash table spilling to disk.</td>
</tr>
<tr>
<td>GroupAggregate</td>
<td>aggr</td>
<td>Operator sometimes chosen for grouped aggregate queries if the Amazon Redshift configuration setting for force_hash_grouping setting is off.</td>
</tr>
</tbody>
</table>

**SORT:** Operators and steps used when queries have to sort or merge result sets.

<table>
<thead>
<tr>
<th>Sort</th>
<th>sort</th>
<th>Sort performs the sorting specified by the ORDER BY clause as well as other operations such as UNIONs and joins. Can operate from disk.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Merge</td>
<td>merge</td>
<td>Produces final sorted results of a query based on intermediate sorted results derived from operations performed in parallel.</td>
</tr>
</tbody>
</table>

**EXCEPT, INTERCEPT, and UNION operations:**

<table>
<thead>
<tr>
<th>SetOp Except [Distinct]</th>
<th>hjoin</th>
<th>Used for EXCEPT queries. Can operate from disk based on virtue of fact that input hash can be disk-based.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hash Intersect [Distinct]</td>
<td>hjoin</td>
<td>Used for INTERSECT queries. Can operate from disk based on virtue of fact that input hash can be disk-based.</td>
</tr>
<tr>
<td>Append [All</td>
<td>Distinct]</td>
<td>save</td>
</tr>
</tbody>
</table>

**Miscellaneous/Other:**

<table>
<thead>
<tr>
<th>Hash</th>
<th>hash</th>
<th>Used for inner joins and left and right outer joins (provides input to a hash join). The Hash operator creates the hash table for the inner table of a join. (The inner table is the table that is checked for matches and, in a join of two tables, is usually the smaller of the two.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Limit</td>
<td>limit</td>
<td>Evaluates the LIMIT clause.</td>
</tr>
<tr>
<td>Materialize</td>
<td>save</td>
<td>Materialize rows for input to nested loop joins and some merge joins. Can operate from disk.</td>
</tr>
<tr>
<td>--</td>
<td>parse</td>
<td>Used to parse textual input data during a load.</td>
</tr>
<tr>
<td>--</td>
<td>project</td>
<td>Used to rearrange columns and compute expressions, that is, project data.</td>
</tr>
<tr>
<td>Result</td>
<td>--</td>
<td>Run scalar functions that do not involve any table access.</td>
</tr>
<tr>
<td>--</td>
<td>return</td>
<td>Return rows to the leader or client.</td>
</tr>
<tr>
<td>Subplan</td>
<td>--</td>
<td>Used for certain subqueries.</td>
</tr>
</tbody>
</table>
EXPLAIN Operators | Query Execution Steps | Description
---|---|---
Unique | unique | Eliminates duplicates from SELECT DISTINCT and UNION queries.
Window | window | Compute aggregate and ranking window functions. Can operate from disk.

Network Operations:
- **Network (Broadcast)**: `bcast` | Broadcast is also an attribute of Join Explain operators and steps.
- **Network (Distribute)**: `dist` | Distribute rows to compute nodes for parallel processing by data warehouse cluster.
- **Network (Send to Leader)**: `return` | Sends results back to the leader for further processing.

DML Operations (operators that modify data):
- **Insert (using Result)**: `insert` | Inserts data.
- **Delete (Scan + Filter)**: `delete` | Deletes data. Can operate from disk.
- **Update (Scan + Filter)**: `delete, insert` | Implemented as delete and Insert.

Examples

**Note**
For these examples, the sample output might vary depending on Amazon Redshift configuration.

The following example returns the query plan for a query that selects the EVENTID, EVENTNAME, VENUEID, and VENUENAME from the EVENT and VENUE tables:

```sql
explain
select eventid, eventname, event.venueid, venuename
from event, venue
where event.venueid = venue.venueid;
```

```
QUERY PLAN
-------------------------------------------------------------------------
XN Hash Join DS_DIST_OUTER  (cost=2.52..58653620.93 rows=8712 width=43)
Hash Cond: ("outer".venueid = "inner".venueid)
  ->  XN Seq Scan on event  (cost=0.00..87.98 rows=8798 width=23)
  ->  XN Hash  (cost=2.02..2.02 rows=202 width=22)
  ->  XN Seq Scan on venue  (cost=0.00..2.02 rows=202 width=22)
(5 rows)
```

The following example returns the query plan for the same query with verbose output:

```sql
explain verbose
select eventid, eventname, event.venueid, venuename
from event, venue
where event.venueid = venue.venueid;
```
The following example returns the query plan for a CREATE TABLE AS (CTAS) statement:

```
explain create table venue_nonulls as
  select * from venue
  where venueseats is not null;
```

```
-----------------------------------------------------------
XN Seq Scan on venue  (cost=0.00..2.02 rows=187 width=45)
  Filter: (venueseats IS NOT NULL)
(2 rows)
```

**FETCH**

Retrieves rows using a cursor. For information about declaring a cursor, see DECLARE (p. 328).

FETCH retrieves rows based on the current position within the cursor. When a cursor is created, it is positioned before the first row. After a FETCH, the cursor is positioned on the last row retrieved. If FETCH runs off the end of the available rows, such as following a FETCH ALL, the cursor is left positioned after the last row.

FORWARD 0 fetches the current row without moving the cursor; that is, it fetches the most recently fetched row. If the cursor is positioned before the first row or after the last row, no row is returned.

When the first row of a cursor is fetched, the entire result set is materialized on the leader node, in memory or on disk, if needed. Because of the potential negative performance impact of using cursors with large result sets, we recommend using alternative approaches whenever possible. For more information, see Performance considerations when using cursors.

For more information, see DECLARE (p. 328), CLOSE (p. 273).
## Synopsis

```
FETCH [ NEXT | ALL | {FORWARD [ count | ALL ] } ] FROM cursor
```

## Parameters

**NEXT**  
Fetches the next row. This is the default.

**ALL**  
Fetches all remaining rows. (Same as FORWARD ALL.) **ALL** is not supported for single-node clusters.

**FORWARD [ count | ALL ]**  
Fetches the next `count` rows, or all remaining rows. **FORWARD 0** fetches the current row. For single-node clusters, the maximum value for `count` is **1000**. **FORWARD ALL** is not supported for single-node clusters.

**cursor**  
Name of the new cursor.

## FETCH Example

The following example declares a cursor named **LOLLAPALOOZA** to select sales information for the Lollapalooza event, and then fetches rows from the result set using the cursor:

```sql
-- Begin a transaction
begin;

-- Declare a cursor
declare lollapalooza cursor for
select eventname, starttime, pricepaid/qtysold as costperticket, qtysold
from sales, event
where sales.eventid = event.eventid
and eventname='Lollapalooza';

-- Fetch the first 5 rows in the cursor lollapalooza:
fetch forward 5 from lollapalooza;

<table>
<thead>
<tr>
<th>eventname</th>
<th>starttime</th>
<th>costperticket</th>
<th>qtysold</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lollapalooza</td>
<td>2008-05-01 19:00:00</td>
<td>92.0000000000</td>
<td>3</td>
</tr>
<tr>
<td>Lollapalooza</td>
<td>2008-11-15 15:00:00</td>
<td>222.0000000000</td>
<td>2</td>
</tr>
<tr>
<td>Lollapalooza</td>
<td>2008-04-17 15:00:00</td>
<td>239.0000000000</td>
<td>3</td>
</tr>
<tr>
<td>Lollapalooza</td>
<td>2008-04-17 15:00:00</td>
<td>239.0000000000</td>
<td>4</td>
</tr>
<tr>
<td>Lollapalooza</td>
<td>2008-04-17 15:00:00</td>
<td>239.0000000000</td>
<td>1</td>
</tr>
</tbody>
</table>

(5 rows)

-- Fetch the next row:
fetch next from lollapalooza;

<table>
<thead>
<tr>
<th>eventname</th>
<th>starttime</th>
<th>costperticket</th>
<th>qtysold</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lollapalooza</td>
<td>2008-05-01 19:00:00</td>
<td>92.0000000000</td>
<td>3</td>
</tr>
<tr>
<td>Lollapalooza</td>
<td>2008-11-15 15:00:00</td>
<td>222.0000000000</td>
<td>2</td>
</tr>
<tr>
<td>Lollapalooza</td>
<td>2008-04-17 15:00:00</td>
<td>239.0000000000</td>
<td>3</td>
</tr>
<tr>
<td>Lollapalooza</td>
<td>2008-04-17 15:00:00</td>
<td>239.0000000000</td>
<td>4</td>
</tr>
<tr>
<td>Lollapalooza</td>
<td>2008-04-17 15:00:00</td>
<td>239.0000000000</td>
<td>1</td>
</tr>
</tbody>
</table>

(5 rows)
GRANT

Defines access privileges for a user or user group.

Privileges include access options such as reading data in tables and views, writing data, and creating
tables. Use this command to give specific privileges on a table, database, or schema. To revoke privileges
from a database object, use the REVOKE (p. 356) command.

Note
Superusers can access all objects regardless of GRANT and REVOKE commands that set object
privileges.

Synopsis

GRANT { { SELECT | INSERT | UPDATE | DELETE | REFERENCES } [, ...] | ALL [ PRIVILEGES ] } 
ON { { TABLE } table_name [, ...] 
TO { { username | GROUP group_name | PUBLIC } [, ...] 
[ WITH GRANT OPTION ] 

GRANT { { CREATE | TEMPORARY | TEMP } [, ...] | ALL [ PRIVILEGES ] } 
ON DATABASE db_name [, ...] 
TO { { username | GROUP group_name | PUBLIC } [, ...] 
[ WITH GRANT OPTION ] 

GRANT { { CREATE | USAGE } [, ...] | ALL [ PRIVILEGES ] } 
ON SCHEMA schema_name [, ...] 
TO { { username | GROUP group_name | PUBLIC } [, ...] 
[ WITH GRANT OPTION ]

Parameters

SELECT
Grants privilege to select data from a table or view using a SELECT statement. The SELECT privilege
is also required to reference existing column values for UPDATE or DELETE operations.

INSERT
Grants privilege to load data into a table using an INSERT statement or a COPY statement.

UPDATE
Grants privilege to update a table column using an UPDATE statement. (UPDATE operations also
require the SELECT privilege, since they must reference table columns to determine which rows to
update, or to compute new values for columns.)

DELETE
Grants privilege to delete a data row from a table. (DELETE operations also require the SELECT
privilege, since they must reference table columns to determine which rows to delete.)

REFERENCES
Grants privilege to create a foreign key constraint. You need to grant this privilege on both the
referenced table and the referencing table; otherwise the user will not be able to create the constraint.
ALL [ PRIVILEGES ]
   Grants all available privileges at once to the specified user or user group. The PRIVILEGES keyword is optional.

ON [ TABLE ] table_name
   Grants the specified privileges on a table or a view. The TABLE keyword is optional. You can list multiple tables and views in one statement.

TO username
   User receiving the privileges.

GROUP group_name
   Grants the privileges to a user group.

PUBLIC
   Grants the specified privileges to all users, including users created later. PUBLIC represents a group that always includes all users. An individual user's privileges consist of the sum of privileges granted to PUBLIC, privileges granted to any groups that the user belongs to, and any privileges granted to the user individually.

WITH GRANT OPTION
   Grants the specified privileges to others.

CREATE
   Depending on the database object, grants the following privileges to the user or user group:
   • Databases: Allows users to create schemas within the database.
   • Schemas: Allows users to create objects within a schema. To rename an object, the user must have the CREATE privilege and own the object to be renamed.

TEMPORARY | TEMP
   Grants the privilege to create temporary tables in the specified database.

   Note
   By default, users are granted permission to create temporary tables by their automatic membership in the PUBLIC group. To remove the privilege for any users to create temporary tables, revoke the TEMP permission from the PUBLIC group and then explicitly grant the permission to create temporary tables to specific users or groups of users.

ON DATABASE db_name
   Grants the privileges on a database.

USAGE
   Grants USAGE privilege on a specific schema, which makes objects in that schema accessible to users. Specific actions on these objects must be granted separately (for example, SELECT or UPDATE privileges on tables). By default, all users have CREATE and USAGE privileges on the PUBLIC schema.

ON SCHEMA schema_name
   Grants the privileges on a schema.

Usage notes

Having ownership of a view, or having privileges granted on a view, does not imply privileges on the underlying tables. Similarly, having privileges granted on a schema does not imply privileges on the tables in the schema. You need to grant access to the underlying tables explicitly.

Examples

The following example grants the SELECT privilege on the SALES table to the user fred.

grant select on table sales to fred;
The following example grants all schema privileges on the schema QA_TICKIT to the user group QA_USERS. Schema privileges are CREATE and USAGE. Usage grants users access to the objects in the schema, but does not grant privileges, such as INSERT or SELECT, on those objects. Privileges must be granted on each object separately.

```sql
create group qa_users;
grant all on schema qa_tickit to group qa_users;
```

The following example grants all privileges on the SALES table in the QA_TICKIT schema to all users in the group QA_USERS.

```sql
grant all on table qa_tickit.sales to group qa_users;
```

The following sequence of commands shows how access to a schema does not grant privileges on a table in the schema.

```sql
create user schema_user in group qa_users password 'Abcd1234';
create schema qa_tickit;
create table qa_tickit.test (col1 int);
grant all on schema qa_tickit to schema_user;

set session authorization schema_user;
select current_user;
current_user
------------
schema_user
(1 row)

select count(*) from qa_tickit.test;
ERROR: permission denied for relation test [SQL State=42501]

set session authorization dw_user;
grant select on table qa_tickit.test to schema_user;
set session authorization schema_user;
select count(*) from qa_tickit.test;

count
------
0
(1 row)
```

The following sequence of commands shows how access to a view does not imply access to its underlying tables. The user called VIEW_USER cannot select from the DATE table although this user has been granted all privileges on VIEW_DATE.

```sql
create user view_user password 'Abcd1234';
create view view_date as select * from date;
grant all on view_date to view_user;
set session authorization view_user;
select current_user;
current_user
------------
```

API Version 2012-12-01

348
view_user
(1 row)

select count(*) from view_date;
count
--------
365
(1 row)

select count(*) from date;
ERROR:  permission denied for relation date

**INSERT**

**Topics**
- Synopsis (p. 349)
- Parameters (p. 349)
- Usage notes (p. 350)
- INSERT examples (p. 351)

Inserts new rows into a table. You can insert a single row with the VALUES syntax, multiple rows with the VALUES syntax, or one or more rows defined by the results of a query (INSERT INTO...SELECT).

**Note**
We strongly encourage you to use the COPY (p. 276) command to load large amounts of data. Using individual INSERT statements to populate a table might be prohibitively slow. Alternatively, if your data already exists in other Amazon Redshift database tables, use INSERT INTO SELECT or CREATE TABLE AS (p. 320) to improve performance. For more information about using the COPY command to load tables, see Loading Data (p. 117).

**Note**
The maximum size for a single SQL statement is 16 MB.

**Synopsis**

```
INSERT INTO table_name [ ( column [, ...] ) ]
{DEFAULT VALUES |
VALUES ( { expression | DEFAULT } [, ...] )
[ , ( { expression | DEFAULT } [, ...] )
[ , ...] ] |
query }
```

**Parameters**

- `table_name`  
  A temporary or persistent table. Only the owner of the table or a user with INSERT privilege on the table can insert rows. If you use the `query` clause to insert rows, you must have SELECT privilege on the tables named in the query.

- `column`  
  You can insert values into one or more columns of the table. You can list the target column names in any order. If you do not specify a column list, the values to be inserted must correspond to the table columns in the order in which they were declared in the CREATE TABLE statement. If the
number of values to be inserted is less than the number of columns in the table, the first \( n \) columns are loaded.

Either the declared default value or a null value is loaded into any column that is not listed (implicitly or explicitly) in the INSERT statement.

**DEFAULT VALUES**

If the columns in the table were assigned default values when the table was created, use these keywords to insert a row that consists entirely of default values. If any of the columns do not have default values, nulls are inserted into those columns. If any of the columns are declared NOT NULL, the INSERT statement returns an error.

**VALUES**

Use this keyword to insert one or more rows, each row consisting of one or more values. The VALUES list for each row must align with the column list. To insert multiple rows, use a comma delimiter between each list of expressions. Do not repeat the VALUES keyword. All VALUES lists for a multiple-row INSERT statement must contain the same number of values.

**expression**

A single value or an expression that evaluates to a single value. Each value must be compatible with the data type of the column where it is being inserted. If possible, a value whose data type does not match the column's declared data type is automatically converted to a compatible data type. For example:

- A decimal value 1.1 is inserted into an INT column as 1.
- A decimal value 100.8976 is inserted into a DEC(5,2) column as 100.90.

You can explicitly convert a value to a compatible data type by including type cast syntax in the expression. For example, if column COL1 in table T1 is a CHAR(3) column:

```sql
insert into t1(col1) values('Incomplete'::char(3));
```

This statement inserts the value Inc into the column.

For a single-row INSERT VALUES statement, you can use a scalar subquery as an expression. The result of the subquery is inserted into the appropriate column.

**Note**

Subqueries are not supported as expressions for multiple-row INSERT VALUES statements.

**DEFAULT**

Use this keyword to insert the default value for a column, as defined when the table was created. If no default value exists for a column, a null is inserted. You cannot insert a default value into a column that has a NOT NULL constraint if that column does not have an explicit default value assigned to it in the CREATE TABLE statement.

**query**

Insert one or more rows into the table by defining any query. All of the rows that the query produces are inserted into the table. The query must return a column list that is compatible with the columns in the table, but the column names do not have to match.

**Usage notes**

**Note**

We strongly encourage you to use the COPY (p. 276) command to load large amounts of data. Using individual INSERT statements to populate a table might be prohibitively slow. Alternatively, if your data already exists in other Amazon Redshift database tables, use INSERT INTO SELECT or CREATE TABLE AS (p. 320) to improve performance. For more information about using the COPY command to load tables, see Loading Data (p. 117).
The data format for the inserted values must match the data format specified by the CREATE TABLE definition.

After inserting a large number of new rows into a table:

- Vacuum the table to reclaim storage space and resort rows.
- Analyze the table to update statistics for the query planner.

When values are inserted into DECIMAL columns and they exceed the specified scale, the loaded values are rounded up as appropriate. For example, when a value of 20.259 is inserted into a DECIMAL(8,2) column, the value that is stored is 20.26.

**INSERT examples**

The CATEGORY table in the TICKIT database contains the following rows:

<table>
<thead>
<tr>
<th>catid</th>
<th>catgroup</th>
<th>catname</th>
<th>catdesc</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Sports</td>
<td>MLB</td>
<td>Major League Baseball</td>
</tr>
<tr>
<td>2</td>
<td>Sports</td>
<td>NHL</td>
<td>National Hockey League</td>
</tr>
<tr>
<td>3</td>
<td>Sports</td>
<td>NFL</td>
<td>National Football League</td>
</tr>
<tr>
<td>4</td>
<td>Sports</td>
<td>NBA</td>
<td>National Basketball Association</td>
</tr>
<tr>
<td>5</td>
<td>Sports</td>
<td>MLS</td>
<td>Major League Soccer</td>
</tr>
<tr>
<td>6</td>
<td>Shows</td>
<td>Musicals</td>
<td>Musical theatre</td>
</tr>
<tr>
<td>7</td>
<td>Shows</td>
<td>Plays</td>
<td>All non-musical theatre</td>
</tr>
<tr>
<td>8</td>
<td>Shows</td>
<td>Opera</td>
<td>All opera and light opera</td>
</tr>
<tr>
<td>9</td>
<td>Concerts</td>
<td>Pop</td>
<td>All rock and pop music concerts</td>
</tr>
<tr>
<td>10</td>
<td>Concerts</td>
<td>Jazz</td>
<td>All jazz singers and bands</td>
</tr>
<tr>
<td>11</td>
<td>Concerts</td>
<td>Classical</td>
<td>All symphony, concerto, and choir concerts</td>
</tr>
</tbody>
</table>

Create a CATEGORY_STAGE table with a similar schema to the CATEGORY table but define default values for the columns:

```sql
create table category_stage
    (catid smallint default 0,
    catgroup varchar(10) default 'General',
    catname varchar(10) default 'General',
    catdesc varchar(50) default 'General');
```

The following INSERT statement selects all of the rows from the CATEGORY table and inserts them into the CATEGORY_STAGE table.

```sql
insert into category_stage
    (select * from category);
```

The parentheses around the query are optional.

This command inserts a new row into the CATEGORY_STAGE table with a value specified for each column in order:

```sql
insert into category_stage values
    (12, 'Concerts', 'Comedy', 'All stand-up comedy performances');
```
You can also insert a new row that combines specific values and default values:

```sql
insert into category_stage values
(13, 'Concerts', 'Other', default);
```

Run the following query to return the inserted rows:

```sql
select * from category_stage
where catid in(12,13) order by 1;
```

<table>
<thead>
<tr>
<th>catid</th>
<th>catgroup</th>
<th>catname</th>
<th>catdesc</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>Concerts</td>
<td>Comedy</td>
<td>All stand-up comedy performances</td>
</tr>
<tr>
<td>13</td>
<td>Concerts</td>
<td>Other</td>
<td>General</td>
</tr>
</tbody>
</table>

(2 rows)

The following examples show some multiple-row INSERT VALUES statements. The first example inserts specific CATID values for two rows and default values for the other columns in both rows.

```sql
insert into category_stage values
(14, default, default, default),
(15, default, default, default);
```

```sql
select * from category_stage where catid in(14,15) order by 1;
```

<table>
<thead>
<tr>
<th>catid</th>
<th>catgroup</th>
<th>catname</th>
<th>catdesc</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>General</td>
<td>General</td>
<td>General</td>
</tr>
<tr>
<td>15</td>
<td>General</td>
<td>General</td>
<td>General</td>
</tr>
</tbody>
</table>

(2 rows)

The next example inserts three rows with various combinations of specific and default values:

```sql
insert into category_stage values
(default, default, default, default),
(20, default, 'Country', default),
(21, 'Concerts', 'Rock', default);
```

```sql
select * from category_stage where catid in(0,20,21) order by 1;
```

<table>
<thead>
<tr>
<th>catid</th>
<th>catgroup</th>
<th>catname</th>
<th>catdesc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>General</td>
<td>General</td>
<td>General</td>
</tr>
<tr>
<td>20</td>
<td>General</td>
<td>Country</td>
<td>General</td>
</tr>
<tr>
<td>21</td>
<td>Concerts</td>
<td>Rock</td>
<td>General</td>
</tr>
</tbody>
</table>

(3 rows)

The first set of VALUES in this example produce the same results as specifying DEFAULT VALUES for a single-row INSERT statement.

The following examples show INSERT behavior when a table has an IDENTITY column. First, create a new version of the CATEGORY table, then insert rows into it from CATEGORY:

```sql
create table category_ident
(catid int identity not null,
catgroup varchar(10) default 'General',
...);```
catname varchar(10) default 'General',
catdesc varchar(50) default 'General');

insert into category_ident(catgroup,catname,catdesc)
select catgroup,catname,catdesc from category;

Note that you cannot insert specific integer values into the CATID IDENTITY column. IDENTITY column
values are automatically generated.

The following example demonstrates that subqueries cannot be used as expressions in multiple-row
INSERT VALUES statements:

insert into category(catid) values
((select max(catid)+1 from category)),
((select max(catid)+2 from category));

ERROR: cannot use subqueries in multi-row VALUES

LOCK

Restricts access to a database table. This command is only meaningful when it is run inside a transaction
block.

The LOCK command obtains a table-level lock in “ACCESS EXCLUSIVE” mode, waiting if necessary for
any conflicting locks to be released. Explicitly locking a table in this way causes reads and writes on the
table to wait when they are attempted from other transactions or sessions. An explicit table lock created
by one user temporarily prevents another user from selecting data from that table or loading data into it.
The lock is released when the transaction that contains the LOCK command completes.

Less restrictive table locks are acquired implicitly by commands that refer to tables, such as write
operations. For example, if a user tries to read data from a table while another user is updating the table,
the data that is read will be a snapshot of the data that has already been committed. (In some cases,
queries will abort if they violate serializable isolation rules.) See Managing concurrent write
operations (p. 169).

Some DDL operations, such as DROP TABLE and TRUNCATE, create exclusive locks. These operations
prevent data reads.

If a lock conflict occurs, Amazon Redshift displays an error message to alert the user who started the
transaction in conflict. The transaction that received the lock conflict is aborted. Every time a lock conflict
occurs, Amazon Redshift writes an entry to the STL_TR_CONFLICT (p. 635) table.

Synopsis

LOCK [ TABLE ] table_name [, ...]

Parameters

TABLE
Optional keyword.

table_name
Name of the table to lock. You can lock more than one table by using a comma-delimited list of table
names. You cannot lock views.
Example

begin;
lock event, sales;
...

**PREPARE**

Prepare a statement for execution.

PREPARE creates a prepared statement. When the PREPARE statement is executed, the specified statement (SELECT, INSERT, UPDATE, or DELETE) is parsed, rewritten, and planned. When an EXECUTE command is then issued for the prepared statement, Amazon Redshift may optionally revise the query execution plan (to improve performance based on the specified parameter values) before executing the prepared statement.

**Synopsis**

PREPARE plan_name [ (datatype [, ...] ) ] AS statement

**Parameters**

- **plan_name**: An arbitrary name given to this particular prepared statement. It must be unique within a single session and is subsequently used to execute or deallocate a previously prepared statement.
- **datatype**: The data type of a parameter to the prepared statement. To refer to the parameters in the prepared statement itself, use $1, $2, and so on.
- **statement**: Any SELECT, INSERT, UPDATE, or DELETE statement.

**Usage Notes**

Prepared statements can take parameters: values that are substituted into the statement when it is executed. To include parameters in a prepared statement, supply a list of data types in the PREPARE statement, and, in the statement to be prepared itself, refer to the parameters by position using the notation $1, $2, ... When executing the statement, specify the actual values for these parameters in the EXECUTE statement. See EXECUTE (p. 339) for more details.

Prepared statements only last for the duration of the current session. When the session ends, the prepared statement is discarded, so it must be re-created before being used again. This also means that a single prepared statement cannot be used by multiple simultaneous database clients; however, each client can create its own prepared statement to use. The prepared statement can be manually removed using the DEALLOCATE command.

Prepared statements have the largest performance advantage when a single session is being used to execute a large number of similar statements. As mentioned, for each new execution of a prepared statement, Amazon Redshift may revise the query execution plan to improve performance based on the specified parameter values. To examine the query execution plan that Amazon Redshift has chosen for any specific EXECUTE statements, use the EXPLAIN (p. 340) command.
For more information on query planning and the statistics collected by Amazon Redshift for query optimization, see the ANALYZE (p. 268) command.

### Examples

Create a temporary table, prepare INSERT statement and then execute it:

| DROP TABLE temp1; |
| CREATE TABLE temp1 (c1 char(20), c2 char(20)); |
| PREPARE prep_insert_plan (char, char) AS insert into temp1 values ($1, $2); |
| EXECUTE prep_insert_plan (1, 'one'); |
| EXECUTE prep_insert_plan (2, 'two'); |
| EXECUTE prep_insert_plan (3, 'three'); |
| DEALLOCATE prep_insert_plan; |

Prepare a SELECT statement and then execute it:

| PREPARE prep_select_plan (char) AS select * from temp1 where c1 = $1; |
| EXECUTE prep_select_plan (2); |
| EXECUTE prep_select_plan (3); |
| DEALLOCATE prep_select_plan; |

### See Also

DEALLOCATE (p. 327), EXECUTE (p. 339)

## RESET

Restores the value of a configuration parameter to its default value.

You can reset either a single specified parameter or all parameters at once. To set a parameter to a specific value, use the SET (p. 389) command. To display the current value of a parameter, use the SHOW (p. 393) command.

### Synopsis

```
RESET { parameter_name | ALL }
```

### Parameters

* **parameter_name**
  
  Name of the parameter to reset. See Modifying the server configuration (p. 693) for more documentation about parameters.

* **ALL**
  
  Resets all run-time parameters.

### Examples

The following example resets the `query_group` parameter to its default value:
reset query_group;

The following example resets all run-time parameters to their default values:

reset all;

**REVOKE**

Removes access privileges, such as privileges to create or update tables, from a user or user group.

Specify in the REVOKE statement the privileges that you want to remove. Use the GRANT (p. 346) command to give privileges.

**Note**

Superusers, such as the *dwuser* user, can access all objects regardless of GRANT and REVOKE commands that set object privileges.

**Synopsis**

```sql
REVOKE [ GRANT OPTION FOR ]
{ { SELECT | INSERT | UPDATE | DELETE | REFERENCES } [, ... ] | ALL [ PRIVILEGES ] }
ON [ TABLE ] table_name [, ...]
FROM { username | GROUP group_name | PUBLIC } [, ...]
[ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
{ { CREATE | TEMPORARY | TEMP } [, ... ] | ALL [ PRIVILEGES ] }
ON DATABASE db_name [, ...]
FROM { username | GROUP group_name | PUBLIC } [, ...]
[ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
{ { CREATE | USAGE } [, ... ] | ALL [ PRIVILEGES ] }
ON SCHEMA schema_name [, ...]
FROM { username | GROUP group_name | PUBLIC } [, ...]
[ CASCADE | RESTRICT ]
```

**Parameters**

**GRANT OPTION FOR**

If specified, revokes only the grant option for the privilege, not the privilege itself.

**SELECT**

Revokes privilege to select data from a table or a view using a SELECT statement.

**INSERT**

Revokes privilege to load data into a table using an INSERT statement or a COPY statement.

**UPDATE**

Revokes privilege to update a table column using an UPDATE statement.

**DELETE**

Revokes privilege to delete a data row from a table.

**REFERENCES**

Revokes privilege to create a foreign key constraint. You should revoke this privilege on both the referenced table and the referencing table.
ALL [ PRIVILEGES ]
Revokes all available privileges at once from the specified user or group. The PRIVILEGES keyword is optional.

ON [ TABLE ] table_name
Revokes the specified privileges on a table or a view. The TABLE keyword is optional.

GROUP group_name
Revokes the privileges from a user group.

PUBLIC
Revokes the specified privileges from all users. PUBLIC represents a group that always includes all users. An individual user’s privileges consist of the sum of privileges granted to PUBLIC; privileges granted to any groups that the user belongs to; and any privileges granted to the user individually.

WITH GRANT OPTION
Allows users to grant the specified privileges to others.

CREATE
Depending on the database object, revokes the following privileges from the user or group:
• Databases: prevents users from creating schemas within the database.
• Schemas: prevents users from creating objects within a schema. To rename an object, the user must have the CREATE privilege and own the object to be renamed.

Note
By default, all users have CREATE and USAGE privileges on the PUBLIC schema.

TEMPORARY | TEMP
Revokes the privilege to create temporary tables in the specified database.

Note
By default, users are granted permission to create temporary tables by their automatic membership in the PUBLIC group. To remove the privilege for any users to create temporary tables, revoke the TEMP permission from the PUBLIC group and then explicitly grant the permission to create temporary tables to specific users or groups of users.

ON DATABASE db_name
Revokes the privileges on a database.

USAGE
Revokes USAGE privileges on objects within a specific schema, which makes these objects inaccessible to users. Specific actions on these objects must be revoked separately (such as EXECUTE privilege on functions).

Note
By default, all users have CREATE and USAGE privileges on the PUBLIC schema.

ON SCHEMA schema_name
Revokes the privileges on a schema. You can use schema privileges to control the creation of tables; the database CREATE privilege only controls schema creation.

CASCADE
If a user holds a privilege with grant option and has granted the privilege to other users, the privileges held by those other users are dependent privileges. If the privilege or the grant option held by the first user is being revoked and dependent privileges exist, those dependent privileges are also revoked if CASCADE is specified; otherwise, the revoke action fails.

For example, if user A has granted a privilege with grant option to user B, and user B has granted the privilege to user C, user A can revoke the grant option from user B and use the CASCADE option to in turn revoke the privilege from user C.

RESTRICT
Revokes only those privileges that the user directly granted. This is the default behavior.
Examples

The following example revokes INSERT privileges on the SALES table from the GUESTS user group. This command prevents GUESTS from being able to load data into the SALES table via the INSERT command:

```
revoke insert on table sales from group guests;
```

The following example revokes the privilege to select from a view for user bobr:

```
revoke select on table eventview from bobr;
```

The following example revokes the privilege to create temporary tables in the TICKIT database from all users:

```
revoke temporary on database tickit from public;
```

The following example controls table creation privileges in the PUBLIC schema. Subsequent to this command, users will be denied permission to create tables in the PUBLIC schema of the TICKIT database. (By default, all users have CREATE and USAGE privileges on the PUBLIC schema.)

```
revoke create on schema public from public;
```

ROLLBACK

Aborts the current transaction and discards all updates made by that transaction.

This command performs the same function as the ABORT (p. 257) command.

Synopsis

```
ROLLBACK [ WORK | TRANSACTION ]
```

Parameters

- WORK
  - Optional keyword.
- TRANSACTION
  - Optional keyword; WORK and TRANSACTION are synonyms.

Example

The following example creates a table then starts a transaction where data is inserted into the table. The ROLLBACK command then rolls back the data insertion to leave the table empty.

The following command creates an example table called MOVIE_GROSS:

```
create table movie_gross( name varchar(30), gross bigint );
```
The next set of commands starts a transaction that inserts two data rows into the table:

```sql
begin;
insert into movie_gross values ( 'Raiders of the Lost Ark', 23400000);
insert into movie_gross values ( 'Star Wars', 10000000 );
```

Next, the following command selects the data from the table to show that it was successfully inserted:

```sql
select * from movie_gross;
```

The command output shows that both rows successfully inserted:

<table>
<thead>
<tr>
<th>name</th>
<th>gross</th>
</tr>
</thead>
<tbody>
<tr>
<td>Raiders of the Lost Ark</td>
<td>23400000</td>
</tr>
<tr>
<td>Star Wars</td>
<td>10000000</td>
</tr>
</tbody>
</table>

This command now rolls back the data changes to where the transaction began:

```sql
rollback;
```

Selecting data from the table now shows an empty table:

```sql
select * from movie_gross;
```

<table>
<thead>
<tr>
<th>name</th>
<th>gross</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(0 rows)

**SELECT**

**Topics**

- Synopsis (p. 360)
- WITH clause (p. 360)
- SELECT list (p. 363)
- FROM clause (p. 366)
- WHERE clause (p. 368)
- GROUP BY clause (p. 373)
- HAVING clause (p. 374)
- UNION, INTERSECT, and EXCEPT (p. 375)
- ORDER BY clause (p. 383)
- Join examples (p. 385)
- Subquery examples (p. 386)
- Correlated subqueries (p. 387)

Returns rows from tables, views, and user-defined functions.
Note
The maximum size for a single SQL statement is 16 MB.

Synopsis

WITH with_subquery [, ... ]

SELECT
[ TOP number | [ ALL | DISTINCT ]
* | expression [ AS output_name ] [, ... ] ]
[ FROM table_reference [, ... ] ]
[ WHERE condition ]
[ GROUP BY expression [, ... ] ]
[ HAVING condition ]
[ [ UNION | ALL | INTERSECT | EXCEPT | MINUS ] query ]
[ ORDER BY expression
[ ASC | DESC ]
[ LIMIT { number | ALL } ]
[ OFFSET start ]

WITH clause

A WITH clause is an optional clause that precedes the SELECT list in a query. The WITH clause defines one or more subqueries. Each subquery defines a temporary table, similar to a view definition. These temporary tables can be referenced in the FROM clause and are used only during the execution of the query to which they belong. Each subquery in the WITH clause specifies a table name, an optional list of column names, and a query expression that evaluates to a table (a SELECT statement).

WITH clause subqueries are an efficient way of defining tables that can be used throughout the execution of a single query. In all cases, the same results can be achieved by using subqueries in the main body of the SELECT statement, but WITH clause subqueries may be simpler to write and read. Where possible, WITH clause subqueries that are referenced multiple times are optimized as common subexpressions; that is, it may be possible to evaluate a WITH subquery once and reuse its results. (Note that common subexpressions are not limited to those defined in the WITH clause.)

Synopsis

WITH with_subquery [, ... ]

where with_subquery is:

with_subquery_table_name [ ( column_name [, ... ] ) ] AS ( query )

Parameters

with_subquery_table_name
A unique name for a temporary table that defines the results of a WITH clause subquery. You cannot use duplicate names within a single WITH clause. Each subquery must be given a table name that can be referenced in the FROM clause (p. 366).

column_name
An optional list of output column names for the WITH clause subquery, separated by commas. The number of column names specified must be equal to or less than the number of columns defined by the subquery.

query
Any SELECT query that Amazon Redshift supports. See SELECT (p. 359).
Usage notes

You can use a WITH clause in the following SQL statements:

- SELECT (including subqueries within SELECT statements)
- SELECT INTO
- CREATE TABLE AS
- CREATE VIEW
- DECLARE
- EXPLAIN
- INSERT INTO...SELECT
- PREPARE
- UPDATE (within a WHERE clause subquery)

If the FROM clause of a query that contains a WITH clause does not reference any of the tables defined by the WITH clause, the WITH clause is ignored and the query executes as normal.

A table defined by a WITH clause subquery can be referenced only in the scope of the SELECT query that the WITH clause begins. For example, you can reference such a table in the FROM clause of a subquery in the SELECT list, WHERE clause, or HAVING clause. You cannot use a WITH clause in a subquery and reference its table in the FROM clause of the main query or another subquery. This query pattern results in an error message of the form relation table_name does not exist for the WITH clause table.

You cannot specify another WITH clause inside a WITH clause subquery.

You cannot make forward references to tables defined by WITH clause subqueries. For example, the following query returns an error because of the forward reference to table W2 in the definition of table W1:

```sql
with w1 as (select * from w2), w2 as (select * from w1)
select * from sales;
ERROR: relation "w2" does not exist
```

A WITH clause subquery may not consist of a SELECT INTO statement; however, you can use a WITH clause in a SELECT INTO statement.

Examples

The following example shows the simplest possible case of a query that contains a WITH clause. The WITH query named VENUECOPY selects all of the rows from the VENUE table. The main query in turn selects all of the rows from VENUECOPY. The VENUECOPY table exists only for the duration of this query.

```sql
with venuecopy as (select * from venue)
select * from venuecopy order by 1 limit 10;
```

<table>
<thead>
<tr>
<th>venueid</th>
<th>venuename</th>
<th>venuecity</th>
<th>venueseats</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Toyota Park</td>
<td>Bridgeview</td>
<td>IL</td>
</tr>
<tr>
<td>2</td>
<td>Columbus Crew Stadium</td>
<td>Columbus</td>
<td>OH</td>
</tr>
<tr>
<td>3</td>
<td>RFK Stadium</td>
<td>Washington</td>
<td>DC</td>
</tr>
</tbody>
</table>
The following example shows a WITH clause that produces two tables, named VENUE_SALES and TOP_VENUES. The second WITH query table selects from the first. In turn, the WHERE clause of the main query block contains a subquery that constrains the TOP_VENUES table.

```sql
with venue_sales as
  (select venuename, venuecity, sum(pricepaid) as venuename_sales
   from sales, venue, event
   where venue.venueid=event.venueid and event.eventid=sales.eventid
   group by venuename, venuecity),

top_venues as
  (select venuename
   from venue_sales
   where venuename_sales > 800000)

select venuename, venuecity, venuestate,
sum(qtysold) as venue_qty,
sum(pricepaid) as venue_sales
from sales, venue, event
where venue.venueid=event.venueid and event.eventid=sales.eventid
and venuename in(select venuename from top_venues)
group by venuename, venuecity, venuestate
order by venuename;
```

<table>
<thead>
<tr>
<th>venuename</th>
<th>venuecity</th>
<th>venuestate</th>
<th>venue_qty</th>
<th>venue_sales</th>
</tr>
</thead>
<tbody>
<tr>
<td>August Wilson Theatre</td>
<td>New York City</td>
<td>NY</td>
<td>3187</td>
<td>1032156.00</td>
</tr>
<tr>
<td>Biltmore Theatre</td>
<td>New York City</td>
<td>NY</td>
<td>2629</td>
<td>828981.00</td>
</tr>
<tr>
<td>Charles Playhouse</td>
<td>Boston</td>
<td>MA</td>
<td>2502</td>
<td>857031.00</td>
</tr>
<tr>
<td>Ethel Barrymore Theatre</td>
<td>New York City</td>
<td>NY</td>
<td>2828</td>
<td>891172.00</td>
</tr>
<tr>
<td>Eugene O’Neill Theatre</td>
<td>New York City</td>
<td>NY</td>
<td>2488</td>
<td>891172.00</td>
</tr>
<tr>
<td>Greek Theatre</td>
<td>Los Angeles</td>
<td>CA</td>
<td>2445</td>
<td>838918.00</td>
</tr>
<tr>
<td>Helen Hayes Theatre</td>
<td>New York City</td>
<td>NY</td>
<td>2948</td>
<td>978765.00</td>
</tr>
<tr>
<td>Hilton Theatre</td>
<td>New York City</td>
<td>NY</td>
<td>2999</td>
<td>885686.00</td>
</tr>
<tr>
<td>Imperial Theatre</td>
<td>New York City</td>
<td>NY</td>
<td>2702</td>
<td>877933.00</td>
</tr>
<tr>
<td>Lunt-Fontanne Theatre</td>
<td>New York City</td>
<td>NY</td>
<td>3326</td>
<td>1115182.00</td>
</tr>
<tr>
<td>Majestic Theatre</td>
<td>New York City</td>
<td>NY</td>
<td>2549</td>
<td>894275.00</td>
</tr>
<tr>
<td>Nederlander Theatre</td>
<td>New York City</td>
<td>NY</td>
<td>2934</td>
<td>936312.00</td>
</tr>
<tr>
<td>Pasadena Playhouse</td>
<td>Pasadena</td>
<td>CA</td>
<td>2739</td>
<td>820435.00</td>
</tr>
<tr>
<td>Winter Garden Theatre</td>
<td>New York City</td>
<td>NY</td>
<td>2838</td>
<td>939257.00</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>14 rows</td>
</tr>
</tbody>
</table>

The following two examples demonstrate the rules for the scope of table references based on WITH clause subqueries. The first query runs, but the second fails with an expected error. The first query has
WITH clause subquery inside the SELECT list of the main query. The table defined by the WITH clause (HOLIDAYS) is referenced in the FROM clause of the subquery in the SELECT list:

```sql
select caldate, sum(pricepaid) as daysales,  
(with holidays as (select * from date where holiday = 't')  
select sum(pricepaid)  
from sales join holidays on sales.dateid=holidays.dateid  
where caldate='2008-12-25') as dec25sales  
from sales join date on sales.dateid=date.dateid  
where caldate in('2008-12-25','2008-12-31')  
group by caldate  
order by caldate;
```

<table>
<thead>
<tr>
<th>caldate</th>
<th>daysales</th>
<th>dec25sales</th>
</tr>
</thead>
<tbody>
<tr>
<td>2008-12-25</td>
<td>70402.00</td>
<td>70402.00</td>
</tr>
<tr>
<td>2008-12-31</td>
<td>12678.00</td>
<td>70402.00</td>
</tr>
</tbody>
</table>

(2 rows)

The second query fails because it attempts to reference the HOLIDAYS table in the main query as well as in the SELECT list subquery. The main query references are out of scope.

```sql
select caldate, sum(pricepaid) as daysales,  
(with holidays as (select * from date where holiday = 't')  
select sum(pricepaid)  
from sales join holidays on sales.dateid=holidays.dateid  
where caldate='2008-12-25') as dec25sales  
from sales join holidays on sales.dateid=holidays.dateid  
where caldate in('2008-12-25','2008-12-31')  
group by caldate  
order by caldate;
```

ERROR: relation "holidays" does not exist

**SELECT list**

**Topics**

- Synopsis (p. 363)
- Parameters (p. 364)
- Usage notes (p. 364)
- Examples with TOP (p. 365)
- SELECT DISTINCT examples (p. 365)

The SELECT list names the columns, functions, and expressions that you want the query to return. The list represents the output of the query.

**Synopsis**

```
SELECT  
[ TOP number ]  
[ ALL | DISTINCT ] * | expression [ AS column_alias ] [, ...]
```
Parameters

TOP number
TOP takes a positive integer as its argument, which defines the number of rows that are returned to the client. The behavior with the TOP clause is the same as the behavior with the LIMIT clause. The number of rows that is returned is fixed, but the set of rows is not; to return a consistent set of rows, use TOP or LIMIT in conjunction with an ORDER BY clause.

ALL
A redundant keyword that defines the default behavior if you do not specify DISTINCT. SELECT ALL * means the same as SELECT * (select all rows for all columns and retain duplicates).

DISTINCT
Option that eliminates duplicate rows from the result set, based on matching values in one or more columns.

* (asterisk)
Returns the entire contents of the table (all columns and all rows).

expression
An expression formed from one or more columns that exist in the tables referenced by the query. An expression can contain SQL functions. For example:

```
SELECT avg(datediff(day, listtime, saletime))
```

AS column_alias
A temporary name for the column that will be used in the final result set. The AS keyword is optional. For example:

```
SELECT avg(datediff(day, listtime, saletime)) as avgwait
```

If you do not specify an alias for an expression that is not a simple column name, the result set applies a default name to that column.

Note
The alias is not recognized until the entire target list has been parsed, which means that you cannot refer to the alias elsewhere within the target list. For example, the following statement will fail:

```
SELECT (qtysold + 1) as q, sum(q) FROM sales GROUP BY 1;
```

ERROR: column "q" does not exist

You must use the same expression that was aliased to q:

```
SELECT (qtysold + 1) as q, sum(qtysold + 1) FROM sales GROUP BY 1;
```

<table>
<thead>
<tr>
<th>q</th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>368</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

Usage notes
TOP is a SQL extension; it provides an alternative to the LIMIT behavior. You cannot use TOP and LIMIT in the same query.
Examples with TOP

Return any 10 rows from the SALES table. Because no ORDER BY clause is specified, the set of rows that this query returns is unpredictable.

```
select top 10 *
from sales;
```

The following query is functionally equivalent, but uses a LIMIT clause instead of a TOP clause:

```
select *
from sales
limit 10;
```

Return the first 10 rows from the SALES table, ordered by the QTYSOLD column in descending order.

```
select top 10 qtysold, sellerid
from sales
order by qtysold desc, sellerid;
```

```
qtysold | sellerid
--------+----------
 8      |    518
 8      |    520
 8      |    574
 8      |    718
 8      |    868
 8      |   2663
 8      |   3396
 8      |   3726
 8      |   5250
 8      |   6216
(10 rows)
```

Return the first two QTYSOLD and SELLERID values from the SALES table, ordered by the QTYSOLD column:

```
select top 2 qtysold, sellerid
from sales
order by qtysold desc, sellerid;
```

```
qtysold | sellerid
--------+----------
 8      |    518
 8      |    520
(2 rows)
```

SELECT DISTINCT examples

Return a list of different category groups from the CATEGORY table:

```
select distinct catgroup from category
order by 1;
```
Return the distinct set of week numbers for December 2008:

```
select distinct week, month, year
from date
where month='DEC' and year=2008
order by 1, 2, 3;
```

```
week | month | year  
-----+-------+------
49   | DEC   | 2008 
50   | DEC   | 2008 
51   | DEC   | 2008 
52   | DEC   | 2008 
53   | DEC   | 2008 
(5 rows)
```

**FROM clause**

The FROM clause in a query lists the table references (tables, views, and subqueries) that data is selected from. If multiple table references are listed, the tables must be joined, using appropriate syntax in either the FROM clause or the WHERE clause. If no join criteria are specified, the system processes the query as a cross-join (Cartesian product).

**Synopsis**

```
FROM table_reference [, ...]
```

*where* *table_reference* is one of the following:

```
with_subquery_table_name [ [ AS ] alias [ ( column_alias [, ...] ) ] ]
table_name [ * ] [ [ AS ] alias [ ( column_alias [, ...] ) ] ]
( subquery ) [ AS ] alias [ ( column_alias [, ...] ) ]
table_reference [ NATURAL ] join_type table_reference
[ ON join_condition | USING ( join_column [, ...] ) ]
```

**Parameters**

*with_subquery_table_name*
A table defined by a subquery in the WITH clause (p. 360).

*table_name*
Name of a table or view.

*alias*
Temporary alternative name for a table or view. An alias must be supplied for a table derived from a subquery. In other table references, aliases are optional. The AS keyword is always optional. Table aliases provide a convenient shortcut for identifying tables in other parts of a query, such as the WHERE clause. For example:
column_alias
Temporary alternative name for a column in a table or view.

subquery
A query expression that evaluates to a table. The table exists only for the duration of the query and is typically given a name or alias; however, an alias is not required. You can also define column names for tables that derive from subqueries. Naming column aliases is important when you want to join the results of subqueries to other tables and when you want to select or constrain those columns elsewhere in the query.

A subquery may contain an ORDER BY clause, but this clause may have no effect if a LIMIT or OFFSET clause is not also specified.

NATURAL
Defines a join that automatically uses all pairs of identically named columns in the two tables as the joining columns. No explicit join condition is required. For example, if the CATEGORY and EVENT tables both have columns named CATID, a natural join of those tables is a join over their CATID columns.

Note
If a NATURAL join is specified but no identically named pairs of columns exist in the tables to be joined, the query defaults to a cross-join.

join_type
Specify one of the following types of join:
• [INNER] JOIN
• LEFT [OUTER] JOIN
• RIGHT [OUTER] JOIN
• FULL [OUTER] JOIN
• CROSS JOIN

ON join_condition
Type of join specification where the joining columns are stated as a condition that follows the ON keyword. For example:

```sql
sales join listing
on sales.listid=listing.listid and sales.eventid=listing.eventid
```

USING ( join_column [, ...] )
Type of join specification where the joining columns are listed in parentheses. If multiple joining columns are specified, they are delimited by commas. The USING keyword must precede the list. For example:

```sql
sales join listing
using (listid, eventid)
```

Join types
Cross-joins are unqualified joins; they return the Cartesian product of the two tables.

Inner and outer joins are qualified joins. They are qualified either implicitly (in natural joins); with the ON or USING syntax in the FROM clause; or with a WHERE clause condition.
An inner join returns matching rows only, based on the join condition or list of joining columns. An outer join returns all of the rows that the equivalent inner join would return plus non-matching rows from the "left" table, "right" table, or both tables. The left table is the first-listed table, and the right table is the second-listed table. The non-matching rows contain NULL values to fill the gaps in the output columns.

**Usage notes**

Joining columns must have comparable data types.

A NATURAL or USING join retains only one of each pair of joining columns in the intermediate result set.

A join with the ON syntax retains both joining columns in its intermediate result set.

See also [WITH clause](p. 360).

**WHERE clause**

The WHERE clause contains conditions that either join tables or apply predicates to columns in tables. Tables can be inner-joined by using appropriate syntax in either the WHERE clause or the FROM clause. Outer join criteria must be specified in the FROM clause.

**Synopsis**

```sql
[ WHERE condition ]
```

**condition**

Any search condition with a Boolean result, such as a join condition or a predicate on a table column. The following examples are valid join conditions:

```sql
sales.listid=listing.listid
sales.listid<>listing.listid
```

The following examples are valid conditions on columns in tables:

```sql
catgroup like 'S%'
venueseats between 20000 and 50000
eventname in('Jersey Boys','Spamalot')
year=2008
length(catdesc)>25
date_part(month, caldate)=6
```

Conditions can be simple or complex; for complex conditions, you can use parentheses to isolate logical units. In the following example, the join condition is enclosed by parentheses.

```sql
where (category.catid=event.catid) and category.catid in(6,7,8)
```

**Usage notes**

You cannot use aliases in the WHERE clause to reference select list expressions.

You cannot restrict the results of aggregate functions in the WHERE clause; use the HAVING clause for this purpose.

Columns that are restricted in the WHERE clause must derive from table references in the FROM clause.
Example

The following query uses a combination of different WHERE clause restrictions, including a join condition for the SALES and EVENT tables, a predicate on the EVENTNAME column, and two predicates on the STARTTIME column.

```sql
select eventname, starttime, pricepaid/qtysold as costperticket, qtysold
from sales, event
where sales.eventid = event.eventid
and eventname='Hannah Montana'
and date_part(quarter, starttime) in(1,2)
and date_part(year, starttime) = 2008
order by 3 desc, 4, 2, 1 limit 10;
```

<table>
<thead>
<tr>
<th>eventname</th>
<th>starttime</th>
<th>costperticket</th>
<th>qtysold</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hannah Montana</td>
<td>2008-06-07 14:00:00</td>
<td>1706.00000000</td>
<td>2</td>
</tr>
<tr>
<td>Hannah Montana</td>
<td>2008-05-01 19:00:00</td>
<td>1658.00000000</td>
<td>2</td>
</tr>
<tr>
<td>Hannah Montana</td>
<td>2008-06-07 14:00:00</td>
<td>1479.00000000</td>
<td>1</td>
</tr>
<tr>
<td>Hannah Montana</td>
<td>2008-06-07 14:00:00</td>
<td>1479.00000000</td>
<td>3</td>
</tr>
<tr>
<td>Hannah Montana</td>
<td>2008-06-07 14:00:00</td>
<td>1163.00000000</td>
<td>1</td>
</tr>
<tr>
<td>Hannah Montana</td>
<td>2008-06-07 14:00:00</td>
<td>1163.00000000</td>
<td>2</td>
</tr>
<tr>
<td>Hannah Montana</td>
<td>2008-06-07 14:00:00</td>
<td>1163.00000000</td>
<td>4</td>
</tr>
<tr>
<td>Hannah Montana</td>
<td>2008-05-01 19:00:00</td>
<td>497.00000000</td>
<td>1</td>
</tr>
<tr>
<td>Hannah Montana</td>
<td>2008-05-01 19:00:00</td>
<td>497.00000000</td>
<td>2</td>
</tr>
<tr>
<td>Hannah Montana</td>
<td>2008-05-01 19:00:00</td>
<td>497.00000000</td>
<td>4</td>
</tr>
</tbody>
</table>

(10 rows)

Oracle-style outer joins in the WHERE clause

For Oracle compatibility, Amazon Redshift supports the Oracle outer-join operator (+) in WHERE clause join conditions. This operator is intended for use only in defining outer-join conditions; do not try to use it in other contexts. Other uses of this operator are silently ignored in most cases.

An outer join returns all of the rows that the equivalent inner join would return, plus non-matching rows from one or both tables. In the FROM clause, you can specify left, right, and full outer joins. In the WHERE clause, you can specify left and right outer joins only.

To outer join tables TABLE1 and TABLE2 and return non-matching rows from TABLE1 (a left outer join), specify `TABLE1 LEFT OUTER JOIN TABLE2` in the FROM clause or apply the (+) operator to all joining columns from TABLE2 in the WHERE clause. For all rows in TABLE1 that have no matching rows in TABLE2, the result of the query contains nulls for any select list expressions that contain columns from TABLE2.

To produce the same behavior for all rows in TABLE2 that have no matching rows in TABLE1, specify `TABLE1 RIGHT OUTER JOIN TABLE2` in the FROM clause or apply the (+) operator to all joining columns from TABLE1 in the WHERE clause.

Basic syntax

```
[ WHERE {
[ table1.column1 = table2.column1(+) ]
[ table1.column1(+) = table2.column1 ]
}
```

The first condition is equivalent to:
from table1 left outer join table2
on table1.column1=table2.column1

The second condition is equivalent to:

from table1 right outer join table2
on table1.column1=table2.column1

**Note**
The syntax shown here covers the simple case of an equijoin over one pair of joining columns. However, other types of comparison conditions and multiple pairs of joining columns are also valid.

For example, the following WHERE clause defines an outer join over two pairs of columns. The (+) operator must be attached to the same table in both conditions:

```sql
where table1.col1 > table2.col1(+)
and table1.col2 = table2.col2(+)
```

**Usage notes**
Where possible, use the standard FROM clause OUTER JOIN syntax instead of the (+) operator in the WHERE clause. Queries that contain the (+) operator are subject to the following rules:

- You can only use the (+) operator in the WHERE clause, and only in reference to columns from tables or views.
- You cannot apply the (+) operator to expressions. However, an expression can contain columns that use the (+) operator. For example, the following join condition returns a syntax error:

```sql
event.eventid*10(+)=category.catid
```

However, the following join condition is valid:

```sql
event.eventid(+)*10=category.catid
```

- You cannot use the (+) operator in a query block that also contains FROM clause join syntax.
- If two tables are joined over multiple join conditions, you must use the (+) operator in all or none of these conditions. A join with mixed syntax styles executes as an inner join, without warning.
- The (+) operator does not produce an outer join if you join a table in the outer query with a table that results from an inner query.
- To use the (+) operator to outer-join a table to itself, you must define table aliases in the FROM clause and reference them in the join condition:

```sql
select count(*)
from event a, event b
where a.eventid(+)=b.catid;
```

```
count
-------
8798
(1 row)
```
• You cannot combine a join condition that contains the (+) operator with an OR condition or an IN condition. For example:

```sql
select count(*) from sales, listing
where sales.listid(+)=listing.listid or sales.salesid=0;
ERROR: Outer join operator (+) not allowed in operand of OR or IN.
```

• In a WHERE clause that outer-joins more than two tables, the (+) operator can be applied only once to a given table. In the following example, the SALES table cannot be referenced with the (+) operator in two successive joins.

```sql
select count(*) from sales, listing, event
where sales.listid(+)=listing.listid and sales.dateid(+)=date.dateid;
ERROR: A table may be outer joined to at most one other table.
```

• If the WHERE clause outer-join condition compares a column from TABLE2 with a constant, apply the (+) operator to the column. If you do not include the operator, the outer-joined rows from TABLE1, which contain nulls for the restricted column, are eliminated. See the Examples section below.

**Examples**

The following join query specifies a left outer join of the SALES and LISTING tables over their LISTID columns:

```sql
select count(*)
from sales, listing
where sales.listid = listing.listid(+);
```

```
count
--------
172456
(1 row)
```

The following equivalent query produces the same result but uses FROM clause join syntax:

```sql
select count(*)
from sales left outer join listing on sales.listid = listing.listid;
```

```
count
--------
172456
(1 row)
```

The SALES table does not contain records for all listings in the LISTING table because not all listings result in sales. The following query outer-joins SALES and LISTING and returns rows from LISTING even when the SALES table reports no sales for a given list ID. The PRICE and COMM columns, derived from the SALES table, contain nulls in the result set for those non-matching rows.

```sql
select listing.listid, sum(pricepaid) as price,
sum(commission) as comm
from listing, sales
where sales.listid(+) = listing.listid and listing.listid between 1 and 5
group by 1 order by 1;
```
Note that when the WHERE clause join operator is used, the order of the tables in the FROM clause does not matter.

An example of a more complex outer join condition in the WHERE clause is the case where the condition consists of a comparison between two table columns and a comparison with a constant:

```sql
where category.catid=event.catid(+) and eventid(+) = 796;
```

Note that the (+) operator is used in two places: first in the equality comparison between the tables and second in the comparison condition for the EVENTID column. The result of this syntax is the preservation of the outer-joined rows when the restriction on EVENTID is evaluated. If you remove the (+) operator from the EVENTID restriction, the query treats this restriction as a filter, not as part of the outer-join condition. In turn, the outer-joined rows that contain nulls for EVENTID are eliminated from the result set.

Here is a complete query that illustrates this behavior:

```sql
select catname, catgroup, eventid
from category, event
where category.catid = event.catid(+) and eventid(+) = 796;
```

The equivalent query using FROM clause syntax is as follows:

```sql
select catname, catgroup, eventid
from category left join event
on category.catid = event.catid and eventid = 796;
```

If you remove the second (+) operator from the WHERE clause version of this query, it returns only 1 row (the row where `eventid = 796`).
select catname, catgroup, eventid
from category, event
where category.catid=event.catid(+) and eventid=796;

catname | catgroup | eventid
-----------+----------+---------
Musicals | Shows    | 796
(1 row)

GROUP BY clause

The GROUP BY clause identifies the grouping columns for the query. Grouping columns must be declared when the query computes aggregates with standard functions such as SUM, AVG, and COUNT.

GROUP BY expression [, ...]

expression

The list of columns or expressions must match the list of non-aggregate expressions in the select list of the query. For example, consider the following simple query:

select listid, eventid, sum(pricepaid) as revenue,
count(qtysold) as numtix
from sales
group by listid, eventid
order by 3, 4, 2, 1
limit 5;

listid | eventid | revenue | numtix
--------+---------+---------+--------
89397   | 47      | 20.00   | 1      
106590  | 76      | 20.00   | 1      
124683  | 393     | 20.00   | 1      
103037  | 403     | 20.00   | 1      
147685  | 429     | 20.00   | 1      
(5 rows)

In this query, the select list consists of two aggregate expressions. The first uses the SUM function and the second uses the COUNT function. The remaining two columns, LISTID and EVENTID, must be declared as grouping columns.

Expressions in the GROUP BY clause can also reference the select list by using ordinal numbers. For example, the previous example could be abbreviated as follows:

select listid, eventid, sum(pricepaid) as revenue,
count(qtysold) as numtix
from sales
group by 1,2
order by 3, 4, 2, 1
limit 5;

listid | eventid | revenue | numtix
--------+---------+---------+--------
89397   | 47      | 20.00   | 1      

HAVING clause

The HAVING clause applies a condition to the intermediate grouped result set that a query returns.

Synopsis

```
[ HAVING condition ]
```

For example, you can restrict the results of a SUM function:

```
having sum(pricepaid) >10000
```

The HAVING condition is applied after all WHERE clause conditions are applied and GROUP BY operations are completed.

The condition itself takes the same form as any WHERE clause condition.

Usage notes

- Any column that is referenced in a HAVING clause condition must be either a grouping column or a column that refers to the result of an aggregate function.
- In a HAVING clause, you cannot specify:
  - An alias that was defined in the select list. You must repeat the original, unaliased expression.
  - An ordinal number that refers to a select list item. Only the GROUP BY and ORDER BY clauses accept ordinal numbers.

Examples

The following query calculates total ticket sales for all events by name, then eliminates events where the total sales were less than $800,000. The HAVING condition is applied to the results of the aggregate function in the select list: `sum(pricepaid)`.

```
select eventname, sum(pricepaid)
from sales join event on sales.eventid = event.eventid
group by 1
having sum(pricepaid) > 800000
order by 2 desc, 1;
```

```
eventname     |    sum
------------------+-----------
Mamma Mia!       | 1135454.00
Spring Awakening |  972855.00
The Country Girl |  910563.00
Macbeth          |  862580.00
Jersey Boys      |  811877.00
```
The following query calculates a similar result set. In this case, however, the HAVING condition is applied to an aggregate that is not specified in the select list: \( \text{sum(qtysold)} \). Events that did not sell more than 2,000 tickets are eliminated from the final result.

```sql
select eventname, sum(pricepaid)
from sales join event on sales.eventid = event.eventid
group by 1
having sum(qtysold) > 2000
order by 2 desc, 1;
```

Eventname     |    Sum  
------------------+-----------
Mamma Mia!       | 1135454.00
Spring Awakening |  972855.00
The Country Girl |  910563.00
Macbeth          |  862580.00
Jersey Boys      |  811877.00
Legally Blonde   |  804583.00
Chicago          |  790993.00
Spamalot         |  714307.00

(8 rows)

**UNION, INTERSECT, and EXCEPT**

**Topics**

- Synopsis (p. 375)
- Parameters (p. 376)
- Order of evaluation for set operators (p. 376)
- Usage notes (p. 377)
- Example UNION queries (p. 378)
- Example UNION ALL query (p. 379)
- Example INTERSECT queries (p. 380)
- Example EXCEPT query (p. 381)

The UNION, INTERSECT, and EXCEPT set operators are used to compare and merge the results of two separate query expressions. For example, if you want to know which users of a web site are both buyers and sellers but their user names are stored in separate columns or tables, you can find the *intersection* of these two types of users. If you want to know which web site users are buyers but not sellers, you can use the EXCEPT operator to find the *difference* between the two lists of users. If you want to build a list of all users, regardless of role, you can use the UNION operator.

**Synopsis**

```sql
query
{ UNION [ ALL ] | INTERSECT | EXCEPT | MINUS }
query
```
Parameters

query

A query expression that corresponds, in the form of its select list, to a second query expression that follows the UNION, INTERSECT, or EXCEPT operator. The two expressions must contain the same number of output columns with compatible data types; otherwise, the two result sets cannot be compared and merged. Set operations do not allow implicit conversion between different categories of data types; see Type compatibility and conversion (p. 232).

You can build queries that contain an unlimited number of query expressions and link them with UNION, INTERSECT, and EXCEPT operators in any combination. For example, the following query structure is valid, assuming that the tables T1, T2, and T3 contain compatible sets of columns:

```
select * from t1
union
select * from t2
except
select * from t3
order by c1;
```

UNION

Set operation that returns rows from two query expressions, regardless of whether the rows derive from one or both expressions.

INTERSECT

Set operation that returns rows that derive from two query expressions. Rows that are not returned by both expressions are discarded.

EXCEPT | MINUS

Set operation that returns rows that derive from one of two query expressions. To qualify for the result, rows must exist in the first result table but not the second. MINUS and EXCEPT are exact synonyms.

ALL

The ALL keyword retains any duplicate rows that are produced by UNION. The default behavior when the ALL keyword is not used is to discard these duplicates. INTERSECT ALL, EXCEPT ALL, and MINUS ALL are not supported.

Order of evaluation for set operators

The UNION and EXCEPT set operators are left-associative. If parentheses are not specified to influence the order of precedence, a combination of these set operators is evaluated from left to right. For example, in the following query, the UNION of T1 and T2 is evaluated first, then the EXCEPT operation is performed on the UNION result:

```
select * from t1
union
select * from t2
except
select * from t3
order by c1;
```

The INTERSECT operator takes precedence over UNION and EXCEPT operators when a combination of operators is used in the same query. For example, the following query will evaluate the intersection of T2 and T3, then union the result with T1:
```
select * from t1
union
select * from t2
intersect
select * from t3
order by c1;
```

By adding parentheses, you can enforce a different order of evaluation. In the following case, the result of the union of T1 and T2 is intersected with T3, and the query is likely to produce a different result.

```
(select * from t1
union
select * from t2)
intersect
(select * from t3)
order by c1;
```

**Usage notes**

- The column names returned in the result of a set operation query are the column names (or aliases) from the tables in the first query expression. Because these column names are potentially misleading, in that the values in the column derive from tables on either side of the set operator, you might want to provide meaningful aliases for the result set.
- A query expression that precedes a set operator should not contain an ORDER BY clause. An ORDER BY clause produces meaningful sorted results only when it is used at the end of a query that contains set operators. This ORDER BY clause applies to the final results of all of the set operations. The outermost query may also contain standard LIMIT and OFFSET clauses.
- The LIMIT and OFFSET clauses are not supported as a means of restricting the number of rows returned by an intermediate result of a set operation. For example, the following query returns an error:

```
(select listid from listing
limit 10)
intersect
select listid from sales;
```

```
ERROR: LIMIT may not be used within input to set operations.
```

- When set operator queries return decimal results, the corresponding result columns are promoted to return the same precision and scale. For example, in the following query, where T1.REVENUE is a DECIMAL(10,2) column and T2.REVENUE is a DECIMAL(8,4) column, the decimal result is promoted to DECIMAL(12,4):

```
select t1.revenue union select t2.revenue;
```

The scale is 4 because that is the maximum scale of the two columns. The precision is 12 because T1.REVENUE requires 8 digits to the left of the decimal point (12 - 4 = 8). This type promotion ensures that all values from both sides of the UNION fit in the result. For 64-bit values, the maximum result precision is 19 and the maximum result scale is 18. For 128-bit values, the maximum result precision is 38 and the maximum result scale is 37.

If the resulting data type exceeds Amazon Redshift precision and scale limits, the query returns an error.

- For set operations, two rows are treated as identical if, for each corresponding pair of columns, the two data values are either **equal** or both **NULL**. For example, if tables T1 and T2 both contain one column
and one row, and that row is NULL in both tables, an INTERSECT operation over those tables returns that row.

**Example UNION queries**

In the following UNION query, rows in the SALES table are merged with rows in the LISTING table. Three compatible columns are selected from each table; in this case, the corresponding columns have the same names and data types.

The final result set is ordered by the first column in the LISTING table and limited to the 5 rows with the highest LISTID value.

```sql
select listid, sellerid, eventid from listing
union select listid, sellerid, eventid from sales
order by listid, sellerid, eventid desc limit 5;
```

<table>
<thead>
<tr>
<th>listid</th>
<th>sellerid</th>
<th>eventid</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>36861</td>
<td>7872</td>
</tr>
<tr>
<td>2</td>
<td>16002</td>
<td>4806</td>
</tr>
<tr>
<td>3</td>
<td>21461</td>
<td>4256</td>
</tr>
<tr>
<td>4</td>
<td>8117</td>
<td>4337</td>
</tr>
<tr>
<td>5</td>
<td>1616</td>
<td>8647</td>
</tr>
</tbody>
</table>

The following example shows how you can add a literal value to the output of a UNION query so you can see which query expression produced each row in the result set. The query identifies rows from the first query expression as "B" (for buyers) and rows from the second query expression as "S" (for sellers).

The query identifies buyers and sellers for ticket transactions that cost $10,000 or more. The only difference between the two query expressions on either side of the UNION operator is the joining column for the SALES table.

```sql
select listid, lastname, firstname, username, pricepaid as price, 'S' as buyorsell
from sales, users
where sales.sellerid=users.userid
and pricepaid >=10000
union
select listid, lastname, firstname, username, pricepaid, 'B' as buyorsell
from sales, users
where sales.buyerid=users.userid
and pricepaid >=10000
order by 1, 2, 3, 4, 5;
```

<table>
<thead>
<tr>
<th>listid</th>
<th>lastname</th>
<th>firstname</th>
<th>username</th>
<th>price</th>
<th>buyorsell</th>
</tr>
</thead>
<tbody>
<tr>
<td>209658</td>
<td>Lamb</td>
<td>Colette</td>
<td>VOR15LYI</td>
<td>10000.00</td>
<td>B</td>
</tr>
<tr>
<td>209658</td>
<td>West</td>
<td>Kato</td>
<td>ELU81XAA</td>
<td>10000.00</td>
<td>S</td>
</tr>
<tr>
<td>212395</td>
<td>Greer</td>
<td>Harlan</td>
<td>GX071KOC</td>
<td>12624.00</td>
<td>S</td>
</tr>
<tr>
<td>212395</td>
<td>Perry</td>
<td>Cora</td>
<td>YWR73YNZ</td>
<td>12624.00</td>
<td>B</td>
</tr>
<tr>
<td>215156</td>
<td>Banks</td>
<td>Patrick</td>
<td>ZNQ69CLT</td>
<td>10000.00</td>
<td>S</td>
</tr>
<tr>
<td>215156</td>
<td>Hayden</td>
<td>Malachi</td>
<td>BBG56AKU</td>
<td>10000.00</td>
<td>B</td>
</tr>
</tbody>
</table>

(6 rows)
The following example uses a UNION ALL operator because duplicate rows, if found, need to be retained in the result. For a specific series of event IDs, the query returns 0 or more rows for each sale associated with each event, and 0 or 1 row for each listing of that event. Event IDs are unique to each row in the LISTING and EVENT tables, but there might be multiple sales for the same combination of event and listing IDs in the SALES table.

The third column in the result set identifies the source of the row. If it comes from the SALES table, it is marked “Yes” in the SALESROW column. (SALESROW is an alias for SALES.LISTID.) If the row comes from the LISTING table, it is marked “No” in the SALESROW column.

In this case, the result set consists of three sales rows for listing 500, event 7787. In other words, three different transactions took place for this listing and event combination. The other two listings, 501 and 502, did not produce any sales, so the only row that the query produces for these list IDs comes from the LISTING table (SALESROW = 'No').

```sql
select eventid, listid, 'Yes' as salesrow
from sales
where listid in(500,501,502)
union all
select eventid, listid, 'No'
from listing
where listid in(500,501,502)
order by listid asc;
```

<table>
<thead>
<tr>
<th>eventid</th>
<th>listid</th>
<th>salesrow</th>
</tr>
</thead>
<tbody>
<tr>
<td>7787</td>
<td>500</td>
<td>No</td>
</tr>
<tr>
<td>7787</td>
<td>500</td>
<td>Yes</td>
</tr>
<tr>
<td>7787</td>
<td>500</td>
<td>Yes</td>
</tr>
<tr>
<td>6473</td>
<td>501</td>
<td>No</td>
</tr>
<tr>
<td>5108</td>
<td>502</td>
<td>No</td>
</tr>
</tbody>
</table>

(6 rows)

If you run the same query without the ALL keyword, the result retains only one of the sales transactions.

```sql
select eventid, listid, 'Yes' as salesrow
from sales
where listid in(500,501,502)
union
select eventid, listid, 'No'
from listing
where listid in(500,501,502)
order by listid asc;
```

<table>
<thead>
<tr>
<th>eventid</th>
<th>listid</th>
<th>salesrow</th>
</tr>
</thead>
<tbody>
<tr>
<td>7787</td>
<td>500</td>
<td>No</td>
</tr>
<tr>
<td>7787</td>
<td>500</td>
<td>Yes</td>
</tr>
<tr>
<td>6473</td>
<td>501</td>
<td>No</td>
</tr>
<tr>
<td>5108</td>
<td>502</td>
<td>No</td>
</tr>
</tbody>
</table>

(4 rows)

Example UNION ALL query

The following example uses a UNION ALL operator because duplicate rows, if found, need to be retained in the result. For a specific series of event IDs, the query returns 0 or more rows for each sale associated
with each event, and 0 or 1 row for each listing of that event. Event IDs are unique to each row in the LISTING and EVENT tables, but there might be multiple sales for the same combination of event and listing IDs in the SALES table.

The third column in the result set identifies the source of the row. If it comes from the SALES table, it is marked "Yes" in the SALESROW column. (SALESROW is an alias for SALES.LISTID.) If the row comes from the LISTING table, it is marked "No" in the SALESROW column.

In this case, the result set consists of three sales rows for listing 500, event 7787. In other words, three different transactions took place for this listing and event combination. The other two listings, 501 and 502, did not produce any sales, so the only row that the query produces for these list IDs comes from the LISTING table (SALESROW = 'No').

Example INTERSECT queries

Compare the following example with the first UNION example. The only difference between the two examples is the set operator that is used, but the results are very different. Only one of the rows is the same:
This is the only row in the limited result of 5 rows that was found in both tables.

The following query finds events (for which tickets were sold) that occurred at venues in both New York City and Los Angeles in March. The difference between the two query expressions is the constraint on the VENUECITY column.

Example EXCEPT query

The CATEGORY table in the TICKIT database contains the following 11 rows:
Assume that a CATEGORY_STAGE table (a staging table) contains one additional row:

| catid | catgroup | catname      |             catdesc                                      |
|-------+----------+-------------+------------------------------------------------------|
| 1     | Sports   | MLB         | Major League Baseball                                 |
| 2     | Sports   | NHL         | National Hockey League                                |
| 3     | Sports   | NFL         | National Football League                              |
| 4     | Sports   | NBA         | National Basketball Association                       |
| 5     | Sports   | MLS         | Major League Soccer                                   |
| 6     | Shows    | Musicals    | Musical theatre                                       |
| 7     | Shows    | Plays       | All non-musical theatre                               |
| 8     | Shows    | Opera       | All opera and light opera                             |
| 9     | Concerts | Pop         | All rock and pop music concerts                       |
| 10    | Concerts | Jazz        | All jazz singers and bands                            |
| 11    | Concerts | Classical    | All symphony, concerto, and choir concerts            |

Return the difference between the two tables. In other words, return rows that are in the CATEGORY_STAGE table but not in the CATEGORY table:

```
select * from category_stage
except
select * from category;
```

| catid | catgroup | catname     |             catdesc                                      |
|-------+----------+-------------+------------------------------------------------------|
| 12    | Concerts | Comedy      | All stand up comedy performances                      |

The following equivalent query uses the synonym MINUS.

```
select * from category_stage
minus
select * from category;
```

| catid | catgroup | catname     |             catdesc                                      |
|-------+----------+-------------+------------------------------------------------------|
| 12    | Concerts | Comedy      | All stand up comedy performances                      |
If you reverse the order of the SELECT expressions, the query returns no rows.

**ORDER BY clause**

**Topics**
- Synopsis (p. 383)
- Parameters (p. 383)
- Usage notes (p. 384)
- Examples with ORDER BY (p. 384)

The ORDER BY clause sorts the result set of a query.

**Synopsis**

```
[ ORDER BY expression
[ ASC | DESC ]
[ LIMIT { count | ALL } ]
[ OFFSET start ]
```

**Parameters**

**expression**

Defines the sort order of the query result set, typically by specifying one or more columns in the select list. You can also specify:

- Columns that are not in the select list
- Expressions formed from one or more columns that exist in the tables referenced by the query
- Ordinal numbers that represent the position of select list entries (or the position of columns in the table if no select list exists)
- Aliases that define select list entries

When the ORDER BY clause contains multiple expressions, the result set is sorted according to the first expression, then the second expression is applied to rows that have matching values from the first expression, and so on.

**ASC | DESC**

Option that defines the sort order for the expression:

- ASC: ascending (for example, low to high for numeric values and 'A' to 'Z' for character strings).
  If no option is specified, data is sorted in ascending order by default.
- DESC: descending (high to low for numeric values; 'Z' to 'A' for strings)

**LIMIT number | ALL**

Option that controls the number of sorted rows that the query returns. The LIMIT number must be a positive integer; the maximum value is 2147483647.

LIMIT 0 returns no rows; you can use this syntax for testing purposes: to check that a query runs (without displaying any rows) or to return a column list from a table. An ORDER BY clause is redundant if you are using LIMIT 0 to return a column list. The default is LIMIT ALL.

**OFFSET start**

Skip the number of rows before start before beginning to return rows. The OFFSET number must be a positive integer; the maximum value is 2147483647. When used with the LIMIT option, OFFSET rows are skipped before starting to count the LIMIT rows that are returned. If the LIMIT option is not used, the number of rows in the result set is reduced by the number of rows that are skipped. The
rows skipped by an OFFSET clause still have to be scanned, so it might be inefficient to use a large OFFSET value.

**Usage notes**

Note the following expected behavior with ORDER BY clauses:

- NULL values are considered “higher” than all other values. With default ascending sort order, NULL values sort at the end.
- When a query does not contain an ORDER BY clause, the system returns result sets with no predictable ordering of the rows. The same query executed twice might return the result set in a different order.
- The LIMIT and OFFSET options can be used without an ORDER BY clause; however, to return a consistent set of rows, use these options in conjunction with ORDER BY.
- In any parallel system like Amazon Redshift, when ORDER BY does not produce a unique ordering, the order of the rows is non-deterministic. That is, if the ORDER BY expression produces duplicate values, the return order of those rows may vary from other systems or from one run of Amazon Redshift to the next.

**Examples with ORDER BY**

Return all 11 rows from the CATEGORY table, ordered by the second column, CATGROUP. For results that have the same CATGROUP value, order the CATDESC column values by the length of the character string. The other two columns, CATID and CATNAME, do not influence the order of results.

```
select * from category order by 2, length(catdesc), 1, 3;
```

<table>
<thead>
<tr>
<th>catid</th>
<th>catgroup</th>
<th>catname</th>
<th>catdesc</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Concerts</td>
<td>Jazz</td>
<td>All jazz singers and bands</td>
</tr>
<tr>
<td>9</td>
<td>Concerts</td>
<td>Pop</td>
<td>All rock and pop music concerts</td>
</tr>
<tr>
<td>11</td>
<td>Concerts</td>
<td>Classical</td>
<td>All symphony, concerto, and choir conce</td>
</tr>
<tr>
<td>6</td>
<td>Shows</td>
<td>Musicals</td>
<td>Musical theatre</td>
</tr>
<tr>
<td>7</td>
<td>Shows</td>
<td>Plays</td>
<td>All non-musical theatre</td>
</tr>
<tr>
<td>8</td>
<td>Shows</td>
<td>Opera</td>
<td>All opera and light opera</td>
</tr>
<tr>
<td>5</td>
<td>Sports</td>
<td>MLS</td>
<td>Major League Soccer</td>
</tr>
<tr>
<td>1</td>
<td>Sports</td>
<td>MLB</td>
<td>Major League Baseball</td>
</tr>
<tr>
<td>2</td>
<td>Sports</td>
<td>NHL</td>
<td>National Hockey League</td>
</tr>
<tr>
<td>3</td>
<td>Sports</td>
<td>NFL</td>
<td>National Football League</td>
</tr>
<tr>
<td>4</td>
<td>Sports</td>
<td>NBA</td>
<td>National Basketball Association</td>
</tr>
</tbody>
</table>

(11 rows)

Return selected columns from the SALES table, ordered by the highest QTYSOLD values. Limit the result to the top 10 rows:

```
select salesid, qtysold, pricepaid, commission, saletime from sales
order by qtysold, pricepaid, commission, salesid, saletime desc
limit 10;
```

<table>
<thead>
<tr>
<th>salesid</th>
<th>qtysold</th>
<th>pricepaid</th>
<th>commission</th>
<th>saletime</th>
</tr>
</thead>
<tbody>
<tr>
<td>15401</td>
<td>8</td>
<td>272.00</td>
<td>40.80</td>
<td>2008-03-18 06:54:56</td>
</tr>
<tr>
<td>61683</td>
<td>8</td>
<td>296.00</td>
<td>44.40</td>
<td>2008-11-26 04:00:23</td>
</tr>
<tr>
<td>90528</td>
<td>8</td>
<td>328.00</td>
<td>49.20</td>
<td>2008-06-11 02:38:09</td>
</tr>
<tr>
<td>74549</td>
<td>8</td>
<td>336.00</td>
<td>50.40</td>
<td>2008-01-19 12:01:21</td>
</tr>
</tbody>
</table>
Return a column list and no rows by using LIMIT 0 syntax:

```
select * from venue limit 0;
```

```
venueid | venuename | venuecity | venuestate | venueseats
---------+-----------+-----------+------------+------------
(0 rows)
```

### Join examples

The following query is an outer join. Left and right outer joins retain values from one of the joined tables when no match is found in the other table. The left and right tables are the first and second tables listed in the syntax. NULL values are used to fill the "gaps" in the result set.

This query matches LISTID column values in LISTING (the left table) and SALES (the right table). The results show that listings 2, 3, and 5 did not result in any sales.

```
select listing.listid, sum(pricepaid) as price, sum(commission) as comm
from listing left outer join sales on sales.listid = listing.listid
where listing.listid between 1 and 5
group by 1
order by 1;
```

```
listid | price  |  comm
--------+--------+--------
1 | 728.00 | 109.20
2 |        |        
3 |        |        
4 |  76.00 |  11.40
5 | 525.00 |  78.75
(5 rows)
```

The following query is an inner join of two subqueries in the FROM clause. The query finds the number of sold and unsold tickets for different categories of events (concerts and shows):

```
select catgroup1, sold, unsold
from
(select catgroup, sum(qtysold) as sold
from category c, event e, sales s
where ccatid = e.catid and e.eventid = s.eventid
group by catgroup) as a(catgroup1, sold)
join
(select catgroup, sum(numtickets)-sum(qtysold) as unsold
from category c, event e, sales s, listing l
where ccatid = e.catid and e.eventid = s.eventid
and s.listid = l.listid
group by catgroup) as b(catgroup2, unsold)
```

```
These FROM clause subqueries are table subqueries; they can return multiple columns and rows.

## Subquery examples

The following examples show different ways in which subqueries fit into SELECT queries. See Join examples (p. 385) for another example of the use of subqueries.

### SELECT list subquery

The following example contains a subquery in the SELECT list. This subquery is scalar: it returns only one column and one value, which is repeated in the result for each row that is returned from the outer query. The query compares the Q1SALES value that the subquery computes with sales values for two other quarters (2 and 3) in 2008, as defined by the outer query.

```sql
select qtr, sum(pricepaid) as qtrsales,
       (select sum(pricepaid)
        from sales join date on sales.dateid=date.dateid
        where qtr='1' and year=2008) as q1sales
from sales join date on sales.dateid=date.dateid
where qtr in('2','3') and year=2008
group by qtr
order by qtr;
```

<table>
<thead>
<tr>
<th>qtr</th>
<th>qtrsales</th>
<th>q1sales</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>30560050.00</td>
<td>24742065.00</td>
</tr>
<tr>
<td>3</td>
<td>3170237.00</td>
<td>24742065.00</td>
</tr>
</tbody>
</table>

### WHERE clause subquery

The following example contains a table subquery in the WHERE clause. This subquery produces multiple rows. In this case, the rows contain only one column, but table subqueries can contain multiple columns and rows, just like any other table.

The query finds the top 10 sellers in terms of maximum tickets sold. The top 10 list is restricted by the subquery, which removes users who live in cities where there are ticket venues. This query can be written in different ways; for example, the subquery could be rewritten as a join within the main query.

```sql
select firstname, lastname, city, max(qtysold) as maxsold
from users join sales on users.userid=sales.sellerid
where users.city not in(select venuecity from venue)
group by firstname, lastname, city
order by maxsold desc, city desc
limit 10;
```
WITH clause subqueries

See WITH clause (p. 360).

Correlated subqueries

The following example contains a *correlated subquery* in the WHERE clause; this kind of subquery contains one or more correlations between its columns and the columns produced by the outer query. In this case, the correlation is \( \text{where } \text{s.listid} = \text{l.listid} \). For each row that the outer query produces, the subquery is executed to qualify or disqualify the row.

```sql
select salesid, listid, sum(pricepaid) from sales s
where qtysold =
  (select max(numtickets) from listing l
  where s.listid = l.listid)
group by 1,2
order by 1,2
limit 5;
```

<table>
<thead>
<tr>
<th>salesid</th>
<th>listid</th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>27</td>
<td>28</td>
<td>111.00</td>
</tr>
<tr>
<td>81</td>
<td>103</td>
<td>181.00</td>
</tr>
<tr>
<td>142</td>
<td>149</td>
<td>240.00</td>
</tr>
<tr>
<td>146</td>
<td>152</td>
<td>231.00</td>
</tr>
<tr>
<td>194</td>
<td>210</td>
<td>144.00</td>
</tr>
</tbody>
</table>

(5 rows)

Correlated subquery patterns that are not supported

The query planner uses a query rewrite method called subquery decorrelation to optimize several patterns of correlated subqueries for execution in an MPP environment. A few types of correlated subqueries follow patterns that Amazon Redshift cannot decorrelate and does not support. Queries that contain the following correlation references return errors:

- Correlation references that skip a query block, also known as "skip-level correlation references." For example, in the following query, the block containing the correlation reference and the skipped block are connected by a NOT EXISTS predicate:

select event.eventname from event
where not exists
(select * from listing
where not exists
(select * from sales where event.eventid=sales.eventid));

The skipped block in this case is the subquery against the LISTING table. The correlation reference correlates the EVENT and SALES tables.

• Correlation references from a subquery that is part of an ON clause in an outer join:

select * from category
left join event
on category.catid=event.catid and eventid =
(select max(eventid) from sales where sales.eventid=event.eventid);

The ON clause contains a correlation reference from SALES in the subquery to EVENT in the outer query.

• Null-sensitive correlation references to an Amazon Redshift system table. For example:

select attrelid
from stv_locks sl, pg_attribute
where sl.table_id=pg_attribute.attrelid and 1 not in
(select 1 from pg_opclass where sl.lock_owner = opcowner);

• Correlation references from within a subquery that contains a window function.

select listid, qtysold
from sales s
where qtysold not in
(select sum(numtickets) over() from listing l where s.listid=l.listid);

• References in a GROUP BY column to the results of a correlated subquery. For example:

select listing.listid,
(select count (sales.listid) from sales where sales.listid=listing.listid) as list
from listing
group by list, listing.listid;

• Correlation references from a subquery with an aggregate function and a GROUP BY clause, connected to the outer query by an IN predicate. (This restriction does not apply to MIN and MAX aggregate functions.) For example:

select * from listing where listid in
(select sum(qtysold)
from sales
where numtickets>4
group by salesid);
SELECT INTO

Selects rows defined by any query and inserts them into a new temporary or persistent table.

Synopsis

```sql
[ WITH with_subquery [, ...] ]
SELECT
[ TOP number ] [ ALL | DISTINCT ]
* | expression [ AS output_name ] [, ...]
INTO [ TEMPORARY | TEMP ] [ TABLE ] new_table
[ FROM table_reference [, ...] ]
[ WHERE condition ]
[ GROUP BY expression [, ...] ]
[ HAVING condition [, ...] ]
[ { UNION | INTERSECT | { EXCEPT | MINUS } } [ ALL ] query ]
[ ORDER BY expression
[ ASC | DESC ]
[ LIMIT { number | ALL } ]
[ OFFSET start ]
```

For details about the parameters of this command, see SELECT (p. 359).

Examples

Select all of the rows from the EVENT table and create a NEWEVENT table:

```sql
select * into newevent from event;
```

Select the result of an aggregate query into a temporary table called PROFITS:

```sql
select username, lastname, sum(pricepaid-commission) as profit
into temp table profits
from sales, users
where sales.sellerid=users.userid
group by 1, 2
order by 3 desc;
```

SET

Sets the value of a server configuration parameter.

Use the RESET (p. 355) command to return a parameter to its default value. See Modifying the server configuration (p. 693) for more information about parameters.

Synopsis

```sql
SET { [ SESSION | LOCAL ]
parameter_name { TO | = }
{ value | 'value' | DEFAULT } |
SEED TO value }
```
Parameters

SESSION
  Specifies that the setting is valid for the current session. Default value.

LOCAL
  Specifies that the setting is valid for the current transaction.

SEED TO value
  Sets an internal seed to be used by the RANDOM function for random number generation.

  SET SEED takes a numeric value between 0 and 1, and multiplies this number by \(2^{31}-1\) for use with the RANDOM function (p. 511) function. If you use SET SEED before making multiple RANDOM calls, RANDOM generates numbers in a predictable sequence.

parameter_name
  Name of the parameter to set. See Modifying the server configuration (p. 693) for information about parameters.

value
  New parameter value. Use single quotes to set the value to a specific string. If using SET SEED, this parameter contains the SEED value.

DEFAULT
  Sets the parameter to the default value.

LOCAL
  Sets the time zone to the local time zone (where the Amazon Redshift server resides).

Examples

Changing a parameter for the current session

The following example sets the datestyle:

```sql
set datestyle to 'SQL,DMY';
```

Setting a query group for workload management

If query groups are listed in a queue definition as part of the cluster's WLM configuration, you can set the QUERY_GROUP parameter to a listed query group name. Subsequent queries are assigned to the associated query queue. The QUERY_GROUP setting remains in effect for the duration of the session or until a RESET QUERY_GROUP command is encountered.

This example runs two queries as part of the query group ‘priority’, then resets the query group.

```sql
set query_group to 'priority';
select tbl, count(*)from stv_blocklist;
select query, elapsed, substring from svl_qlog order by query desc limit 5;
reset query_group;
```

See Implementing workload management (p. 196)

Setting a label for a group of queries

The QUERY_GROUP parameter defines a label for one or more queries that are executed in the same session after a SET command. In turn, this label is logged when queries are executed and can be used to constrain results returned from the STL_QUERY and STV_INFLIGHT system tables and the SVL_QLOG view.
show query_group;
query_group
------------
unset
(1 row)

set query_group to '6 p.m.';

show query_group;
query_group
------------
6 p.m.
(1 row)

select * from sales where salesid=500;
salesid | listid | sellerid | buyerid | eventid | dateid | ...
---------+--------+----------+---------+---------+--------+-----
500 | 504 | 3858 | 2123 | 5871 | 2052 | ...
(1 row)

reset query_group;

select query, trim(label) querygroup, pid, trim(querytxt) sql
from stl_query
where label = '6 p.m.';
query | querygroup |  pid  |                  sql
-------+------------+-------+----------------------------------------
57 | 6 p.m.     | 30711 | select * from sales where salesid=500;
(1 row)

Query group labels are a useful mechanism for isolating individual queries or groups of queries that are
run as part of scripts. You do not need to identify and track queries by their IDs; you can track them by
their labels.

Setting a seed value for random number generation

The following example uses the SEED option with SET to cause the RANDOM function to generate
numbers in a predictable sequence.

First, return three RANDOM integers without setting the SEED value first:

select cast (random() * 100 as int);
int4
-----
6
(1 row)

select cast (random() * 100 as int);
int4
-----
68
(1 row)

select cast (random() * 100 as int);
int4
-----
Now, set the SEED value to .25, and return three more RANDOM numbers:

```
set seed to .25;
select cast (random() * 100 as int);
int4
------
21
(1 row)

select cast (random() * 100 as int);
int4
------
79
(1 row)

select cast (random() * 100 as int);
int4
------
12
(1 row)
```

Finally, reset the SEED value to .25, and verify that RANDOM returns the same results as the previous three calls:

```
set seed to .25;
select cast (random() * 100 as int);
int4
------
21
(1 row)

select cast (random() * 100 as int);
int4
------
79
(1 row)

select cast (random() * 100 as int);
int4
------
12
(1 row)
```

**SET SESSION AUTHORIZATION**

Sets the user name for the current session.

You can use the SET SESSION AUTHORIZATION command, for example, to test database access by temporarily running a session or transaction as an unprivileged user.
Synopsis

```
SET [ SESSION | LOCAL ] SESSION AUTHORIZATION ( user_name | DEFAULT )
```

Parameters

**SESSION**
Specifies that the setting is valid for the current session. Default value.

**LOCAL**
Specifies that the setting is valid for the current transaction.

**user_name**
Name of the user to set. The user name may be written as an identifier or a string literal.

**DEFAULT**
Sets the session user name to the default value.

Examples

The following example sets the user name for the current session to `dwuser`:

```
SET SESSION AUTHORIZATION 'dwuser';
```

The following example sets the user name for the current transaction to `dwuser`:

```
SET LOCAL SESSION AUTHORIZATION 'dwuser';
```

This example sets the user name for the current session to the default user name:

```
SET SESSION AUTHORIZATION DEFAULT;
```

SET SESSION CHARACTERISTICS

This command is deprecated.

SHOW

Displays the current value of a server configuration parameter. This value may be specific to the current session if a SET command is in effect. For a list of configuration parameters, see Configuration Reference (p. 693).

Synopsis

```
SHOW { parameter_name | ALL }
```

Parameters

**parameter_name**
Displays the current value of the specified parameter.
Displays the current values of all of the parameters.

Examples

The following example displays the value for the query_group parameter:

```sql
show query_group;
query_group
unset
(1 row)
```

The following example displays a list of all parameters and their values:

```sql
show all;
name        |   setting
--------------------+--------------
datestyle          | ISO, MDY
extra_float_digits | 0
query_group        | unset
search_path        | $user,public
statement_timeout  | 0
```

START TRANSACTION

Synonym of the BEGIN function.

See BEGIN (p. 270).

TRUNCATE

Deletes all of the rows from a table without doing a table scan: this operation is a faster alternative to an unqualified DELETE operation.

Synopsis

```sql
TRUNCATE [ TABLE ] table_name
```

Parameters

TABLE
Optional keyword.

`table_name`
A temporary or persistent table. Only the owner of the table or a superuser may truncate it.

You can truncate any table, including tables that are referenced in foreign-key constraints.

After truncating a table, run the ANALYZE command against the table. You do not need to vacuum a table after truncating it.
Usage notes

The TRUNCATE command commits the transaction in which it is run; therefore, you cannot roll back a TRUNCATE operation, and a TRUNCATE command may commit other operations when it commits itself.

Examples

Use the TRUNCATE command to delete all of the rows from the CATEGORY table:

```
truncate category;
```

Attempt to roll back a TRUNCATE operation:

```
begin;
truncate date;
rollback;
```

```
select count(*) from date;
count
--------
0
(1 row)
```

The DATE table remains empty after the ROLLBACK command because the TRUNCATE command committed automatically.

UNLOAD

Unloads the result of a query to one or more files on Amazon S3.

Synopsis

```
UNLOAD ('select_statement')
TO 's3://object_path_prefix'
[ WITH ] CREDENTIALS [AS] 'aws_access_credentials'
[ option [ ... ] ]

where option is

{ MANIFEST
 | DELIMITER [ AS ] 'delimiter_char'
 | FIXEDWIDTH [ AS ] 'fixedwidth_spec'
 | ENCRYPTED
 | GZIP
 | ADDQUOTES
 | NULL [ AS ] 'null_string'
 | ESCAPE
 | ALLOWOVERWRITE
 | PARALLEL [ { ON | TRUE } | { OFF | FALSE } ]
```
Parameters

\[ \text{('select\_statement')} \]
Defines a SELECT query. The results of the query are unloaded. In most cases, it is worthwhile to unload data in sorted order by specifying an ORDER BY clause in the query; this approach will save the time required to sort the data when it is reloaded.

The query must be enclosed in single quotes. For example:

\[ \text{('select * from venue order by venueid')} \]

**Note**
If your query contains quotes (enclosing literal values, for example), you need to escape them in the query text. For example:

\[ \text{('select * from venue where venuestate='NV\'\')} \]

**TO 'S3://object\_path\_prefix'**
The path prefix on Amazon S3 where Amazon Redshift will write the output files, including the manifest file if MANIFEST is specified. By default, UNLOAD writes one or more files per slice. File names are written in the format `<object_path_prefix><slice-number>_part_<file-number>`. If MANIFEST is specified, the manifest file is written in the format `<object_path_prefix>manifest`.

**Important**
The Amazon S3 bucket where Amazon Redshift will write the output files must reside in the same region as your cluster.

**WITH**
This keyword is optional.

**CREDENTIALS [AS] 'aws\_access\_credentials'**
The AWS access credentials for the Amazon S3 bucket. The access credentials must belong to an AWS account user or an IAM user with READ and WRITE permission for the Amazon S3 bucket to which the data files are being unloaded.

You can optionally use temporary credentials to access the Amazon S3 bucket. If you use temporary credentials, include the temporary session token in the credentials string. For more information, see Temporary security credentials (p. 290) in Usage notes for the COPY command.

The `aws_access_credentials` string must not contain spaces.

If only access key and secret access key are supplied, the `aws_access_credentials` string is in the format:

\[ 'aws\_access\_key\_id=<access\_key\_id>\; aws\_secret\_access\_key=<secret\_access\_key>\] \]

To use temporary token credentials, you must provide the temporary access key ID, the temporary secret access key, and the temporary token. The `aws_access_credentials` string is in the format

\[ 'aws\_access\_key\_id=<temporary\_access\_key\_id>\; aws\_secret\_access\_key=<temporary\_secret\_access\_key>\; token=<temporary\_token>\] \]

If the ENCRYPTED option is used, the `aws_access_credentials` string is in the format
aws_access_key_id=<access-key-id>; aws_secret_access_key=<secret-access-key>; master_symmetric_key=<master_key>

where <master_key> is the value of the master key that UNLOAD will use to encrypt the files. The master key must be a base64 encoded 256 bit AES symmetric key.

MANIFEST
Creates a manifest file that explicitly lists the data files that are created by the UNLOAD process. The manifest is a text file in JSON format that lists the URL of each file that was written to Amazon S3. The manifest file is written to the same Amazon S3 path prefix as the unload files in the format <object_path_prefix>manifest. For example, if the UNLOAD specifies the Amazon S3 path prefix 'S3://mybucket/venue_', the manifest file location will be 'S3://mybucket/venue_manifest'.

DELIMITER AS 'delimiter_character'
Single ASCII character that is used to separate fields in the output file, such as a pipe character (|), a comma (,), or a tab (\t). The default delimiter is a pipe character. The AS keyword is optional. DELIMITER cannot be used with FIXEDWIDTH. If the data contains the delimiter character, you will need to specify the ESCAPE option to escape the delimiter, or use ADDQUOTES to enclose the data in double quotes. Alternatively, specify a delimiter that is not contained in the data.

FIXEDWIDTH 'fixedwidth_spec'
Unloads the data to a file where each column width is a fixed length, rather than separated by a delimiter. The fixedwidth_spec is a string that specifies the number of columns and the width of the columns. The AS keyword is optional. FIXEDWIDTH cannot be used with DELIMITER. Because FIXEDWIDTH does not truncate data, the specification for each column in the UNLOAD statement needs to be at least as long as the length of the longest entry for that column. The format for fixedwidth_spec is shown below:

'colID1:colWidth1,colID2:colWidth2, ...'

ENCrypted
Specifies that the output files on Amazon S3 will be encrypted using Amazon S3 client-side encryption. See Unloading encrypted data files (p. 176). To unload to encrypted gzip-compressed files, add the GZIP option.

GZIP
Unloads data to one or more gzip-compressed file per slice. Each resulting file is appended with a .gz extension.

ADDQUOTES
Places quotation marks around each unloaded data field, so that Amazon Redshift can unload data values that contain the delimiter itself. For example, if the delimiter is a comma, you could unload and reload the following data successfully:

"1","Hello, World"

Without the added quotes, the string Hello, World would be parsed as two separate fields.

If you use ADDQUOTES, you must specify REMOVEQUOTES in the COPY if you reload the data.

NULL AS 'null_string'
Specifies a string that represents a null value in unload files. If this option is used, all output files contain the specified string in place of any null values found in the selected data. If this option is not specified, null values are unloaded as:
- Zero-length strings for delimited output
- Whitespace strings for fixed-width output
If a null string is specified for a fixed-width unload and the width of an output column is less than the width of the null string, the following behavior occurs:
- An empty field is output for non-character columns
- An error is reported for character columns

**ESCAPE**
For CHAR and VARCHAR columns in delimited unload files, an escape character (\) is placed before every occurrence of the following characters:
- Linefeed: \n
- Carriage return: \r
- The delimiter character specified for the unloaded data.
- The escape character: \n
- A quote character: " or ' (if both ESCAPE and ADDQUOTES are specified in the UNLOAD command).

**Important**
If you loaded your data using a COPY with the ESCAPE option, you must also specify the ESCAPE option with your UNLOAD command to generate the reciprocal output file. Similarly, if you UNLOAD using the ESCAPE option, you will need to use ESCAPE when you COPY the same data.

**ALLOWOVERWRITE**
By default, UNLOAD fails if it finds files that it would possibly overwrite. If ALLOWOVERWRITE is specified, UNLOAD will overwrite existing files, including the manifest file.

**PARALLEL**
By default, UNLOAD writes data in parallel to multiple files, according to the number of slices in the cluster. The default option is ON or TRUE. If PARALLEL is OFF or FALSE, UNLOAD writes to one or more data files serially, sorted absolutely according to the ORDER BY clause, if one is used. The maximum size for a data file is 6.2 GB. So, for example, if you unload 13.4 GB of data, UNLOAD creates the following three files:

<table>
<thead>
<tr>
<th>Location</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>s3://mybucket/key000</td>
<td>6.2 GB</td>
</tr>
<tr>
<td>s3://mybucket/key001</td>
<td>6.2 GB</td>
</tr>
<tr>
<td>s3://mybucket/key002</td>
<td>1.0 GB</td>
</tr>
</tbody>
</table>

**Note**
The UNLOAD command is designed to use parallel processing. We recommend leaving PARALLEL enabled for most cases, especially if the files will be used to load tables using a COPY command.

**Usage notes**

**Using ESCAPE for all delimited UNLOAD operations**
When you UNLOAD using a delimiter and there is any possibility that your data includes the delimiter or any of the characters listed in the ESCAPE option description, you must use the ESCAPE option with the UNLOAD statement. If you do not use the ESCAPE option with the UNLOAD, subsequent COPY operations using the unloaded data might fail.

**Important**
We strongly recommend that you always use ESCAPE with both UNLOAD and COPY statements unless you are certain that your data does not contain any delimiters or other characters that might need to be escaped.
Loss of floating-point precision

You might encounter loss of precision for floating-point data that is successively unloaded and reloaded.

Limit clause

The SELECT query cannot use a LIMIT clause in the outer SELECT. For example, the following UNLOAD statement will fail:

```
unload ('select * from venue limit 10')
ton 's3://mybucket/venue_pipe_' credentials
'aws_access_key_id=<access-key-id>;aws_secret_access_key=<secret-access-key>
'';
```

Instead, use a nested LIMIT clause. For example:

```
unload ('select * from venue where venueid in
(select venueid from venue order by venueid desc limit 10)')
ton 's3://mybucket/venue_pipe_' credentials
'aws_access_key_id=<access-key-id>;aws_secret_access_key=<secret-access-key>
'';
```

Alternatively, you could populate a table using SELECT...INTO or CREATE TABLE AS using a LIMIT clause, then unload from that table.

UNLOAD examples

Unload VENUE to a pipe-delimited file (default delimiter)

**Note**

These examples contain line breaks for readability. Do not include line breaks or spaces in your `aws_access_credentials` string.

The following example unloads the VENUE table and writes the data to `s3://mybucket/unload/`:

```
unload ('select * from venue')
ton 's3://mybucket/unload/' credentials
'aws_access_key_id=<access-key-id>;aws_secret_access_key=<secret-access-key>
'';
```

By default, UNLOAD writes one or more files per slice. Assuming a two-node cluster with two slices per node, the previous example creates these files in `mybucket`:

```
unload/0000_part_00
unload/0001_part_00
unload/0002_part_00
unload/0003_part_00
```

To better differentiate the output files, you can include a prefix in the location. The following example unloads the VENUE table and writes the data to `s3://mybucket/venue_pipe_`:

```
unload ('select * from venue')
ton 's3://mybucket/unload/venue_pipe_' credentials
'aws_access_key_id=<access-key-id>;aws_secret_access_key=<secret-access-key>
'';
```

The result is these four files in the `unload` folder, again assuming four slices.
Unload VENUE serially

To unload serially, specify PARALLEL OFF. UNLOAD will then write one file at a time, up to a maximum of 6.2 GB per file.

The following example unloads the VENUE table and writes the data serially to `s3://mybucket/unload/`:

```
unload ('select * from venue')
to 's3://mybucket/unload/venue_serial_' credentials 'aws_access_key_id=<access-key-id>;aws_secret_access_key=<secret-access-key>' parallel off;
```

The result is one file named `venue_serial_000`.

If the unload data is larger than 6.2 GB, UNLOAD creates a new file for each 6.2 GB data segment. The following example unloads the LINEORDER table and writes the data serially to `s3://mybucket/unload/`:

```
unload ('select * from lineorder')
to 's3://mybucket/unload/lineorder_serial_' credentials 'aws_access_key_id=<access-key-id>;aws_secret_access_key=<secret-access-key>' parallel off gzip;
```

The result is the following series of files:

```
lineorder_serial_0000.gz
lineorder_serial_0001.gz
lineorder_serial_0002.gz
lineorder_serial_0003.gz
```

To better differentiate the output files, you can include a prefix in the location. The following example unloads the VENUE table and writes the data to `s3://mybucket/venue_pipe_`:

```
unload ('select * from venue')
to 's3://mybucket/venue_pipe_' credentials 'aws_access_key_id=<access-key-id>;aws_secret_access_key=<secret-access-key>' parallel off;
```

The result is these four files in the `unload` folder, again assuming four slices:

```
venue_pipe_0000_part_00
venue_pipe_0001_part_00
venue_pipe_0002_part_00
venue_pipe_0003_part_00
```

Unload VENUE with a manifest file

To create a manifest file, include the MANIFEST option. The following example unloads the VENUE table and writes a manifest file along with the data files to `s3://mybucket/venue_pipe_`:

```
unload ('select * from venue')
to 's3://mybucket/venue_pipe_' credentials 'aws_access_key_id=<access-key-id>;aws_secret_access_key=<secret-access-key>' manifest;
```

The result is these four files in the `unload` folder, again assuming four slices:

```
venue_pipe_0000_part_00
venue_pipe_0001_part_00
venue_pipe_0002_part_00
venue_pipe_0003_part_00
```
Important
If you unload files with the MANIFEST option, you should use the MANIFEST option with the COPY command when you load the files. If you use the same prefix to load the files and do not specify the MANIFEST option, COPY will fail because it assumes the manifest file is a data file.

```
unload ('select * from venue')
to 's3://mybucket/venue_pipe_' credentials 'aws_access_key_id=<access-key-id>;aws_secret_access_key=<secret-access-key>' manifest;
```

The result is these five files:

```
s3://mybucket/venue_pipe_0000_part_00
s3://mybucket/venue_pipe_0001_part_00
s3://mybucket/venue_pipe_0002_part_00
s3://mybucket/venue_pipe_0003_part_00
s3://mybucket/venue_pipe_manifest
```

Load VENUE from unload files
To load a table from a set of unload files, simply reverse the process by using a COPY command. The following example creates a new table, LOADVENUE, and loads the table from the data files created in the previous example.

```
create table loadvenue (like venue);

copy loadvenue from 's3://mybucket/venue_pipe_' credentials 'aws_access_key_id=<access-key-id>;aws_secret_access_key=<secret-access-key>';
```

If you used the MANIFEST option to create a manifest file with your unload files, you can load the data using the same manifest file with a COPY command with the MANIFEST option. The following example loads data using a manifest file.

```
copy loadvenue from 's3://mybucket/venue_pipe_manifest' credentials 'aws_access_key_id=<access-key-id>;aws_secret_access_key=<secret-access-key>' manifest;
```

Unload VENUE to encrypted files
The following example unloads the VENUE table to a set of encrypted files. If you specify a manifest file with the ENCRYPTED option, the manifest file is not encrypted. For more information, see Unloading encrypted data files (p. 176)

```
unload ('select * from venue')
to 's3://mybucket/venue_encrypt_' credentials 'aws_access_key_id=<your-access-key-id>;aws_secret_access_key=<your-secret-access-key>;master_symmetric_key=EXAMPLEMASTERKEYtkbjk/OpCwtYSx/M4/t7DMCDIK722' manifest encrypted;
```
Load VENUE from encrypted files

To load tables from a set of files that were created by using UNLOAD with the ENCRYPT option, reverse the process by using a COPY command with the ENCRYPTED option and specify the same master symmetric key that was used for the UNLOAD command. The following example loads the LOADVENUE table from the encrypted data files created in the previous example.

```sql
create table loadvenue (like venue);
copy loadvenue from 's3://mybucket/venue_encrypt_manifest' credentials 'aws_access_key_id=<your-access-key-id>;aws_secret_access_key=<your-secret-access-key>;master_symmetric_key=EXAMPLEMASTERKEYtkbjk/OpCwtYSx/M4/t7DMCDIK722' manifest encrypted;
```

Unload VENUE data to a tab-delimited file

```sql
unload ('select venueid, venuename, venueseats from venue') to 's3://mybucket/venue_tab_' credentials 'aws_access_key_id=<access-key-id>;aws_secret_access_key=<secret-access-key>' delimiter as '\t';
```

The output data files look like this:

```
1 Toyota Park Bridgeview IL 0
2 Columbus Crew Stadium Columbus OH 0
3 RFK Stadium Washington DC 0
4 CommunityAmerica Ballpark Kansas City KS 0
5 Gillette Stadium Foxborough MA 68756
...
```

Unload VENUE using temporary credentials

You can limit the access users have to your data by using temporary security credentials. Temporary security credentials provide enhanced security because they have short life spans and cannot be reused after they expire. A user who has these temporary security credentials can access your resources only until the credentials expire. For more information, see Temporary security credentials (p. 290) in the usage notes for the COPY command.

The following example unloads the LISTING table using temporary credentials:

```sql
unload ('select venueid, venuename, venueseats from venue') to 's3://mybucket/venue_tab' credentials 'aws_access_key_id=<temporary-access-key-id>;aws_secret_access_key=<temporary-secret-access-key>;token=<temporary-token>' delimiter as '\t';
```

Important

The temporary security credentials must be valid for the entire duration of the UNLOAD statement. If the temporary security credentials expire during the load process, the UNLOAD will fail and the transaction will be rolled back. For example, if temporary security credentials expire after 15 minutes and the UNLOAD requires one hour, the UNLOAD will fail before it completes.
Unload VENUE to a fixed-width data file

unload ('select * from venue')
to 's3://mybucket/venue_fw_'
credentials 'aws_access_key_id=<access-key-id>;aws_secret_access_key=<secret-access-key>'
fixedwidth as 'venueid:3,venuename:39,venuecity:16,venuestate:2,venueseats:6';

The output data files will look like this:

1  Toyota Park              Bridgeview  IL0
2  Columbus Crew Stadium    Columbus    OH0
3  RFK Stadium              Washington  DC0
4  CommunityAmerica Ballpark Kansas City KS0
5  Gillette Stadium        Foxborough  MA68756
...

Unload VENUE to a set of tab-delimited GZIP-compressed files

unload ('select * from venue')
to 's3://mybucket/venue_tab_'
credentials 'aws_access_key_id=<access-key-id>;aws_secret_access_key=<secret-access-key>'
delimiter as '"t'
gzip;

Unload data that contains a delimiter

This example uses the ADDQUOTES option to unload comma-delimited data where some of the actual data fields contain a comma.

First, create a table that contains quotes.

create table location (id int, location char(64));

insert into location values (1,'Phoenix, AZ'),(2,'San Diego, CA'),(3,'Chicago, IL');

Then, unload the data using the ADDQUOTES option.

unload ('select id, location from location')
to 's3://mybucket/location_'
credentials 'aws_access_key_id=<access-key-id>;aws_secret_access_key=<secret-access-key>'
delimiter ',' addquotes;

The unloaded data files look like this:

1,"Phoenix, AZ"
2,"San Diego, CA"
3,"Chicago, IL"
...
Unload the results of a join query

The following example unloads the results of a join query that contains a window function.

```sql
unload ('select venuecity, venuestate, caldate, pricepaid,
        sum(pricepaid) over(partition by venuecity, venuestate
        order by caldate rows between 3 preceding and 3 following) as winsum
        from sales join date on sales.dateid=date.dateid
        join event on event.eventid=sales.eventid
        join venue on event.venueid=venue.venueid
        order by 1,2')
to 's3://mybucket/tickit/winsum'
credentials 'aws_access_key_id=<access-key-id>
            ;aws_secret_access_key=<secret-access-key>';
```

The output files look like this:

```
Atlanta|GA|2008-01-04|363.00|1362.00
Atlanta|GA|2008-01-05|233.00|2030.00
Atlanta|GA|2008-01-06|310.00|3135.00
Atlanta|GA|2008-01-08|166.00|8338.00
Atlanta|GA|2008-01-11|268.00|7630.00
...```

Unload using NULL AS

UNLOAD outputs null values as empty strings by default. The following examples show how to use NULL AS to substitute a text string for nulls.

For these examples, we will add some null values to the VENUE table.

```sql
update venue set venuestate = NULL
where venuecity = 'Cleveland';
```

Select from VENUE where VENUESTATE is null to verify that the columns contain NULL.

```sql
select * from venue where venuestate is null;
```

```
venueid | venuename         | venuecity | venuestate | venueseats
---------+--------------------------+-----------+------------+------------
22       | Quicken Loans Arena      | Cleveland |            |          0
101      | Progressive Field        | Cleveland |            | 43345
72       | Cleveland Browns Stadium | Cleveland |            | 73200
(3 rows)
```

Now, UNLOAD the VENUE table using the NULL AS option to replace null values with the character string 'fred'.

```sql
unload ('select * from venue')
to 's3://mybucket/nulls/'
credentials 'aws_access_key_id=<access-key-id>
            ;aws_secret_access_key=<secret-access-key>'
null as 'fred';
```
The following sample from the unload file shows that null values were replaced with "fred". It turns out that some values for VENUESEATS were also null and were replaced with "fred". Even though the data type for VENUESEATS is integer, UNLOAD converts the values to text in the unload files, and then COPY converts them back to integer. If you are unloading to a fixed-width file, the NULL AS string must not be larger than the field width.

| 248 | Charles Playhouse | Boston | MA | 0 |
| 251 | Paris Hotel | Las Vegas | NV | fred |
| 258 | Tropicana Hotel | Las Vegas | NV | fred |
| 300 | Kennedy Center Opera House | Washington | DC | 0 |
| 306 | Lyric Opera House | Baltimore | MD | 0 |
| 308 | Metropolitan Opera | New York City | NY | 0 |
| 5 | Gillette Stadium | Foxborough | MA | 5 |
| 22 | Quicken Loans Arena | Cleveland | fred | 0 |
| 101 | Progressive Field | Cleveland | fred | 43345 |

To load a table from the unload files, use a COPY command with the same NULL AS option.

Note
If you attempt to load nulls into a column defined as NOT NULL, the COPY command will fail.

```
create table loadvenuenulls (like venue);

copy loadvenuenulls from 's3://mybucket/nulls/'
credentials 'aws_access_key_id=<access-key-id>;aws_secret_access_key=<secret-access-key>'
null as 'fred';
```

To verify that the columns contain null, not just empty strings, select from LOADVENUENULLS and filter for null.

```
select * from loadvenuenulls where venuestate is null or venueseats is null;
```

<table>
<thead>
<tr>
<th>venueid</th>
<th>venuename</th>
<th>venuecity</th>
<th>venuestate</th>
<th>venueseats</th>
</tr>
</thead>
<tbody>
<tr>
<td>72</td>
<td>Cleveland Browns Stadium</td>
<td>Cleveland</td>
<td></td>
<td>73200</td>
</tr>
<tr>
<td>253</td>
<td>Mirage Hotel</td>
<td>Las Vegas</td>
<td>NV</td>
<td></td>
</tr>
<tr>
<td>255</td>
<td>Venetian Hotel</td>
<td>Las Vegas</td>
<td>NV</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>Quicken Loans Arena</td>
<td>Cleveland</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>101</td>
<td>Progressive Field</td>
<td>Cleveland</td>
<td></td>
<td>43345</td>
</tr>
<tr>
<td>225</td>
<td>Paris Hotel</td>
<td>Las Vegas</td>
<td>NV</td>
<td></td>
</tr>
</tbody>
</table>

You can UNLOAD a table that contains nulls using the default NULL AS behavior and then COPY the data back into a table using the default NULL AS behavior; however, any non-numeric fields in the target table will contain empty strings, not nulls. By default UNLOAD converts nulls to empty strings (white space or zero-length). COPY converts empty strings to NULL for numeric columns, but inserts empty strings into non-numeric columns. The following example shows how to perform an UNLOAD followed by a COPY using the default NULL AS behavior.

```
unload ('select * from venue')
to 's3://mybucket/nulls/
credentials 'aws_access_key_id=<access-key-id>;aws_secret_access_key=<secret-
```
In this case, when you filter for nulls, only the rows where VENUESEATS contained nulls. Where VENUESTATE contained nulls in the table (VENUE), VENUESTATE in the target table (LOADVENUENULLS) contains empty strings.

```
select * from loadvenuenulls where venuestate is null or venueseats is null;
```

```
venueid | venuename         | venuecity | venuestate | venueseats
---------+--------------------------+-----------+------------+------------
    253 | Mirage Hotel             | Las Vegas | NV         |            |
    255 | Venetian Hotel           | Las Vegas | NV         |            |
    251 | Paris Hotel              | Las Vegas | NV         |            |
    ... |
```

To load empty strings to non-numeric columns as NULL, include the EMPTYASNULL or BLANKSASNULL options. It's OK to use both.

```
unload ('select * from venue')
to 's3://mybucket/nulls/'
credentials 'aws_access_key_id=<access-key-id>;aws_secret_access_key=<secret-access-key>' allowoverwrite;
```

```
truncate loadvenuenulls;
```

```
unload ('select * from venue')
to 's3://mybucket/nulls/'
credentials 'aws_access_key_id=<access-key-id>;aws_secret_access_key=<secret-access-key>' allowoverwrite;
```

```
truncate loadvenuenulls;
```

```
copy loadvenuenulls from 's3://mybucket/nulls/'
credentials 'aws_access_key_id=<access-key-id>;aws_secret_access_key=<secret-access-key>' EMPTYASNULL;
```

To verify that the columns contain NULL, not just whitespace or empty, select from LOADVENUENULLS and filter for null.

```
select * from loadvenuenulls where venuestate is null or venueseats is null;
```

```
venueid | venuename         | venuecity | venuestate | venueseats
---------+--------------------------+-----------+------------+------------
     72 | Cleveland Browns Stadium | Cleveland |            | 73200      |
    253 | Mirage Hotel             | Las Vegas | NV         |            |
    255 | Venetian Hotel           | Las Vegas | NV         |            |
     22 | Quicken Loans Arena      | Cleveland |            |            |
     101 | Progressive Field        | Cleveland |            | 43345      |
    251 | Paris Hotel              | Las Vegas | NV         |            |
    ... |
```

**ALLOWOVERWRITE example**

By default, UNLOAD will not overwrite existing files in the destination bucket. For example, if you run the same UNLOAD statement twice without modifying the files in the destination bucket, the second UNLOAD will fail. To overwrite the existing files, including the manifest file, specify the ALLOWOVERWRITE option.
UPDATE

Topics

- Synopsis (p. 407)
- Parameters (p. 407)
- Usage notes (p. 408)
- Examples of UPDATE statements (p. 408)

Updates values in one or more table columns when a condition is satisfied.

**Note**
The maximum size for a single SQL statement is 16 MB.

**Synopsis**

```
UPDATE table_name SET column = { expression | DEFAULT } [,...]  
[ FROM fromlist ]  
[ WHERE condition ]
```

**Parameters**

**table_name**
A temporary or persistent table. Only the owner of the table or a user with UPDATE privilege on the table may update rows. If you use the FROM clause or select from tables in an expression or condition, you must have SELECT privilege on those tables. You cannot give the table an alias here; however, you can specify an alias in the FROM clause.

**SET column =**
One or more columns that you want to modify. Columns that are not listed retain their current values. Do not include the table name in the specification of a target column. For example, `UPDATE tab SET tab.col = 1` is invalid.

**expression**
An expression that defines the new value for the specified column.

**DEFAULT**
Updates the column with the default value that was assigned to the column in the CREATE TABLE statement.

**FROM tablelist**
You can update a table by referencing information in other tables. List these other tables in the FROM clause or use a subquery as part of the WHERE condition. Tables listed in the FROM clause can have aliases. If you need to include the target table of the UPDATE statement in the list, use an alias.

**WHERE condition**
Optional clause that restricts updates to rows that match a condition. When the condition returns true, the specified SET columns are updated. The condition can be a simple predicate on a column or a condition based on the result of a subquery.

You can name any table in the subquery, including the target table for the UPDATE.
Usage notes

After updating a large number of rows in a table:

- Vacuum the table to reclaim storage space and resort rows.
- Analyze the table to update statistics for the query planner.

Left, right, and full outer joins are not supported in the FROM clause of an UPDATE statement; they return the following error:

```
ERROR: Target table must be part of an equijoin predicate
```

If you need to specify an outer join, use a subquery in the WHERE clause of the UPDATE statement.

If your UPDATE statement requires a self-join to the target table, you need to specify the join condition as well as the WHERE clause criteria that qualify rows for the update operation. In general, when the target table is joined to itself or another table, a best practice is to use a subquery that clearly separates the join conditions from the criteria that qualify rows for updates.

Examples of UPDATE statements

The CATEGORY table in the TICKIT database contains the following rows:

<table>
<thead>
<tr>
<th>catid</th>
<th>catgroup</th>
<th>catname</th>
<th>catdesc</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Sports</td>
<td>MLB</td>
<td>Major League Baseball</td>
</tr>
<tr>
<td>2</td>
<td>Sports</td>
<td>NHL</td>
<td>National Hockey League</td>
</tr>
<tr>
<td>3</td>
<td>Sports</td>
<td>NFL</td>
<td>National Football League</td>
</tr>
<tr>
<td>4</td>
<td>Sports</td>
<td>NBA</td>
<td>National Basketball Association</td>
</tr>
<tr>
<td>5</td>
<td>Sports</td>
<td>MLS</td>
<td>Major League Soccer</td>
</tr>
<tr>
<td>6</td>
<td>Shows</td>
<td>Musicals</td>
<td>Musical theatre</td>
</tr>
<tr>
<td>7</td>
<td>Shows</td>
<td>Plays</td>
<td>All non-musical theatre</td>
</tr>
<tr>
<td>8</td>
<td>Shows</td>
<td>Opera</td>
<td>All opera and light opera</td>
</tr>
<tr>
<td>9</td>
<td>Shows</td>
<td>Pop</td>
<td>All rock and pop music concerts</td>
</tr>
<tr>
<td>10</td>
<td>Concerts</td>
<td>Jazz</td>
<td>All jazz singers and bands</td>
</tr>
<tr>
<td>11</td>
<td>Concerts</td>
<td>Classical</td>
<td>All symphony, concerto, and choir concerts</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(11 rows)</td>
</tr>
</tbody>
</table>

Updating a table based on a range of values

Update the CATGROUP column based on a range of values in the CATID column.

```
update category
set catgroup='Theatre'
where catid between 6 and 8;
```

```
select * from category
where catid between 6 and 8;
```

<table>
<thead>
<tr>
<th>catid</th>
<th>catgroup</th>
<th>catname</th>
<th>catdesc</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>Theatre</td>
<td>Musicals</td>
<td>Musical theatre</td>
</tr>
<tr>
<td>7</td>
<td>Theatre</td>
<td>Plays</td>
<td>All non-musical theatre</td>
</tr>
</tbody>
</table>
Updating a table based on a current value

Update the CATNAME and CATDESC columns based on their current CATGROUP value:

```sql
update category
set catdesc=default, catname='Shows'
where catgroup='Theatre';
```

```sql
select * from category
where catname='Shows';
```

<table>
<thead>
<tr>
<th>catid</th>
<th>catgroup</th>
<th>catname</th>
<th>catdesc</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>Theatre</td>
<td>Shows</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Theatre</td>
<td>Shows</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Theatre</td>
<td>Shows</td>
<td></td>
</tr>
</tbody>
</table>

In this case, the CATDESC column was set to null because no default value was defined when the table was created.

Run the following commands to set the CATEGORY table data back to the original values:

```sql
truncate category;
```

```sql
copy category from
's3://mybucket/data/category_pipe.txt'
credentials 'aws_access_key_id=<access-key-id>;aws_secret_access_key=<secret-access-key>'
delimiter '|';
```

Updating a table based on the result of a WHERE clause subquery

Update the CATEGORY table based on the result of a subquery in the WHERE clause:

```sql
update category
set catdesc='Broadway Musical'
where category.catid in
(select category.catid from category
join event on category.catid = event.catid
join venue on venue.venueid = event.venueid
join sales on sales.eventid = event.eventid
where venuecity='New York City' and catname='Musicals');
```

View the updated table:

```sql
select * from category order by 1;
```

<table>
<thead>
<tr>
<th>catid</th>
<th>catgroup</th>
<th>catname</th>
<th>catdesc</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

API Version 2012-12-01

409
Updating a table based on the result of a join condition

Update the original 11 rows in the CATEGORY table based on matching CATID rows in the EVENT table:

```
update category set catid=100
from event
where event.catid=category.catid;
```

Select all columns from the CATEGORY table ordered by catid:

```
select * from category order by 1;
```

<table>
<thead>
<tr>
<th>catid</th>
<th>catgroup</th>
<th>catname</th>
<th>catdesc</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Sports</td>
<td>MLB</td>
<td>Major League Baseball</td>
</tr>
<tr>
<td>2</td>
<td>Sports</td>
<td>NHL</td>
<td>National Hockey League</td>
</tr>
<tr>
<td>3</td>
<td>Sports</td>
<td>NFL</td>
<td>National Football League</td>
</tr>
<tr>
<td>4</td>
<td>Sports</td>
<td>NBA</td>
<td>National Basketball Association</td>
</tr>
<tr>
<td>5</td>
<td>Sports</td>
<td>MLS</td>
<td>Major League Soccer</td>
</tr>
<tr>
<td>10</td>
<td>Concerts</td>
<td>Jazz</td>
<td>All jazz singers and bands</td>
</tr>
<tr>
<td>11</td>
<td>Concerts</td>
<td>Classical</td>
<td>All symphony, concerto, and choir concerts</td>
</tr>
<tr>
<td>100</td>
<td>Shows</td>
<td>Opera</td>
<td>All opera and light opera</td>
</tr>
<tr>
<td>100</td>
<td>Shows</td>
<td>Musicals</td>
<td>Musical theatre</td>
</tr>
<tr>
<td>100</td>
<td>Concerts</td>
<td>Pop</td>
<td>All rock and pop music concerts</td>
</tr>
<tr>
<td>100</td>
<td>Shows</td>
<td>Plays</td>
<td>All non-musical theatre</td>
</tr>
</tbody>
</table>

Note that the EVENT table is listed in the FROM clause and the join condition to the target table is defined in the WHERE clause. Only four rows qualified for the update. These four rows are the rows whose CATID values were originally 6, 7, 8, and 9; only those four categories are represented in the EVENT table:

```
select distinct catid from event;
catid  
-------
9
8
6
7
```

Update the original 11 rows in the CATEGORY table by extending the previous example and adding another condition to the WHERE clause. Because of the restriction on the CATGROUP column, only one row qualifies for the update (although four rows qualify for the join).
update category set catid=100
from event
where event.catid=category.catid
and catgroup='Concerts';

select * from category where catid=100;
catid | catgroup | catname | catdesc
-------+----------+---------+---------------------------------
100 | Concerts | Pop     | All rock and pop music concerts
(1 row)

An alternative way to write this example is as follows:

update category set catid=100
from event join category cat on event.catid=cat.catid
where cat.catgroup='Concerts';

The advantage to this approach is that the join criteria are clearly separated from any other criteria that qualify rows for the update. Note the use of the alias CAT for the CATEGORY table in the FROM clause.

**Updates with outer joins in the FROM clause**

The previous example showed an inner join specified in the FROM clause of an UPDATE statement. The following example returns an error because the FROM clause does not support outer joins to the target table:

update category set catid=100
from event left join category cat on event.catid=cat.catid
where cat.catgroup='Concerts';

ERROR: Target table must be part of an equijoin predicate

If the outer join is required for the UPDATE statement, you can move the outer join syntax into a subquery:

update category set catid=100
from (select event.catid from event left join category cat on event.catid=cat.catid)
eventcat
where category.catid=eventcat.catid
and catgroup='Concerts';

**VACUUM**

Reclaims space and/or resorts rows in either a specified table or all tables in the current database.

**Note**

The Amazon Redshift VACUUM command syntax and behavior are substantially different from the PostgreSQL VACUUM operation. For example, the default VACUUM operation in Amazon Redshift is VACUUM FULL, which reclaims disk space and resorts all rows; however, the default VACUUM operation in PostgreSQL simply reclaims space and makes it available for re-use.

See Vacuuming tables (p. 161).
Synopsis

VACUUM [ FULL | SORT ONLY | DELETE ONLY ] [ table_name ]

Parameters

FULL
  Sorts the specified table (or all tables in the current database) and reclaims disk space occupied by rows that were marked for deletion by previous UPDATE and DELETE operations.

SORT ONLY
  Sorts the specified table (or all tables in the current database) without reclaiming space freed by deleted rows. This option is useful when reclaiming disk space is not important but resorting new rows is important. A SORT ONLY vacuum reduces the elapsed time for vacuum operations when the unsorted region does not contain a large number of deleted rows and does not span the entire sorted region. Applications that do not have disk space constraints but do depend on query optimizations associated with keeping table rows sorted can benefit from this kind of vacuum.

DELETE ONLY
  Reclaims disk space occupied by rows that were marked for deletion by previous UPDATE and DELETE operations, and compacts the table to free up the consumed space. A DELETE ONLY vacuum operation does not sort the data. This option reduces the elapsed time for vacuum operations when reclaiming disk space is important but resorting new rows is not important. This option might also apply when your query performance is already optimal, such that resorting rows to optimize query performance is not a requirement.

table_name
  Names a database table that you want to vacuum. If you do not specify a table name, the vacuum operation applies to all tables in the current database. You can specify any permanent or temporary user-created table. The command is not meaningful for other objects, such as views and system tables.

Usage notes

Note
  Only the table owner or a superuser can effectively vacuum a table. If VACUUM is run without the necessary table privileges, the operation completes successfully but has no effect.

For most Amazon Redshift applications, a full vacuum is recommended. See Vacuuming tables (p. 161).

Note the following behavior before running vacuum operations:

- Only one VACUUM command can be run at any given time. If you attempt to run multiple vacuum operations concurrently, Amazon Redshift returns an error.
- Some amount of table growth may occur when tables are vacuumed; this is expected behavior when there are no deleted rows to reclaim or the new sort order of the table results in a lower ratio of data compression.
- During vacuum operations, some degree of query performance degradation is expected. Normal performance resumes as soon as the vacuum operation is complete.
- Concurrent write operations proceed during vacuum operations but are not recommended. It is more efficient to complete write operations before running the vacuum. Also, any data that is written after a vacuum operation has been started cannot be vacuumed by that operation; therefore, a second vacuum operation will be necessary.
- A vacuum operation might not be able to start if a load or insert operation is already in progress. Vacuum operations temporarily require exclusive access to tables in order to start. This exclusive access is required briefly, so vacuum operations do not block concurrent loads and inserts for any significant period of time.
• Vacuum operations are skipped when there is no work to do for a particular table; however, there is some overhead associated with discovering that the operation can be skipped. If you know that a table is pristine, do not run a vacuum operation against it.
• A DELETE ONLY vacuum operation on a small table might not reduce the number of blocks used to store the data, especially when the table has a large number of columns or the cluster uses a large number of slices per node. These vacuum operations add one block per column per slice to account for concurrent inserts into the table, and there is potential for this overhead to outweigh the reduction in block count from the reclaimed disk space. For example, if a 10-column table on an 8-node cluster occupies 1000 blocks before a vacuum, the vacuum will not reduce the actual block count unless more than 80 blocks of disk space are reclaimed because of deleted rows. (Each data block uses 1 MB.)

Examples
Reclaim space and resort rows in all tables in the current database (TICKIT):

```sql
vacuum;
```

Reclaim space and resort rows in the SALES table:

```sql
vacuum sales;
```

Resort rows in the SALES table:

```sql
vacuum sort only sales;
```

Reclaim space in the SALES table:

```sql
vacuum delete only sales;
```

SQL Functions Reference

Topics
• Leader-node only functions (p. 414)
• Aggregate functions (p. 415)
• Bit-wise aggregate functions (p. 424)
• Window functions (p. 428)
• Conditional expressions (p. 465)
• Date functions (p. 471)
• Math functions (p. 494)
• String functions (p. 518)
• JSON Functions (p. 553)
• Data type formatting functions (p. 556)
• System administration functions (p. 565)
• System information functions (p. 568)

Amazon Redshift supports a number of functions that are extensions to the SQL standard, as well as standard aggregate functions, scalar functions, and window functions.
Note
Amazon Redshift is based on PostgreSQL 8.0.2. Amazon Redshift and PostgreSQL have a number of very important differences that you must be aware of as you design and develop your data warehouse applications. For more information about how Amazon Redshift SQL differs from PostgreSQL, see Amazon Redshift and PostgreSQL (p. 206).

Leader-node only functions

Some Amazon Redshift queries are distributed and executed on the compute nodes, other queries execute exclusively on the leader node.

The leader node distributes SQL to the compute nodes when a query references user-created tables or system tables (tables with an STL or STV prefix and system views with an SVL or SVV prefix). A query that references only catalog tables (tables with a PG prefix, such as PG_TABLE_DEF) or that does not reference any tables, runs exclusively on the leader node.

Some Amazon Redshift SQL functions are supported only on the leader node and are not supported on the compute nodes. A query that uses a leader-node function must execute exclusively on the leader node, not on the compute nodes, or it will return an error.

The documentation for each leader-node only function includes a note stating that the function will return an error if it references user-defined tables or Amazon Redshift system tables.

For more information, see SQL functions supported on the leader node (p. 205).

The following SQL functions are leader-node only functions and are not supported on the compute nodes:

System information functions

• CURRENT_SCHEMA
• CURRENT_SCHEMAS
• HAS_DATABASE_PRIVILEGE
• HAS_SCHEMA_PRIVILEGE
• HAS_TABLE_PRIVILEGE

The following leader-node only functions are deprecated:

Date functions

• AGE
• CURRENT_TIME
• CURRENT_TIMESTAMP
• LOCALTIME
• ISFINITE
• NOW

String functions

• ASCII
• GET_BIT
• GET_BYTE
• OCTET_LENGTH
• SET_BIT
• SET_BYTE
Aggregate functions

Topics

- AVG function (p. 415)
- COUNT function (p. 417)
- MAX function (p. 418)
- MIN function (p. 419)
- STDDEV_SAMP and STDDEV_POP functions (p. 420)
- SUM function (p. 421)
- VAR_SAMP and VAR_POP functions (p. 423)

Aggregate functions compute a single result value from a set of input values. The aggregate functions supported by Amazon Redshift include AVG, COUNT, MAX, MIN, and SUM.

SELECT statements using any of these aggregate functions can include two optional clauses: GROUP BY and HAVING. The syntax for these clauses is as follows (using the COUNT function as an example):

```
SELECT count (*) expression FROM table_reference
WHERE condition [GROUP BY expression ] [ HAVING condition]
```

The GROUP BY clause aggregates and groups results by the unique values in a specified column or columns. The HAVING clause restricts the results returned to rows where a particular aggregate condition is true, such as count (*) > 1. The HAVING clause is used in the same way as WHERE to restrict rows based on the value of a column.

See the COUNT function description for an example of these additional clauses.

**AVG function**

The AVG function returns the average (arithmetic mean) of the input expression values. The AVG function works with numeric values and ignores NULL values.

**Synopsis**

```
AVG ( [ DISTINCT | ALL ] expression )
```

**Arguments**

- `expression` The target column or expression that the function operates on.
- `DISTINCT | ALL`
  
  With the argument DISTINCT, the function eliminates all duplicate values from the specified expression before calculating the average. With the argument ALL, the function retains all duplicate values from the expression for calculating the average. ALL is the default.

**Data types**

The argument types supported by the AVG function are SMALLINT, INTEGER, BIGINT, NUMERIC, DECIMAL, REAL, and DOUBLE PRECISION.
The return types supported by the AVG function are:

- NUMERIC for any integer type argument
- DOUBLE PRECISION for a floating point argument

The default precision for an AVG function result with a 64-bit NUMERIC or DECIMAL argument is 19. The default precision for a result with a 128-bit NUMERIC or DECIMAL argument is 38. The scale of the result is the same as the scale of the argument. For example, an AVG of a DEC(5,2) column returns a DEC(19,2) data type.

**Examples**

Find the average quantity sold per transaction from the SALES table:

```sql
select avg(qtysold) from sales;
```

```
avg
-----
2
(1 row)
```

Find the average total price listed for all listings:

```sql
select avg(numtickets*priceperticket) as avg_total_price from listing;
```

```
avg_total_price
-----------------
3034.41
(1 row)
```

Find the average price paid, grouped by month in descending order:

```sql
select avg(pricepaid) as avg_price, month
from sales, date
where sales.dateid = date.dateid
order by avg_price desc;
```

```
  avg_price | month
-----------+-------
  659.34 | MAR
  655.06 | APR
  645.82 | JAN
  643.10 | MAY
  642.72 | JUN
  642.37 | SEP
  640.72 | OCT
  640.57 | DEC
  635.34 | JUL
  635.24 | FEB
  634.24 | NOV
  632.78 | AUG
(12 rows)
```
COUNT function

The COUNT function counts the rows defined by the expression.

The COUNT function has three variations. COUNT ( * ) counts all the rows in the target table whether they include nulls or not. COUNT ( expression ) computes the number of rows with non-NULL values in a specific column or expression. COUNT ( DISTINCT expression ) computes the number of distinct non-NULL values in a column or expression.

Synopsis

[ APPROXIMATE ] COUNT ( [ DISTINCT | ALL ] * | expression )

Arguments

expression

The target column or expression that the function operates on.

DISTINCT | ALL

With the argument DISTINCT, the function eliminates all duplicate values from the specified expression before doing the count. With the argument ALL, the function retains all duplicate values from the expression for counting. ALL is the default.

APPROXIMATE

When used with APPROXIMATE, a COUNT ( DISTINCT expression ) function uses a HyperLogLog algorithm to approximate the number of distinct non-NULL values in a column or expression. Queries that use the APPROXIMATE keyword execute much faster, with a low relative error of around 2%. Approximation is warranted for queries that return a large number of distinct values, in the millions or more per query, or per group, if there is a group by clause. For smaller sets of distinct values, in the thousands, approximation might be slower than a precise count. APPROXIMATE can only be used with COUNT ( DISTINCT ).

Data Types

The COUNT function supports all argument data types.

The COUNT function returns BIGINT.

Examples

Count all of the users from the state of Florida:

```
select count (*) from users where state='FL';
```

```
count
-------
510
```

Count all of the unique venue IDs from the EVENT table:

```
select count (distinct venueid) as venues from event;
```

```
venues
-------
```

Count the number of times each seller listed batches of more than four tickets for sale. Group the results by seller ID:

```
select count(*), sellerid from listing
group by sellerid
having min(numtickets)>4
order by 1 desc, 2;
```

<table>
<thead>
<tr>
<th>count</th>
<th>sellerid</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>17304</td>
</tr>
<tr>
<td>11</td>
<td>25428</td>
</tr>
<tr>
<td>11</td>
<td>48950</td>
</tr>
<tr>
<td>11</td>
<td>49585</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>(16840 rows)</td>
<td></td>
</tr>
</tbody>
</table>

**MAX function**

The MAX function returns the maximum value in a set of rows. DISTINCT or ALL may be used but do not affect the result.

**Synopsis**

```
MAX ( [ DISTINCT | ALL ] expression )
```

**Arguments**

*expression*

The target column or expression that the function operates on.

DISTINCT | ALL

With the argument DISTINCT, the function eliminates all duplicate values from the specified expression before calculating the maximum. With the argument ALL, the function retains all duplicate values from the expression for calculating the maximum. ALL is the default.

**Data Types**

The argument types supported by the MAX function are:

- Numeric - SMALLINT, INTEGER, BIGINT, NUMERIC, DECIMAL, REAL, DOUBLE PRECISION
- String - CHAR, VARCHAR, TEXT
- Datetime - DATE, TIMESTAMP

The return type supported by the MAX function is the same as the argument type.

**Examples**

Find the highest price paid from all sales:
Find the highest price paid per ticket from all sales:

```
select max(pricepaid/qtysold) as max_ticket_price
from sales;
```

```
max_ticket_price
-----------------
2500.00000000
(1 row)
```

**MIN function**

The MIN function returns the minimum value in a set of rows. DISTINCT or ALL may be used but do not affect the result.

**Synopsis**

```
MIN ( [ DISTINCT | ALL ] expression )
```

**Arguments**

- **expression**
  - The target column or expression that the function operates on.
- **DISTINCT | ALL**
  - With the argument DISTINCT, the function eliminates all duplicate values from the specified expression before calculating the minimum. With the argument ALL, the function retains all duplicate values from the expression for calculating the minimum. ALL is the default.

**Data Types**

The argument types supported by the MIN function are:

- Numeric - SMALLINT, INTEGER, BIGINT, NUMERIC, DECIMAL, REAL, DOUBLE PRECISION
- String - CHAR, VARCHAR
- Datetime - DATE, TIMESTAMP

The return type supported by the MIN function is the same as the argument type.

**Examples**

Find the lowest price paid from all sales:

```
select min(pricepaid) from sales;
```
Find the lowest price paid per ticket from all sales:

```sql
select min(pricepaid/qtysold) as min_ticket_price
from sales;
```

```
min_ticket_price
------------------
20.00000000
(1 row)
```

**STDDEV_SAMP and STDDEV_POP functions**

The STDDEV_SAMP and STDDEV_POP functions return the sample and population standard deviation of a set of numeric values (integer, decimal, or floating-point). The result of the STDDEV_SAMP function is equivalent to the square root of the sample variance of the same set of values. STDDEV_SAMP and STDDEV are synonyms for the same function.

STDDEV_SAMP and STDDEV are synonyms for the same function.

**Syntax**

```
STDDEV_SAMP | STDDEV ( [ DISTINCT | ALL ] expression)
STDDEV_POP ( [ DISTINCT | ALL ] expression)
```

The expression must have an integer, decimal, or floating-point data type. Regardless of the data type of the expression, the return type of this function is a double precision number.

**Note**

Standard-deviation is calculated using floating-point arithmetic, which might result in slight imprecision.

**Usage notes**

When the sample standard deviation (STDDEV or STDDEV_SAMP) is calculated for an expression that consists of a single value, the result of the function is NULL not 0.

**Examples**

The following query returns the average of the values in the VENUESEATS column of the VENUE table, followed by the sample standard deviation and population standard deviation of the same set of values. VENUESEATS is an INTEGER column. The scale of the result is reduced to 2 digits.

```sql
select avg(venueseats),
cast(stddev_samp(venueseats) as dec(14,2)) stddevsamp,
cast(stddev_pop(venueseats) as dec(14,2)) stddevpop
from venue;
```

```
avg  | stddevsamp | stddevpop
-------+------------+-----------
257   | 1.625531915 | 1.625531915
(1 row)
```
The following query returns the sample standard deviation for the COMMISSION column in the SALES table. COMMISSION is a DECIMAL column. The scale of the result is reduced to 10 digits.

```
select cast(stddev(commission) as dec(18,10))
from sales;
```

```
<table>
<thead>
<tr>
<th>stddev</th>
</tr>
</thead>
<tbody>
<tr>
<td>130.3912659086</td>
</tr>
</tbody>
</table>
```

(1 row)

The following query casts the sample standard deviation for the COMMISSION column as an integer.

```
select cast(stddev(commission) as integer)
from sales;
```

```
<table>
<thead>
<tr>
<th>stddev</th>
</tr>
</thead>
<tbody>
<tr>
<td>130</td>
</tr>
</tbody>
</table>
```

(1 row)

The following query returns both the sample standard deviation and the square root of the sample variance for the COMMISSION column. The results of these calculations are the same.

```
select
  cast(stddev_samp(commission) as dec(18,10)) stddevsamp,
  cast(sqrt(var_samp(commission)) as dec(18,10)) sqrtvarsamp
from sales;
```

```
<table>
<thead>
<tr>
<th>stddevsamp</th>
<th>sqrtvarsamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>130.3912659086</td>
<td>130.3912659086</td>
</tr>
</tbody>
</table>
```

(1 row)

**SUM function**

The SUM function returns the sum of the input column or expression values. The SUM function works with numeric values and ignores NULL values.

**Synopsis**

```
SUM ( [ DISTINCT | ALL ] expression )
```

**Arguments**

- `expression`
  - The target column or expression that the function operates on.
DISTINCT | ALL
With the argument DISTINCT, the function eliminates all duplicate values from the specified expression
before calculating the sum. With the argument ALL, the function retains all duplicate values from the
expression for calculating the sum. ALL is the default.

Data types
The argument types supported by the SUM function are SMALLINT, INTEGER, BIGINT, NUMERIC,
DECIMAL, REAL, and DOUBLE PRECISION.

The return types supported by the SUM function are
• BIGINT for BIGINT, SMALLINT, and INTEGER arguments
• NUMERIC for NUMERIC arguments
• DOUBLE PRECISION for floating point arguments

The default precision for a SUM function result with a 64-bit NUMERIC or DECIMAL argument is 19. The
default precision for a result with a 128-bit NUMERIC or DECIMAL argument is 38. The scale of the result
is the same as the scale of the argument. For example, a SUM of a DEC(5,2) column returns a DEC(19,2)
data type.

Examples
Find the sum of all commissions paid from the SALES table:

```
select sum(commission) from sales;
```

```
sum
------------
16614814.65
(1 row)
```

Find the number of seats in all venues in the state of Florida:

```
select sum(venueseats) from venue
where venuestate = 'FL';
```

```
sum
-------
250411
(1 row)
```

Find the number of seats sold in May:

```
select sum(qtysold) from sales, date
where sales.dateid = date.dateid and date.month = 'MAY';
```

```
sum
-------
32291
(1 row)
```
VAR_SAMP and VAR_POP functions

The VAR_SAMP and VAR_POP functions return the sample and population variance of a set of numeric values (integer, decimal, or floating-point). The result of the VAR_SAMP function is equivalent to the squared sample standard deviation of the same set of values.

VAR_SAMP and VARIANCE are synonyms for the same function.

Syntax

```
VAR_SAMP | VARIANCE ( [ DISTINCT | ALL ] expression)
VAR_POP ( [ DISTINCT | ALL ] expression)
```

The expression must have an integer, decimal, or floating-point data type. Regardless of the data type of the expression, the return type of this function is a double precision number.

**Note**

The results of these functions might vary across data warehouse clusters, depending on the configuration of the cluster in each case.

Usage notes

When the sample variance (VARIANCE or VAR_SAMP) is calculated for an expression that consists of a single value, the result of the function is NULL not 0.

Examples

The following query returns the rounded sample and population variance of the NUMTICKETS column in the LISTING table.

```
select avg(numtickets),
       round(var_samp(numtickets)) varsamp,
       round(var_pop(numtickets)) varpop
from listing;
```

```
<table>
<thead>
<tr>
<th>avg</th>
<th>varsamp</th>
<th>varpop</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>54</td>
<td>54</td>
</tr>
</tbody>
</table>
```

(1 row)

The following query runs the same calculations but cast the results to decimal values.

```
select avg(numtickets),
       cast(var_samp(numtickets) as dec(10,4)) varsamp,
       cast(var_pop(numtickets) as dec(10,4)) varpop
from listing;
```

```
<table>
<thead>
<tr>
<th>avg</th>
<th>varsamp</th>
<th>varpop</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>53.6291</td>
<td>53.6288</td>
</tr>
</tbody>
</table>
```

(1 row)
Bit-wise aggregate functions

Topics
- BIT_AND and BIT_OR (p. 424)
- BOOL_AND and BOOL_OR (p. 425)
- NULLs in bit-wise aggregations (p. 425)
- DISTINCT support for bit-wise aggregations (p. 425)
- BIT_AND function (p. 425)
- BIT_OR function (p. 426)
- BOOL_AND function (p. 426)
- BOOL_OR function (p. 426)
- Bit-wise function examples (p. 427)

Amazon Redshift supports the following bit-wise aggregate functions:
- BIT_AND
- BIT_OR
- BOOL_AND
- BOOL_OR

BIT_AND and BIT_OR

The BIT_AND and BIT_OR functions run bit-wise AND and OR operations on all of the values in a single integer column or expression. These functions aggregate each bit of each binary value that corresponds to each integer value in the expression.

The BIT_AND function returns a result of 0 if none of the bits is set to 1 across all of the values. If one or more bits is set to 1 across all values, the function returns an integer value. This integer is the number that corresponds to the binary value for those bits.

For example, a table contains four integer values in a column: 3, 7, 10, and 22. These integers are represented in binary form as follows:

<table>
<thead>
<tr>
<th>Integer</th>
<th>Binary value</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>0000 0011</td>
</tr>
<tr>
<td>7</td>
<td>0000 0111</td>
</tr>
<tr>
<td>10</td>
<td>0000 1010</td>
</tr>
<tr>
<td>22</td>
<td>0001 0110</td>
</tr>
</tbody>
</table>

A BIT_AND operation on this data set finds that all bits are set to 1 in the second-to-last position only. The result is a binary value of 0000 0010, which represents the integer value 2; therefore, the BIT_AND function returns 2.

If you apply the BIT_OR function to the same set of integer values, the operation looks for any value in which a 1 is found in each position. In this case, a 1 exists in the last five positions for at least one of the values, yielding a binary result of 0001 1111; therefore, the function returns 31 (or $16 + 8 + 4 + 2 + 1$).
BOOL_AND and BOOL_OR

The BOOL_AND and BOOL_OR functions operate on a single Boolean or integer column or expression. These functions apply similar logic to the BIT_AND and BIT_OR functions. For these functions, the return type is a Boolean value (true or false):

- If all values in a set are true, the BOOL_AND function returns true(t). If all values are false, the function returns false(f).
- If any value in a set is true, the BOOL_OR function returns true(t). If no value in a set is true, the function returns false(f).

NULLs in bit-wise aggregations

When a bit-wise function is applied to a column that is nullable, any NULL values are eliminated before the function result is calculated. If no rows qualify for aggregation, the bit-wise function returns NULL. The same behavior applies to regular aggregate functions. For example:

```sql
select sum(venueseats), bit_and(venueseats) from venue
where venueseats is null;
```

<table>
<thead>
<tr>
<th>sum</th>
<th>bit_and</th>
</tr>
</thead>
<tbody>
<tr>
<td>null</td>
<td>null</td>
</tr>
</tbody>
</table>

(1 row)

DISTINCT support for bit-wise aggregations

Like other aggregate functions, bit-wise functions support the DISTINCT keyword. However, using DISTINCT with these functions has no impact on the results. The first instance of a value is sufficient to satisfy bitwise AND or OR operations, and it makes no difference if duplicate values are present in the expression being evaluated. Because the DISTINCT processing is likely to incur some query execution overhead, do not use DISTINCT with these functions.

BIT_AND function

Synopsis

```
BIT_AND ( [DISTINCT | ALL] expression )
```

Arguments

expression

The target column or expression that the function operates on. This expression must have an INT, INT2, or INT8 data type. The function returns an equivalent INT, INT2, or INT8 data type.

DISTINCT | ALL

With the argument DISTINCT, the function eliminates all duplicate values for the specified expression before calculating the result. With the argument ALL, the function retains all duplicate values. ALL is the default. See DISTINCT support for bit-wise aggregations (p. 425).
BIT_OR function

Synopsis

```
BIT_OR ( [DISTINCT | ALL] expression )
```

Arguments

expression
The target column or expression that the function operates on. This expression must have an INT, INT2, or INT8 data type. The function returns an equivalent INT, INT2, or INT8 data type.

DISTINCT | ALL
With the argument DISTINCT, the function eliminates all duplicate values for the specified expression before calculating the result. With the argument ALL, the function retains all duplicate values. ALL is the default. See DISTINCT support for bit-wise aggregations (p. 425).

BOOL_AND function

Synopsis

```
BOOL_AND ( [DISTINCT | ALL] expression )
```

Arguments

expression
The target column or expression that the function operates on. This expression must have a BOOLEAN or integer data type. The return type of the function is BOOLEAN.

DISTINCT | ALL
With the argument DISTINCT, the function eliminates all duplicate values for the specified expression before calculating the result. With the argument ALL, the function retains all duplicate values. ALL is the default. See DISTINCT support for bit-wise aggregations (p. 425).

BOOL_OR function

Synopsis

```
BOOL_OR ( [DISTINCT | ALL] expression )
```

Arguments

expression
The target column or expression that the function operates on. This expression must have a BOOLEAN or integer data type. The return type of the function is BOOLEAN.

DISTINCT | ALL
With the argument DISTINCT, the function eliminates all duplicate values for the specified expression before calculating the result. With the argument ALL, the function retains all duplicate values. ALL is the default. See DISTINCT support for bit-wise aggregations (p. 425).
Bit-wise function examples

The USERS table in the TICKIT sample database contains several Boolean columns that indicate whether each user is known to like different types of events, such as sports, theatre, opera, and so on. For example:

```
select userid, username, lastname, city, state, likesports, liketheatre
from users limit 10;
```

```
<table>
<thead>
<tr>
<th>userid</th>
<th>username</th>
<th>lastname</th>
<th>city</th>
<th>state</th>
<th>likesports</th>
<th>liketheatre</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>JSG99FHE</td>
<td>Taylor</td>
<td>Kent</td>
<td>WA</td>
<td>t</td>
<td>t</td>
</tr>
<tr>
<td>9</td>
<td>MSD36KVR</td>
<td>Watkins</td>
<td>Port Orford</td>
<td>MD</td>
<td>t</td>
<td>f</td>
</tr>
</tbody>
</table>
```

Assume that a new version of the USERS table is built in a different way, with a single integer column that defines (in binary form) eight types of events that each user likes or dislikes. In this design, each bit position represents a type of event, and a user who likes all eight types has all eight bits set to 1 (as in the first row of the following table). A user who does not like any of these events has all eight bits set to 0 (see second row). A user who likes only sports and jazz is represented in the third row:

```
<table>
<thead>
<tr>
<th>SPORTS</th>
<th>THEATRE</th>
<th>JAZZ</th>
<th>OPERA</th>
<th>ROCK</th>
<th>VEGAS</th>
<th>BROADWAY</th>
<th>CLASSICAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>User 1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>User 2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>User 3</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
```

In the database table, these binary values could be stored in a single LIKES column as integers:

<table>
<thead>
<tr>
<th>User</th>
<th>Binary value</th>
<th>Stored value (integer)</th>
</tr>
</thead>
<tbody>
<tr>
<td>User 1</td>
<td>11111111</td>
<td>255</td>
</tr>
<tr>
<td>User 2</td>
<td>00000000</td>
<td>0</td>
</tr>
<tr>
<td>User 3</td>
<td>10100000</td>
<td>160</td>
</tr>
</tbody>
</table>

**BIT_AND and BIT_OR examples**

Given that meaningful business information is stored in integer columns, you can use bit-wise functions to extract and aggregate that information. The following query applies the BIT_AND function to the LIKES column in a table called USERLIKES and groups the results by the CITY column.

```
select city, bit_and(likes) from userlikes group by city
order by city;
```

```
<table>
<thead>
<tr>
<th>city</th>
<th>bit_and</th>
</tr>
</thead>
<tbody>
<tr>
<td>Los Angeles</td>
<td>0</td>
</tr>
<tr>
<td>Sacramento</td>
<td>0</td>
</tr>
<tr>
<td>San Francisco</td>
<td>0</td>
</tr>
<tr>
<td>San Jose</td>
<td>64</td>
</tr>
<tr>
<td>Santa Barbara</td>
<td>192</td>
</tr>
</tbody>
</table>
```

(5 rows)
These results can be interpreted as follows:

- The integer value 192 for Santa Barbara translates to the binary value 11000000. In other words, all users in this city like sports and theatre, but not all users like any other type of event.
- The integer 64 translates to 01000000, so for users in San Jose, the only type of event that they all like is theatre.
- The values of 0 for the other three cities indicate that no "likes" are shared by all users in those cities.

If you apply the BIT_OR function to the same data, the results are as follows:

```sql
select city, bit_or(likes) from userlikes group by city
order by city;
```

<table>
<thead>
<tr>
<th>city</th>
<th>bit_or</th>
</tr>
</thead>
<tbody>
<tr>
<td>Los Angeles</td>
<td>127</td>
</tr>
<tr>
<td>Sacramento</td>
<td>255</td>
</tr>
<tr>
<td>San Francisco</td>
<td>255</td>
</tr>
<tr>
<td>San Jose</td>
<td>255</td>
</tr>
<tr>
<td>Santa Barbara</td>
<td>255</td>
</tr>
</tbody>
</table>

(5 rows)

For four of the cities listed, all of the event types are liked by at least one user (255=11111111). For Los Angeles, all of the event types except sports are liked by at least one user (127=01111111).

**BOOL_AND and BOOL_OR examples**

You can use the Boolean functions against either Boolean expressions or integer expressions. For example, the following query return results from the standard USERS table in the TICKIT database, which has several Boolean columns.

The BOOL_OR function returns true for all five rows. At least one user in each of those states likes sports. The BOOL_AND function returns false for all five rows. Not all users in each of those states likes sports.

```sql
select state, bool_or(likesports), bool_and(likesports) from users
group by state order by state limit 5;
```

<table>
<thead>
<tr>
<th>state</th>
<th>bool_or</th>
<th>bool_and</th>
</tr>
</thead>
<tbody>
<tr>
<td>AB</td>
<td>t</td>
<td>f</td>
</tr>
<tr>
<td>AK</td>
<td>t</td>
<td>f</td>
</tr>
<tr>
<td>AL</td>
<td>t</td>
<td>f</td>
</tr>
<tr>
<td>AZ</td>
<td>t</td>
<td>f</td>
</tr>
<tr>
<td>BC</td>
<td>t</td>
<td>f</td>
</tr>
</tbody>
</table>

(5 rows)

**Window functions**

**Topics**
- Window function syntax summary (p. 430)
- AVG window function (p. 432)
- COUNT window function (p. 433)
- DENSE_RANK window function (p. 434)
Window functions provide application developers the ability to create analytic business queries more efficiently. Window functions operate on a partition or “window” of a result set, and return a value for every row in that window. In contrast, non-windowed functions perform their calculations with respect to every row in the result set. Unlike group functions that aggregate result rows, all rows in the table expression are retained.

The values returned are calculated by utilizing values from the sets of rows in that window. The window defines, for each row in the table, a set of rows that is used to compute additional attributes. A window is defined using a window specification (the OVER clause), and is based on three main concepts:

- **Window partitioning**, which forms groups of rows (PARTITION clause)
- **Window ordering**, which defines an order or sequence of rows within each partition (ORDER BY clause)
- **Window frames**, which are defined relative to each row to further restrict the set of rows (ROWS specification)

Window functions are the last set of operations performed in a query except for the final ORDER BY clause. All joins and all WHERE, GROUP BY, and HAVING clauses are completed before the window functions are processed. Therefore, window functions can appear only in the select list or ORDER BY clause. Multiple window functions can be used within a single query with different frame clauses. Window functions may be present in other scalar expressions, such as CASE.

Amazon Redshift supports two types of window functions: aggregate and ranking. These are the supported aggregate functions:

- AVG
- COUNT
- FIRST_VALUE
- LAG
- LAST_VALUE
- LEAD
- MAX
- MIN
- NTH_VALUE
- STDDEV_POP

Window function examples (p. 447)
• STDDEV_SAMP (synonym for STDDEV)
• SUM
• VAR_POP
• VAR_SAMP (synonym for VARIANCE)

These are the supported ranking functions:
• DENSE_RANK
• NTILE
• RANK
• ROW_NUMBER

Aggregate window functions can have different PARTITION/ORDER BY clauses in the same query, but ranking functions can not.

Window function syntax summary

```
function (expression) OVER
{
  [ PARTITION BY expr_list ]
  [ ORDER BY order_list [ frame_clause ] ]
}
```

where `function` is one of the functions described in this section and `expr_list` is:

```
expression | column_name [, expr_list ]
```

and `order_list` is:

```
expression | column_name [ASC | DESC] [, order_list ]
```

and `frame_clause` is:

```
ROWS
  {UNBOUNDED PRECEDING | unsigned_value PRECEDING | CURRENT ROW} |
BETWEEN
  {UNBOUNDED PRECEDING | unsigned_value { PRECEDING | FOLLOWING } | CURRENT ROW}
  AND
  {UNBOUNDED FOLLOWING | unsigned_value { PRECEDING | FOLLOWING } | CURRENT ROW}
```

Note
STDDEV_SAMP and VAR_SAMP are synonyms for STDDEV and VARIANCE, respectively.

Arguments

`function`
See the individual function descriptions.
OVER

The OVER keyword is mandatory for window functions and differentiates window functions from other SQL functions.

PARTITION BY `expr_list`

The PARTITION BY clause is optional and subdivides the result set into partitions, much like the GROUP BY clause. If a partition clause is present, the function is calculated for the rows in each partition. If no partitioning clause is specified, a single partition contains the entire table, and the function is computed for that complete table.

The ranking functions, DENSE_RANK, NTILE, RANK, and ROW_NUMBER, require a global comparison of all the rows in the result set. When a PARTITION BY clause is used, the query optimizer is able to execute each aggregation in parallel by spreading the workload across multiple slices according to the partitions. If the PARTITION BY clause is not present, the aggregation step must be executed serially on a single slice, which might significantly impact performance, especially on large clusters.

ORDER BY `order_list`

The window function is applied to the rows within each partition sorted according to the order specification. This ORDER BY clause is distinct from and completely unrelated to an ORDER BY clause in a non-window function (outside of the OVER clause). The ORDER BY clause can be used without the PARTITION BY clause.

For the ranking functions, the ORDER BY clause identifies the measure(s) for the ranking values. For aggregation functions, the partitioned rows must be ordered before the aggregate function is computed for each frame.

Column identifiers or expressions that evaluate to column identifiers are required in the order list. Neither constants nor constant expressions can be used as substitutes for column names.

NULLS are treated as their own group, sorted and ranked last in ASC, and sorted and ranked first in DESC.

**Note**

In any parallel system such as Amazon Redshift, when an ORDER BY clause does not produce a unique and total ordering of the data, the order of the rows is non-deterministic. That is, if the ORDER BY expression produces duplicate values (a partial ordering), the return order of those rows may vary from one run of Amazon Redshift to the next. In turn, window functions may return unexpected or inconsistent results. See Unique ordering of data for window functions (p. 464).

`column_name`

The name of a column to be partitioned by or ordered by.

ASC | DESC

Specifies whether to sort ascending or descending. Ascending is the default.

`frame_clause`

For aggregate functions, the frame clause further refines the set of rows in a function's window when using ORDER BY. It provides the ability to include or exclude sets of rows within the ordered result. The frame clause consists of the ROWS keyword and associated specifiers.

The frame clause does not apply to ranking functions and is not required when no ORDER BY clause is used in the OVER clause for an aggregate function. If an ORDER BY clause is used for an aggregate function, an explicit frame clause is required.

When no ORDER BY clause is specified, the implied frame is unbounded: equivalent to ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING.

Range-based window frames are not supported.

ROWS

The ROWS clause defines the window frame, and specifies the number of rows in the current partition before or after the current row that the value on the current row is to be combined with. ROWS utilizes...
arguments that specify row position; the reference point for all window frames is the current row. Row-based window frames are defined in Amazon Redshift as any number of rows preceding or equal to the current row. Each row becomes the current row in turn as the window frame slides forward in the partition.

The frame can be a simple set of rows up to and including the current row:

\[
\{ \text{UNBOUNDED PRECEDING} \mid \text{unsigned-value PRECEDING} \mid \text{CURRENT ROW} \}
\]

or it can be a set of rows between two boundaries:

\[
\text{BETWEEN} \\
\{ \text{UNBOUNDED PRECEDING} \mid \text{unsigned-value \{ PRECEDING | FOLLOWING \}} \mid \text{CURRENT ROW} \} \\
\text{AND} \\
\{ \text{UNBOUNDED FOLLOWING} \mid \text{unsigned-value \{ PRECEDING | FOLLOWING \}} \mid \text{CURRENT ROW} \}
\]

UNBOUNDED PRECEDING indicates that the window starts at the first row of the partition; unsigned_value PRECEDING means some number of rows before; and CURRENT ROW indicates the window begins or ends at the current row. UNBOUNDED PRECEDING is the default and is implied when not stated.

UNBOUNDED FOLLOWING specifies an ending boundary at the last row of the partition; unsigned_value FOLLOWING specifies a starting row-based boundary at n rows after the current row.

**Note**
You cannot specify a frame in which the starting boundary is greater than the ending boundary. For example, you cannot specify any of these frames:

\[
\begin{align*}
\text{between 5 following and 5 preceding} \\
\text{between current row and 2 preceding} \\
\text{between 3 following and current row}
\end{align*}
\]

Range-based window frames are not supported.

## AVG window function

The AVG window function returns the average (arithmetic mean) of the input expression values. The AVG function works with numeric values and ignores NULL values.

### Synopsis

\[
\text{AVG} \{ \text{[ALL] } \text{expression} \} \ \text{OVER} \ \\ 
\{ \text{[PARTITION BY expr_list]} \ \\
\text{[ORDER BY order_list]} \ \\
\quad \text{frame_clause} \ \\
\}
\]
Arguments

*expression*

The target column or expression that the function operates on.

**ALL**

With the argument ALL, the function retains all duplicate values from the expression for counting. ALL is the default. DISTINCT is not supported.

**OVER**

Specifies the window clauses for the aggregation functions. The OVER clause distinguishes window aggregation functions from normal set aggregation functions.

**PARTITION BY expr_list**

Defines the window for the AVG function in terms of one or more expressions.

**ORDER BY order_list**

Sorts the rows within each partition. If no PARTITION BY is specified, ORDER BY uses the entire table.

**frame_clause**

If an ORDER BY clause is used for an aggregate function, an explicit frame clause is required. The frame clause refines the set of rows in a function’s window, including or excluding sets of rows within the ordered result. The frame clause consists of the ROWS keyword and associated specifiers. See Window function syntax summary (p. 430).

Data types

The argument types supported by the AVG function are SMALLINT, INTEGER, BIGINT, NUMERIC, DECIMAL, REAL, and DOUBLE PRECISION.

The return types supported by the AVG function are:

- BIGINT for SMALLINT or INTEGER arguments
- NUMERIC for BIGINT arguments
- DOUBLE PRECISION for floating point arguments

Examples

See AVG window function examples (p. 449).

COUNT window function

The COUNT window function counts the rows defined by the expression.

The COUNT function has two variations. COUNT(*) counts all the rows in the target table whether they include nulls or not. COUNT(expression) computes the number of rows with non-NULL values in a specific column or expression.

Synopsis

```
COUNT ( * | [ ALL ] expression) OVER
{
[ PARTITION BY expr_list ]
[ ORDER BY order_list ]
   frame_clause ]
}
```
Arguments

expression
The target column or expression that the function operates on.

ALL
With the argument ALL, the function retains all duplicate values from the expression for counting.
ALL is the default. DISTINCT is not supported.

OVER
Specifies the window clauses for the aggregation functions. The OVER clause distinguishes window aggregation functions from normal set aggregation functions.

PARTITION BY expr_list
Defines the window for the COUNT function in terms of one or more expressions.

ORDER BY order_list
Sorts the rows within each partition. If no PARTITION BY is specified, ORDER BY uses the entire table.

frame_clause
If an ORDER BY clause is used for an aggregate function, an explicit frame clause is required. The frame clause refines the set of rows in a function's window, including or excluding sets of rows within the ordered result. The frame clause consists of the ROWS keyword and associated specifiers. See Window function syntax summary (p. 430).

Data Types

The COUNT function supports all argument data types.

The return type supported by the COUNT function is BIGINT.

Examples

See COUNT window function examples (p. 449).

DENSE_RANK window function

The DENSE_RANK window function determines the rank of a value in a group of values, based on the ORDER BY expression in the OVER clause. If the optional PARTITION BY clause is present, the rankings are reset for each group of rows. Rows with equal values for the ranking criteria receive the same rank. The DENSE_RANK function differs from RANK in one respect: If two or more rows tie, there is no gap in the sequence of ranked values. For example, if two rows are ranked 1, the next rank would be 2.

You can have ranking functions with different PARTITION BY and ORDER BY clauses in the same query.

Synopsis

```
DENSE_RANK () OVER
{
[ PARTITION BY expr_list ]
ORDER BY order_list
}
```

Arguments

() The DENSE_RANK function takes no arguments, but the empty parentheses must be specified.
OVER
  Specifies the window clauses for the DENSE_RANK function.

PARTITION BY expr_list
  Defines the window for the DENSE_RANK function in terms of one or more expressions.

ORDER BY order_list
  Defines the column(s) on which the ranking values are based. If no PARTITION BY is specified, ORDER BY uses the entire table. ORDER BY is required for the DENSE_RANK function.

Data types
The return type supported by the DENSE_RANK function is INTEGER.

Examples
See DENSE_RANK window function examples (p. 450).

FIRST_VALUE and LAST_VALUE window functions
Given an ordered set of rows, FIRST_VALUE returns the value of the specified expression with respect to the first row in the window frame. The LAST_VALUE function returns the value of the expression with respect to the last row in the frame.

Synopsis

FIRST_VALUE | LAST_VALUE
( expression [ IGNORE NULLS | RESPECT NULLS ] ) OVER
( [ PARTITION BY expr_list ]
[ ORDER BY order_list ]
  frame_clause ]
)

Arguments

expression
  The target column or expression that the function operates on.

IGNORE NULLS
  When this option is used with FIRST_VALUE, the function returns the first value in the frame that is not NULL (or NULL if all values are NULL). When this option is used with LAST_VALUE, the function returns the last value in the frame that is not NULL (or NULL if all values are NULL).

RESPECT NULLS
  Indicates that Amazon Redshift should include null values in the determination of which row to use. RESPECT NULLS is supported by default if you do not specify IGNORE NULLS.

OVER
  Introduces the window clauses for the function.

PARTITION BY expr_list
  Defines the window for the function in terms of one or more expressions.

ORDER BY order_list
  Sorts the rows within each partition. If no PARTITION BY clause is specified, ORDER BY sorts the entire table. If you specify an ORDER BY clause, you must also specify a frame_clause.

  The results of the FIRST_VALUE and LAST_VALUE functions depend on the ordering of the data. The results are non-deterministic in the following cases:
- When no ORDER BY clause is specified and a partition contains two different values for an expression
- When the expression evaluates to different values that correspond to the same value in the ORDER BY list

**frame_clause**

If an ORDER BY clause is used for an aggregate function, an explicit frame clause is required. The frame clause refines the set of rows in a function's window, including or excluding sets of rows in the ordered result. The frame clause consists of the ROWS keyword and associated specifiers. See Window function syntax summary (p. 430).

**Data types**

These functions support expressions that use any of the Amazon Redshift data types. The return type is the same as the type of the *expression*.

**Examples**

See FIRST_VALUE and LAST_VALUE window function examples (p. 451).

**LAG window function**

The LAG window function returns the values for a row at a given offset above (before) the current row in the partition.

**Synopsis**

```
LAG (value_expr [, offset ])
[ IGNORE NULLS | RESPECT NULLS ]
OVER ( [ PARTITION BY window_partition ] ORDER BY window_ordering )
```

**Arguments**

- **value_expr**
  - The target column or expression that the function operates on.
- **offset**
  - An optional parameter that specifies the number of rows before the current row to return values for.
  - The offset can be a constant integer or an expression that evaluates to an integer. If you do not specify an offset, Amazon Redshift uses 1 as the default value. An offset of 0 indicates the current row.
- **IGNORE NULLS**
  - An optional specification that indicates that Amazon Redshift should skip null values in the determination of which row to use. Null values are included if IGNORE NULLS is not listed.
  
  **Note**
  - You can use an NVL or COALESCE expression to replace the null values with another value. For more information, see NVL expression (p. 469).
- **RESPECT NULLS**
  - Indicates that Amazon Redshift should include null values in the determination of which row to use.
  - RESPECT NULLS is supported by default if you do not specify IGNORE NULLS.
- **OVER**
  - Specifies the window partitioning and ordering. The OVER clause cannot contain a window frame specification.
- **PARTITION BY**
  - An optional argument that sets the range of records for each group in the OVER clause.
ORDER BY window_ordering
   Sorts the rows within each partition.

The LAG window function supports expressions that use any of the Amazon Redshift data types. The return type is the same as the type of the value_expr.

**Examples**

See LAG window function examples (p. 453).

**LEAD window function**

The LEAD window function returns the values for a row at a given offset below (after) the current row in the partition.

**Synopsis**

```
LEAD (value_expr [, offset ])  
[ IGNORE NULLS | RESPECT NULLS ]
OVER ([ PARTITION BY window_partition ] ORDER BY window_ordering )
```

**Arguments**

- **value_expr**
  - The target column or expression that the function operates on.

- **offset**
  - An optional parameter that specifies the number of rows below the current row to return values for. The offset can be a constant integer or an expression that evaluates to an integer. If you do not specify an offset, Amazon Redshift uses 1 as the default value. An offset of 0 indicates the current row.

- **IGNORE NULLS**
  - An optional specification that indicates that Amazon Redshift should skip null values in the determination of which row to use. Null values are included if IGNORE NULLS is not listed.

  **Note**
  You can use an NVL or COALESCE expression to replace the null values with another value. For more information, see NVL expression (p. 469).

- **RESPECT NULLS**
  - Indicates that Amazon Redshift should include null values in the determination of which row to use. RESPECT NULLS is supported by default if you do not specify IGNORE NULLS.

- **OVER**
  - Specifies the window partitioning and ordering. The OVER clause cannot contain a window frame specification.

- **PARTITION BY window_partition**
  - An optional argument that sets the range of records for each group in the OVER clause.

- **ORDER BY window_ordering**
  - Sorts the rows within each partition.

The LEAD window function supports expressions that use any of the Amazon Redshift data types. The return type is the same as the type of the value_expr.

**Examples**

See LEAD window function examples (p. 454).
MAX window function

The MAX window function returns the maximum of the input expression values. The MAX function works with numeric values and ignores NULL values.

Synopsis

```
MAX ( [ ALL ] expression ) OVER

{ [ PARTITION BY expr_list ]
  [ ORDER BY order_list
    frame_clause ]
}
```

Arguments

- **expression**: The target column or expression that the function operates on.
- **ALL**: With the argument ALL, the function retains all duplicate values from the expression. ALL is the default. DISTINCT is not supported.
- **OVER**: Specifies the window clauses for the aggregation functions. The OVER clause distinguishes window aggregation functions from normal set aggregation functions.
- **PARTITION BY expr_list**: Defines the window for the MAX function in terms of one or more expressions.
- **ORDER BY order_list**: Sorts the rows within each partition. If no PARTITION BY is specified, ORDER BY uses the entire table.
- **frame_clause**: If an ORDER BY clause is used for an aggregate function, an explicit frame clause is required. The frame clause refines the set of rows in a function's window, including or excluding sets of rows within the ordered result. The frame clause consists of the ROWS keyword and associated specifiers. See Window function syntax summary (p. 430).

Data types

- Numeric: SMALLINT, INTEGER, BIGINT, NUMERIC, DECIMAL, REAL, DOUBLE PRECISION
- String: CHAR, VARCHAR
- Datetime: DATE, TIMESTAMP

The return type supported by the MAX function is the same as the argument type.

Examples

See MAX window function examples (p. 454).

MIN window function

The MIN window function returns the minimum of the input expression values. The MIN function works with numeric values and ignores NULL values.
Synopsis

\[
\text{MIN ( [ ALL ] expression ) OVER ( [ PARTITION BY expr_list ] [ ORDER BY order_list ] frame_clause )}
\]

Arguments

expression
  The target column or expression that the function operates on.
ALL
  With the argument ALL, the function retains all duplicate values from the expression. ALL is the default. DISTINCT is not supported.
OVER
  Specifies the window clauses for the aggregation functions. The OVER clause distinguishes window aggregation functions from normal set aggregation functions.
PARTITION BY expr_list
  Defines the window for the MIN function in terms of one or more expressions.
ORDER BY order_list
  Sorts the rows within each partition. If no PARTITION BY is specified, ORDER BY uses the entire table.
frame_clause
  If an ORDER BY clause is used for an aggregate function, an explicit frame clause is required. The frame clause refines the set of rows in a function's window, including or excluding sets of rows within the ordered result. The frame clause consists of the ROWS keyword and associated specifiers. See Window function syntax summary (p. 430).

Data types

- Numeric: SMALLINT, INTEGER, BIGINT, NUMERIC, DECIMAL, REAL, DOUBLE PRECISION
- String: CHAR, VARCHAR
- Datetime: DATE, TIMESTAMP

The return type supported by the MIN function is the same as the argument type.

Examples

See MIN window function examples (p. 455).

NTH_VALUE window function

The NTH_VALUE window function returns the expression value of the specified row of the window frame relative to the first row of the window.

Synopsis

\[
\text{NTH_VALUE (expr, offset) [ IGNORE NULLS | RESPECT NULLS ] OVER ( [ PARTITION BY expr_list ] [ ORDER BY order_list ] frame_clause )}
\]
Arguments

*expr*
- The target column or expression that the function operates on.

*offset*
- Determines the row number relative to the first row in the window for which to return the expression. The *offset* can be a constant or an expression and must be a positive integer that is greater than 0.

**IGNORE NULLS**
- An optional specification that indicates that Amazon Redshift should skip null values in the determination of which row to use. Null values are included if IGNORE NULLS is not listed.

**RESPECT NULLS**
- Indicates that Amazon Redshift should include null values in the determination of which row to use. RESPECT NULLS is supported by default if you do not specify IGNORE NULLS.

**OVER**
- Specifies the window partitioning, ordering, and window frame.

**PARTITION BY** *window_partition*
- Sets the range of records for each group in the OVER clause.

**ORDER BY** *window_ordering*
- Sorts the rows within each partition. If ORDER BY is omitted, the default frame consists of all rows in the partition.

**frame_clause**
- If an ORDER BY clause is used for an aggregate function, an explicit frame clause is required. The frame clause refines the set of rows in a function's window, including or excluding sets of rows in the ordered result. The frame clause consists of the ROWS keyword and associated specifiers. See Window function syntax summary (p. 430).

The NTH_VALUE window function supports expressions that use any of the Amazon Redshift data types. The return type is the same as the type of the *expr*.

Examples

See NTH_VALUE window function examples (p. 456).

**NTILE window function**

The NTILE window function divides ordered rows in the partition into the specified number of ranked groups of as equal size as possible and returns the group that a given row falls into.

Synopsis

```sql
NTILE (expr)
OVER ( [ PARTITION BY window_partition ] ORDER BY window_ordering )
```

Arguments

*expr*
- Defines the number of ranking groups and must result in a positive integer value (greater than 0) for each partition. The *expr* argument must not be nullable.
OVER
   Specifies the window partitioning and ordering. The OVER clause cannot contain a window frame
   specification.

PARTITION BY window_partition
   An optional argument that sets the range of records for each group in the OVER clause.

ORDER BY window_ordering
   Sorts the rows within each partition.

The NTILE window function supports BIGINT argument types and returns BIGINT values.

Examples
See NTILE window function examples (p. 457).

PERCENTILE_CONT window function

PERCENTILE_CONT is an inverse distribution function that assumes a continuous distribution model. It
takes a percentile value and a sort specification, and returns an interpolated value that would fall into the
given percentile value with respect to the sort specification.

PERCENTILE_CONT computes a linear interpolation between values after ordering them. Using the
percentile value \( P \) and the number of not null rows \( N \) in the aggregation group, the function computes
the row number after ordering the rows according to the sort specification. This row number \( RN \) is
computed according to the formula \( RN = (1+ P*(N-1)) \). The final result of the aggregate function is
computed by linear interpolation between the values from rows at row numbers \( CRN = \text{CEILING}(RN) \)
and \( FRN = \text{FLOOR}(RN) \).

The final result will be as follows.

If \( CRN = FRN = RN \) then the result is (value of expression from row at RN)

Otherwise the result is as follows:

\[(CRN - RN) \times (\text{value of expression for row at FRN}) + (RN - FRN) \times (\text{value of}
   \text{expression for row at CRN})\].

You can specify only the PARTITION clause in the OVER clause. If PARTITION is specified, for each
row, PERCENTILE_CONT returns the value that would fall into the specified percentile among a set of
values within a given partition.

Syntax

```
PERCENTILE_CONT ( percentile )
WITHIN GROUP (ORDER BY expr)
OVER ( [ PARTITION BY expr_list ] )
```

Arguments

- \textit{percentile}
  Numeric constant between 0 and 1. Nulls are ignored in the calculation.

- \textit{WITHIN GROUP (ORDER BY expr)}
  Specifies numeric or date/time values to sort and compute the percentile over.

- \textit{OVER}
  Specifies the window partitioning. The OVER clause cannot contain a window ordering or window
  frame specification.
PARTITION BY `expr`
Optional argument that sets the range of records for each group in the OVER clause.

**Returns**

The return type is determined by the data type of the ORDER BY expression in the WITHIN GROUP clause. The following table shows the return type for each ORDER BY expression data type.

<table>
<thead>
<tr>
<th>Input Type</th>
<th>Return Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>INT2, INT4, INT8, NUMERIC, DECIMAL</td>
<td>DECIMAL</td>
</tr>
<tr>
<td>FLOAT, DOUBLE</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>DATE</td>
<td>DATE</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>TIMESTAMP</td>
</tr>
</tbody>
</table>

**Usage Notes**

If the ORDER BY expression is a DECIMAL data type defined with the maximum precision of 38 digits, it is possible that PERCENTILE_CONT will return either an inaccurate result or an error. If the return value of the PERCENTILE_CONT function exceeds 38 digits, the result is truncated to fit, which causes a loss of precision. If, during interpolation, an intermediate result exceeds the maximum precision, a numeric overflow occurs and the function returns an error. To avoid these conditions, we recommend either using a data type with lower precision or casting the ORDER BY expression to a lower precision.

For example, a SUM function with a DECIMAL argument returns a default precision of 38 digits. The scale of the result is the same as the scale of the argument. So, for example, a SUM of a DECIMAL(5,2) column returns a DECIMAL(38,2) data type.

The following example uses a SUM function in the ORDER BY clause of a PERCENTILE_CONT function. The data type of the PRICEPAID column is DECIMAL (8,2), so the SUM function returns DECIMAL(38,2).

```sql
select salesid, sum(pricepaid), percentile_cont(0.6) within group (order by sum(pricepaid) desc) over() from sales where salesid < 10 group by salesid;
```

To avoid a potential loss of precision or an overflow error, cast the result to a DECIMAL data type with lower precision, as the following example shows.

```sql
select salesid, sum(pricepaid), percentile_cont(0.6) within group (order by sum(pricepaid)::decimal(30,2) desc) over() from sales where salesid < 10 group by salesid;
```

**Examples**

See PERCENTILE_CONT window function examples (p. 457).

**PERCENTILE_DISC window function**

PERCENTILE_DISC is an inverse distribution function that assumes a discrete distribution model. It takes a percentile value and a sort specification and returns an element from the given set.
For a given percentile value P, PERCENTILE_DISC sorts the values of the expression in the ORDER BY clause and returns the value with the smallest cumulative distribution value (with respect to the same sort specification) that is greater than or equal to P.

You can specify only the PARTITION clause in the OVER clause.

**Syntax**

\[
\text{PERCENTILE\_DISC} \ ( \text{percentile} ) \\
\text{WITHIN GROUP} \ ( \text{ORDER BY} \ \text{expr} ) \\
\text{OVER} \ ( \ [ \ \text{PARTITION BY} \ \text{expr\_list} \ ] \ )
\]

**Arguments**

- **percentile**: Numeric constant between 0 and 1. Nulls are ignored in the calculation.
- **WITHIN GROUP (ORDER BY expr)**: Specifies numeric or date/time values to sort and compute the percentile over.
- **OVER**: Specifies the window partitioning. The OVER clause cannot contain a window ordering or window frame specification.
- **PARTITION BY expr**: Optional argument that sets the range of records for each group in the OVER clause.

**Returns**

The same data type as the ORDER BY expression in the WITHIN GROUP clause.

**Examples**

See PERCENTILE_DISC window function examples (p. 458).

**RANK window function**

The RANK window function determines the rank of a value in a group of values, based on the ORDER BY expression in the OVER clause. If the optional PARTITION BY clause is present, the rankings are reset for each group of rows. Rows with equal values for the ranking criteria receive the same rank. Amazon Redshift adds the number of tied rows to the tied rank to calculate the next rank and thus the ranks may not be consecutive numbers. For example, if two rows are ranked 1, the next rank would be 3.

You can have ranking functions with different PARTITION BY and ORDER BY clauses in the same query.

**Synopsis**

\[
\text{RANK} \ () \ \text{OVER} \\
\{ \\
\ [ \ \text{PARTITION BY} \ \text{expr\_list} \ ] \\
\text{ORDER BY} \ \text{order\_list} \\
\}
\]
Arguments

() The RANK function takes no arguments, but the empty parentheses must be specified.
OVER
   Specifies the window clauses for the RANK function.
PARTITION BY expr_list
   Defines the window for the RANK function in terms of one or more expressions.
ORDER BY order_list
   Defines the column(s) on which the ranking values are based. If no PARTITION BY is specified, ORDER BY uses the entire table. ORDER BY is required for the RANK function.

Data types

The return type supported by the RANK function is INTEGER.

Examples

See RANK window function examples (p. 459).

ROW_NUMBER window function

Determines the ordinal number of the current row within a group of rows, counting from 1, based on the ORDER BY expression in the OVER clause. If the optional PARTITION BY clause is present, the ordinal numbers are reset for each group of rows. Rows with equal values for the ORDER BY expressions receive the different row numbers non-deterministically.

Synopsis

```
ROW_NUMBER () OVER
{
[ PARTITION BY expr_list ]
ORDER BY order_list
}
```

Arguments

() The ROW_NUMBER function takes no arguments, but the empty parentheses must be specified.
OVER
   Specifies the window clauses for the ROW_NUMBER function.
PARTITION BY expr_list
   Defines the window for the ROW_NUMBER function in terms of one or more expressions.
ORDER BY order_list
   Defines the columns on which the row numbers are based. If no PARTITION BY is specified, ORDER BY uses the entire table. ORDER BY is required for the ROW_NUMBER function.

   If the ORDER BY does not produce a unique ordering, the order of the rows is non-deterministic. That is, the return order of those rows might be inconsistent from one run to another. For more information, see Unique ordering of data for window functions (p. 464).
**Data types**

Returns INTEGER.

**Examples**

See ROW_NUMBER window function example (p. 461).

**STDDEV_SAMP and STDDEV_POP window functions**

The STDDEV_SAMP and STDDEV_POP window functions return the sample and population standard deviation of a set of numeric values (integer, decimal, or floating-point). See also STDDEV_SAMP and STDDEV_POP functions (p. 420).

STDDEV_SAMP and STDDEV are synonyms for the same function.

**Synopsis**

```sql
STDDEV_SAMP | STDDEV | STDDEV_POP
( [ ALL ] expression ) OVER
{ [ PARTITION BY expr_list ]
[ ORDER BY order_list
                          frame_clause ]
}
```

**Arguments**

- **expression**
  
  The target column or expression that the function operates on.

- **ALL**
  
  With the argument ALL, the function retains all duplicate values from the expression. ALL is the default. DISTINCT is not supported.

- **OVER**
  
  Specifies the window clauses for the aggregation functions. The OVER clause distinguishes window aggregation functions from normal set aggregation functions.

- **PARTITION BY expr_list**
  
  Defines the window for the function in terms of one or more expressions.

- **ORDER BY order_list**
  
  Sorts the rows within each partition. If no PARTITION BY is specified, ORDER BY uses the entire table.

- **frame_clause**
  
  If an ORDER BY clause is used for an aggregate function, an explicit frame clause is required. The frame clause refines the set of rows in a function's window, including or excluding sets of rows within the ordered result. The frame clause consists of the ROWS keyword and associated specifiers. See Window function syntax summary (p. 430).

**Data types**

The argument types supported by the STDDEV functions are SMALLINT, INTEGER, BIGINT, NUMERIC, DECIMAL, REAL, and DOUBLE PRECISION.

Regardless of the data type of the expression, the return type of a STDDEV function is a double precision number.
Examples

See STDDEV_POP and VAR_POP window function examples (p. 461).

SUM window function

The SUM window function returns the sum of the input column or expression values. The SUM function works with numeric values and ignores NULL values.

Synopsis

```
SUM ( [ ALL ] expression ) OVER
(
[ PARTITION BY expr_list ]
[ ORDER BY order_list
   frame_clause ]
)
```

Arguments

expression

The target column or expression that the function operates on.

ALL

With the argument ALL, the function retains all duplicate values from the expression. ALL is the default. DISTINCT is not supported.

OVER

Specifies the window clauses for the aggregation functions. The OVER clause distinguishes window aggregation functions from normal set aggregation functions.

PARTITION BY expr_list

Defines the window for the SUM function in terms of one or more expressions.

ORDER BY order_list

Sorts the rows within each partition. If no PARTITION BY is specified, ORDER BY uses the entire table.

frame_clause

If an ORDER BY clause is used for an aggregate function, an explicit frame clause is required. The frame clause refines the set of rows in a function's window, including or excluding sets of rows within the ordered result. The frame clause consists of the ROWS keyword and associated specifiers. See Window function syntax summary (p. 430).

Data types

The argument types supported by the SUM function are SMALLINT, INTEGER, BIGINT, NUMERIC, DECIMAL, REAL, and DOUBLE PRECISION.

The return types supported by the SUM function are:

- BIGINT for SMALLINT or INTEGER arguments
- NUMERIC for BIGINT arguments
- DOUBLE PRECISION for floating-point arguments

Examples

See SUM window function examples (p. 462).
VAR_SAMP and VAR_POP window functions

The VAR_SAMP and VAR_POP window functions return the sample and population variance of a set of numeric values (integer, decimal, or floating-point). See also VAR_SAMP and VAR_POP functions (p. 423).

VAR_SAMP and VARIANCE are synonyms for the same function.

Synopsis

```
VAR_SAMP | VARIANCE | VAR_POP
( [ ALL ] expression ) OVER
{ [ PARTITION BY expr_list ]
[ ORDER BY order_list ]
} frame_clause
```

Arguments

`expression`
- The target column or expression that the function operates on.

`ALL`
- With the argument ALL, the function retains all duplicate values from the expression. ALL is the default. DISTINCT is not supported.

`OVER`
- Specifies the window clauses for the aggregation functions. The OVER clause distinguishes window aggregation functions from normal set aggregation functions.

`PARTITION BY expr_list`
- Defines the window for the function in terms of one or more expressions.

`ORDER BY order_list`
- Sorts the rows within each partition. If no PARTITION BY is specified, ORDER BY uses the entire table.

`frame_clause`
- If an ORDER BY clause is used for an aggregate function, an explicit frame clause is required. The frame clause refines the set of rows in a function's window, including or excluding sets of rows within the ordered result. The frame clause consists of the ROWS keyword and associated specifiers. See Window function syntax summary (p. 430).

Data types

The argument types supported by the VARIANCE functions are SMALLINT, INTEGER, BIGINT, NUMERIC, DECIMAL, REAL, and DOUBLE PRECISION.

Regardless of the data type of the expression, the return type of a VARIANCE function is a double precision number.

Window function examples

Topics
- AVG window function examples (p. 449)
- COUNT window function examples (p. 449)
- DENSE_RANK window function examples (p. 450)
- FIRST_VALUE and LAST_VALUE window function examples (p. 451)
This section provides examples for using the window functions.

Some of the window function examples in this section use a table named WINSALES, which contains 11 rows:

<table>
<thead>
<tr>
<th>SALESID</th>
<th>DATEID</th>
<th>SELLERID</th>
<th>BUYERID</th>
<th>QTY</th>
<th>QTY_SHIPPED</th>
</tr>
</thead>
<tbody>
<tr>
<td>30001</td>
<td>8/2/2003</td>
<td>3</td>
<td>B</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>10001</td>
<td>12/24/2003</td>
<td>1</td>
<td>C</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>10005</td>
<td>12/24/2003</td>
<td>1</td>
<td>A</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>40001</td>
<td>1/9/2004</td>
<td>4</td>
<td>A</td>
<td>40</td>
<td></td>
</tr>
<tr>
<td>10006</td>
<td>1/18/2004</td>
<td>1</td>
<td>C</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>20001</td>
<td>2/12/2004</td>
<td>2</td>
<td>B</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>40005</td>
<td>2/12/2004</td>
<td>4</td>
<td>A</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>20002</td>
<td>2/16/2004</td>
<td>2</td>
<td>C</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>30003</td>
<td>4/18/2004</td>
<td>3</td>
<td>B</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>30004</td>
<td>4/18/2004</td>
<td>3</td>
<td>B</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>30007</td>
<td>9/7/2004</td>
<td>3</td>
<td>C</td>
<td>30</td>
<td></td>
</tr>
</tbody>
</table>

The following script creates and populates the sample WINSALES table.

```sql
create table winsales(
    salesid int,
    dateid date,
    sellerid int,
    buyerid char(10),
    qty int,
    qty_shipped int);

insert into winsales values
    (30001, '8/2/2003', 3, 'b', 10, 10),
```
Compute a rolling average of quantities sold by date; order the results by date ID and sales ID:

```sql
select salesid, dateid, sellerid, qty,
    avg(qty) over
        (order by dateid, salesid rows unbounded preceding) as avg
from winsales
order by 2,1;
```

```
salesid |   dateid   | sellerid | qty | avg
---------+------------+----------+-----+-----
 30001   | 2003-08-02 |        3 |  10 |  10
 10001   | 2003-12-24 |        1 |  10 |  10
 10005   | 2003-12-24 |        1 |  30 |  16
 40001   | 2004-01-09 |        4 |  40 |  22
 10006   | 2004-01-18 |        1 |  10 |  20
 20001   | 2004-02-12 |        2 |  20 |  20
 40005   | 2004-02-12 |        4 |  10 |  18
 20002   | 2004-02-16 |        2 |  20 |  18
 30003   | 2004-04-18 |        3 |  15 |  18
 30004   | 2004-04-18 |        3 |  20 |  18
 30007   | 2004-09-07 |        3 |  30 |  19
(11 rows)
```

For a description of the WINSALES table, see Window function examples (p. 447).

Show the sales ID, quantity, and count of all rows from the beginning of the data window:

```sql
select salesid, qty,
    count(*) over (order by salesid rows unbounded preceding) as count
from winsales
order by salesid;
```

```
salesid | qty | count
---------+-----+-----
 10001   |  10 |   1
 10005   |  30 |   2
 10006   |  10 |   3
 20001   |  20 |   4
 20002   |  20 |   5
 30001   |  10 |   6
 30003   |  15 |   7
```

**AVG window function examples**

**COUNT window function examples**
For a description of the WINSALES table, see Window function examples (p. 447).

Show the sales ID, quantity, and count of non-null rows from the beginning of the data window. (In the WINSALES table, the QTY_SHIPPED column contains some NULLs.)

```sql
select salesid, qty, qty_shipped,
count(qty_shipped)
over (order by salesid rows unbounded preceding) as count
from winsales
order by salesid;
```

```
salesid | qty | qty_shipped | count
---------+-----+-------------+-------
10001 | 10 | 10 | 1
10005 | 30 | | 1
10006 | 10 | | 1
20001 | 20 | 20 | 2
20002 | 20 | 20 | 3
30001 | 10 | 10 | 4
30003 | 15 | | 4
30004 | 20 | | 4
30007 | 30 | | 4
40001 | 40 | | 4
40005 | 10 | 10 | 5
```

(11 rows)

**DENSE_RANK window function examples**

**Dense ranking with ORDER BY**

Order the table by the quantity sold (in descending order), and assign both a dense rank and a regular rank to each row. The results are sorted after the window function results are applied.

```sql
select salesid, qty,
dense_rank() over(order by qty desc) as d_rnk,
rank() over(order by qty desc) as rnk
from winsales
order by 2,1;
```

```
salesid | qty | d_rnk | rnk
---------+-----+-------+-----
10001 | 10 | 5 | 8
10006 | 10 | 5 | 8
30001 | 10 | 5 | 8
40005 | 10 | 5 | 8
30003 | 15 | 4 | 7
20001 | 20 | 3 | 4
20002 | 20 | 3 | 4
30004 | 20 | 3 | 4
10005 | 30 | 2 | 2
```

(11 rows)
Note the difference in rankings assigned to the same set of rows when the DENSE_RANK and RANK functions are used side by side in the same query. For a description of the WINSALES table, see Window function examples (p. 447).

**Dense ranking with PARTITION BY and ORDER BY**

Partition the table by SELLERID and order each partition by the quantity (in descending order) and assign a dense rank to each row. The results are sorted after the window function results are applied.

```sql
select salesid, sellerid, qty,
  dense_rank() over(partition by sellerid order by qty desc) as d_rnk
from winsales
order by 2,3,1;
```

<table>
<thead>
<tr>
<th>salesid</th>
<th>sellerid</th>
<th>qty</th>
<th>d_rnk</th>
</tr>
</thead>
<tbody>
<tr>
<td>10001</td>
<td>1</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>10006</td>
<td>1</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>10005</td>
<td>1</td>
<td>30</td>
<td>1</td>
</tr>
<tr>
<td>20001</td>
<td>2</td>
<td>20</td>
<td>1</td>
</tr>
<tr>
<td>20002</td>
<td>2</td>
<td>20</td>
<td>1</td>
</tr>
<tr>
<td>30001</td>
<td>3</td>
<td>10</td>
<td>4</td>
</tr>
<tr>
<td>30003</td>
<td>3</td>
<td>15</td>
<td>3</td>
</tr>
<tr>
<td>30004</td>
<td>3</td>
<td>20</td>
<td>2</td>
</tr>
<tr>
<td>30007</td>
<td>3</td>
<td>30</td>
<td>1</td>
</tr>
<tr>
<td>40005</td>
<td>4</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>40001</td>
<td>4</td>
<td>40</td>
<td>1</td>
</tr>
</tbody>
</table>

For a description of the WINSALES table, see Window function examples (p. 447).

**FIRST_VALUE and LAST_VALUE window function examples**

The following example returns the seating capacity for each venue in the VENUE table, with the results ordered by capacity (high to low). The FIRST_VALUE function is used to select the name of the venue that corresponds to the first row in the frame: in this case, the row with the highest number of seats. The results are partitioned by state, so when the VENUESTATE value changes, a new first value is selected. The window frame is unbounded so the same first value is selected for each row in each partition.

For California, Qualcomm Stadium has the highest number of seats (70561), so this name is the first value for all of the rows in the CA partition.

```sql
select venuestate, venueseats, venuename,
  first_value(venuename)
over(partition by venuestate
order by venueseats desc
rows between unbounded preceding and unbounded following)
from (select * from venue where venueseats >0)
order by venuestate;
```

<table>
<thead>
<tr>
<th>venuestate</th>
<th>venueseats</th>
<th>venuename</th>
<th>first_value</th>
</tr>
</thead>
</table>
| API Version 2012-12-01
451 |
<table>
<thead>
<tr>
<th>venuestate</th>
<th>venueseats</th>
<th>venuename</th>
<th>last_value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CA</td>
<td>70561</td>
<td>Qualcomm Stadium</td>
<td>Shoreline Amphitheatre</td>
</tr>
<tr>
<td>CA</td>
<td>69843</td>
<td>Monster Park</td>
<td>Shoreline Amphitheatre</td>
</tr>
<tr>
<td>CA</td>
<td>63026</td>
<td>McAfee Coliseum</td>
<td>Shoreline Amphitheatre</td>
</tr>
<tr>
<td>CA</td>
<td>56000</td>
<td>Dodger Stadium</td>
<td>Shoreline Amphitheatre</td>
</tr>
<tr>
<td>CA</td>
<td>45050</td>
<td>Angel Stadium of Anaheim</td>
<td>Shoreline Amphitheatre</td>
</tr>
<tr>
<td>CA</td>
<td>42445</td>
<td>PETCO Park</td>
<td>Shoreline Amphitheatre</td>
</tr>
<tr>
<td>CA</td>
<td>41503</td>
<td>AT&amp;T Park</td>
<td>Shoreline Amphitheatre</td>
</tr>
<tr>
<td>CA</td>
<td>22000</td>
<td>Shoreline Amphitheatre</td>
<td>Shoreline Amphitheatre</td>
</tr>
<tr>
<td>CO</td>
<td>76125</td>
<td>INVESCO Field</td>
<td>Coors Field</td>
</tr>
<tr>
<td>CO</td>
<td>50445</td>
<td>Coors Field</td>
<td>Coors Field</td>
</tr>
<tr>
<td>DC</td>
<td>41888</td>
<td>Nationals Park</td>
<td>Nationals Park</td>
</tr>
<tr>
<td>FL</td>
<td>74916</td>
<td>Dolphin Stadium</td>
<td>Dolphin Stadium</td>
</tr>
<tr>
<td>FL</td>
<td>73800</td>
<td>Jacksonville Municipal Stadium</td>
<td>Dolphin Stadium</td>
</tr>
<tr>
<td>FL</td>
<td>65647</td>
<td>Raymond James Stadium</td>
<td>Dolphin Stadium</td>
</tr>
<tr>
<td>FL</td>
<td>36048</td>
<td>Tropicana Field</td>
<td>Tropicana Field</td>
</tr>
</tbody>
</table>

The next example uses the LAST_VALUE function instead of FIRST_VALUE; otherwise, the query is the same as the previous example. For California, Shoreline Amphitheatre is returned for every row in the partition because it has the lowest number of seats (22000).

```sql
select venuestate, venueseats, venuename,
last_value(venuename)
over(partition by venuestate
order by venueseats desc
rows between unbounded preceding and unbounded following)
from (select * from venue where venueseats >0)
order by venuestate;
```

<table>
<thead>
<tr>
<th>venuestate</th>
<th>venueseats</th>
<th>venuename</th>
<th>last_value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CA</td>
<td>70561</td>
<td>Qualcomm Stadium</td>
<td>Shoreline Amphitheatre</td>
</tr>
<tr>
<td>CA</td>
<td>69843</td>
<td>Monster Park</td>
<td>Shoreline Amphitheatre</td>
</tr>
<tr>
<td>CA</td>
<td>63026</td>
<td>McAfee Coliseum</td>
<td>Shoreline Amphitheatre</td>
</tr>
<tr>
<td>CA</td>
<td>56000</td>
<td>Dodger Stadium</td>
<td>Shoreline Amphitheatre</td>
</tr>
<tr>
<td>CA</td>
<td>45050</td>
<td>Angel Stadium of Anaheim</td>
<td>Shoreline Amphitheatre</td>
</tr>
<tr>
<td>CA</td>
<td>42445</td>
<td>PETCO Park</td>
<td>Shoreline Amphitheatre</td>
</tr>
<tr>
<td>CA</td>
<td>41503</td>
<td>AT&amp;T Park</td>
<td>Shoreline Amphitheatre</td>
</tr>
<tr>
<td>CA</td>
<td>22000</td>
<td>Shoreline Amphitheatre</td>
<td>Shoreline Amphitheatre</td>
</tr>
<tr>
<td>CO</td>
<td>76125</td>
<td>INVESCO Field</td>
<td>Coors Field</td>
</tr>
<tr>
<td>CO</td>
<td>50445</td>
<td>Coors Field</td>
<td>Coors Field</td>
</tr>
<tr>
<td>DC</td>
<td>41888</td>
<td>Nationals Park</td>
<td>Nationals Park</td>
</tr>
<tr>
<td>FL</td>
<td>74916</td>
<td>Dolphin Stadium</td>
<td>Tropicana Field</td>
</tr>
<tr>
<td>FL</td>
<td>73800</td>
<td>Jacksonville Municipal Stadium</td>
<td>Tropicana Field</td>
</tr>
<tr>
<td>FL</td>
<td>65647</td>
<td>Raymond James Stadium</td>
<td>Tropicana Field</td>
</tr>
<tr>
<td>FL</td>
<td>36048</td>
<td>Tropicana Field</td>
<td>Tropicana Field</td>
</tr>
</tbody>
</table>

The following example shows the use of the IGNORE NULLS option and relies on the addition of a new row to the VENUE table:

```sql
insert into venue values(2000,null,'Stanford','CA',90000);
```
This new row contains a NULL value for the VENUENAME column. Now repeat the FIRST_VALUE query that was shown earlier in this section:

```sql
select venuestate, venueseats, venuename,
    first_value(venuename)
over(partition by venuestate
    order by venueseats desc
    rows between unbounded preceding and unbounded following)
from (select * from venue where venueseats > 0)
order by venuestate;
```

```plaintext
<table>
<thead>
<tr>
<th>venuestate</th>
<th>venueseats</th>
<th>venuename</th>
<th>first_value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CA</td>
<td>90000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CA</td>
<td>70561</td>
<td>Qualcomm Stadium</td>
<td></td>
</tr>
<tr>
<td>CA</td>
<td>69843</td>
<td>Monster Park</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Because the new row contains the highest VENUESEATS value (90000) and its VENUENAME is NULL, the FIRST_VALUE function returns NULL for the CA partition. To ignore rows like this in the function evaluation, add the IGNORE NULLS option to the function argument:

```sql
select venuestate, venueseats, venuename,
    first_value(venuename ignore nulls)
over(partition by venuestate
    order by venueseats desc
    rows between unbounded preceding and unbounded following)
from (select * from venue where venuestate = 'CA')
order by venuestate;
```

```plaintext
<table>
<thead>
<tr>
<th>venuestate</th>
<th>venueseats</th>
<th>venuename</th>
<th>first_value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CA</td>
<td>90000</td>
<td>Qualcomm Stadium</td>
<td>Qualcomm Stadium</td>
</tr>
<tr>
<td>CA</td>
<td>70561</td>
<td>Qualcomm Stadium</td>
<td>Qualcomm Stadium</td>
</tr>
<tr>
<td>CA</td>
<td>69843</td>
<td>Monster Park</td>
<td>Qualcomm Stadium</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**LAG window function examples**

The following example shows the quantity of tickets sold to the buyer with a buyer ID of 3 and the time that buyer 3 bought the tickets. To compare each sale with the previous sale for buyer 3, the query returns the previous quantity sold for each sale. Since there is no purchase before 1/16/2008, the first previous quantity sold value is null:

```sql
select buyerid, saletime, qtysold,
    lag(qtysold, 1) over (order by buyerid, saletime) as prev_qtysold
from sales where buyerid = 3 order by buyerid, saletime;
```

```plaintext
<table>
<thead>
<tr>
<th>buyerid</th>
<th>saletime</th>
<th>qtysold</th>
<th>prev_qtysold</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2008-01-16 01:06:09</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2008-01-28 02:10:01</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2008-03-12 10:39:53</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2008-03-13 02:56:07</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2008-03-29 08:21:39</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2008-04-27 02:39:01</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>
```
LEAD window function examples

The following example provides the commission for events in the SALES table for which tickets were sold on January 1, 2008 and January 2, 2008 and the commission paid for ticket sales for the subsequent sale.

```sql
select eventid, commission, saletime,
    lead(commission, 1) over (order by saletime) as next_comm
from sales where saletime between '2008-01-01 00:00:00' and '2008-01-02 12:59:59'
order by saletime;
```

```plaintext
eventid | commission |      saletime       | next_comm
---------+------------+---------------------+-----------
6213    |      52.05 | 2008-01-01 01:00:19 |    106.20
7003    |     106.20 | 2008-01-01 02:30:52 |    103.20
8762    |     103.20 | 2008-01-01 03:50:02 |     70.80
1150    |     103.20 | 2008-01-01 06:06:57 |     50.55
1749    |     103.20 | 2008-01-01 07:05:02 |    125.40
8649    |    125.40  | 2008-01-01 07:26:20 |     35.10
2903    |      35.10 | 2008-01-01 09:41:06 |    259.50
6605    |     259.50 | 2008-01-01 12:50:55 |    628.80
6870    |     628.80 | 2008-01-01 12:59:34 |     74.10
6977    |     74.10  | 2008-01-01 13:11:16 |     13.50
4650    |     13.50  | 2008-01-02 01:40:59 |     26.55
4515    |     26.55  | 2008-01-02 01:52:35 |     22.80
5465    |     22.80  | 2008-01-02 02:28:01 |     45.60
5465    |     45.60  | 2008-01-02 02:28:02 |     53.10
7003    |     53.10  | 2008-01-02 02:31:12 |     70.35
4124    |     70.35  | 2008-01-02 03:12:15 |     36.15
1673    |     36.15  | 2008-01-02 03:15:00 |    1300.80
...
(39 rows)
```

MAX window function examples

Show the sales ID, quantity, and maximum quantity from the beginning of the data window:

```sql
select salesid, qty,
    max(qty) over (order by salesid rows unbounded preceding) as max
from winsales
order by salesid;
```

```plaintext
salesid | qty | max
---------+-----+-----
10001   | 10  | 10
10005   | 30  | 30
10006   | 10  | 30
```

(39 rows)
For a description of the WINSALES table, see Window function examples (p. 447).

Show the salesid, quantity, and maximum quantity in a restricted frame:

```sql
select salesid, qty, 
max(qty) over (order by salesid rows between 2 preceding and 1 preceding) as max
from winsales
order by salesid;
```

<table>
<thead>
<tr>
<th>salesid</th>
<th>qty</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>10001</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>10005</td>
<td>30</td>
<td>10</td>
</tr>
<tr>
<td>10006</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>20001</td>
<td>20</td>
<td>30</td>
</tr>
<tr>
<td>20002</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>30001</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>30003</td>
<td>15</td>
<td>20</td>
</tr>
<tr>
<td>30004</td>
<td>20</td>
<td>15</td>
</tr>
<tr>
<td>30007</td>
<td>30</td>
<td>20</td>
</tr>
<tr>
<td>40001</td>
<td>40</td>
<td>30</td>
</tr>
<tr>
<td>40005</td>
<td>10</td>
<td>40</td>
</tr>
</tbody>
</table>

(11 rows)

**MIN window function examples**

Show the sales ID, quantity, and minimum quantity from the beginning of the data window:

```sql
select salesid, qty, 
min(qty) over (order by salesid rows unbounded preceding)
from winsales
order by salesid;
```

<table>
<thead>
<tr>
<th>salesid</th>
<th>qty</th>
<th>min</th>
</tr>
</thead>
<tbody>
<tr>
<td>10001</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>10005</td>
<td>30</td>
<td>10</td>
</tr>
<tr>
<td>10006</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>20001</td>
<td>20</td>
<td>10</td>
</tr>
<tr>
<td>20002</td>
<td>20</td>
<td>10</td>
</tr>
<tr>
<td>30001</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>30003</td>
<td>15</td>
<td>10</td>
</tr>
<tr>
<td>30004</td>
<td>20</td>
<td>10</td>
</tr>
<tr>
<td>30007</td>
<td>30</td>
<td>10</td>
</tr>
</tbody>
</table>

(11 rows)
For a description of the WINSALES table, see Window function examples (p. 447).

Show the sales ID, quantity, and minimum quantity in a restricted frame:

```
select salesid, qty,
min(qty) over
(order by salesid rows between 2 preceding and 1 preceding) as min
from winsales
order by salesid;
```

```
salesid | qty | min
---------+-----+-----
10001   | 10  |
10005   | 30  | 10  
10006   | 10  | 10  
20001   | 20  | 10  
20002   | 20  | 10  
30001   | 10  | 20  
30003   | 15  | 10  
30004   | 20  | 10  
30007   | 30  | 15  
40001   | 40  | 20  
40005   | 10  | 30  
(11 rows)
```

**NTH_VALUE window function examples**

The following example shows the number of seats in the third largest venue in California, Florida, and New York compared to the number of seats in the other venues in those states:

```
select venuestate, venuename, venueseats,
nth_value(venueseats, 3) ignore nulls
over(partition by venuestate order by venueseats desc
rows between unbounded preceding and unbounded following)
as third_most_seats
from (select * from venue where venueseats > 0 and
venuestate in('CA', 'FL', 'NY'))
order by venuestate;
```

```
venuestate | venuename                | venueseats | third_most_seats
------------+--------------------------+------------+------------------
CA          | Qualcomm Stadium         | 70561      | 63026
CA          | Monster Park             | 69843      | 63026
CA          | McAfee Coliseum          | 63026      | 63026
CA          | Dodger Stadium           | 56000      | 63026
CA          | Angel Stadium of Anaheim | 45050      | 63026
CA          | PETCO Park               | 42445      | 63026
CA          | AT&T Park                | 41503      | 63026
CA          | Shoreline Amphitheatre   | 22000      | 63026
FL          | Dolphin Stadium          | 74916      | 65647
FL          | Jacksonville Municipal Stadium | 73800   | 65647
```

Amazon Redshift Database Developer Guide
Window functions

API Version 2012-12-01
456
NTILE window function examples

The following example ranks into four ranking groups the price paid for Hamlet tickets on August 26, 2008. The result set is 17 rows, divided almost evenly among the rankings 1 through 4:

```sql
select eventname, caldate, pricepaid, ntile(4)
over(order by pricepaid desc) from sales, event, date
where sales.eventid=event.eventid and event.dateid=date.dateid and eventname='Hamlet'
and caldate='2008-08-26'
order by 4;
```

<table>
<thead>
<tr>
<th>eventname</th>
<th>caldate</th>
<th>pricepaid</th>
<th>ntile</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hamlet</td>
<td>2008-08-26</td>
<td>1883.00</td>
<td>1</td>
</tr>
<tr>
<td>Hamlet</td>
<td>2008-08-26</td>
<td>1065.00</td>
<td>1</td>
</tr>
<tr>
<td>Hamlet</td>
<td>2008-08-26</td>
<td>589.00</td>
<td>1</td>
</tr>
<tr>
<td>Hamlet</td>
<td>2008-08-26</td>
<td>530.00</td>
<td>1</td>
</tr>
<tr>
<td>Hamlet</td>
<td>2008-08-26</td>
<td>472.00</td>
<td>1</td>
</tr>
<tr>
<td>Hamlet</td>
<td>2008-08-26</td>
<td>460.00</td>
<td>2</td>
</tr>
<tr>
<td>Hamlet</td>
<td>2008-08-26</td>
<td>355.00</td>
<td>2</td>
</tr>
<tr>
<td>Hamlet</td>
<td>2008-08-26</td>
<td>334.00</td>
<td>2</td>
</tr>
<tr>
<td>Hamlet</td>
<td>2008-08-26</td>
<td>296.00</td>
<td>2</td>
</tr>
<tr>
<td>Hamlet</td>
<td>2008-08-26</td>
<td>230.00</td>
<td>3</td>
</tr>
<tr>
<td>Hamlet</td>
<td>2008-08-26</td>
<td>216.00</td>
<td>3</td>
</tr>
<tr>
<td>Hamlet</td>
<td>2008-08-26</td>
<td>212.00</td>
<td>3</td>
</tr>
<tr>
<td>Hamlet</td>
<td>2008-08-26</td>
<td>106.00</td>
<td>3</td>
</tr>
<tr>
<td>Hamlet</td>
<td>2008-08-26</td>
<td>100.00</td>
<td>4</td>
</tr>
<tr>
<td>Hamlet</td>
<td>2008-08-26</td>
<td>94.00</td>
<td>4</td>
</tr>
<tr>
<td>Hamlet</td>
<td>2008-08-26</td>
<td>53.00</td>
<td>4</td>
</tr>
<tr>
<td>Hamlet</td>
<td>2008-08-26</td>
<td>25.00</td>
<td>4</td>
</tr>
</tbody>
</table>

PERCENTILE_CONT window function examples

The following examples uses the WINSALES table. For a description of the WINSALES table, see Window function examples (p. 447).

```sql
select sellerid, qty, percentile_cont(0.5)
within group (order by qty)
over() as median from winsales;
```

<table>
<thead>
<tr>
<th>sellerid</th>
<th>qty</th>
<th>median</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>20.0</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>20.0</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>20.0</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>20.0</td>
</tr>
</tbody>
</table>
The following example calculates the PERCENTILE_CONT and PERCENTILE_DISC of the ticket sales for sellers in Washington state.

```
SELECT sellerid, state, sum(qtysold*pricepaid) sales,
percentile_cont(0.6) within group (order by sum(qtysold*pricepaid::decimal(14,2) ) desc) over(),
percentile_disc(0.6) within group (order by sum(qtysold*pricepaid::decimal(14,2) ) desc) over()
from sales s, users u
where s.sellerid = u.userid and state = 'WA' and sellerid < 1000
group by sellerid, state;
```

```
sellerid | state | sales  | percentile_cont | percentile_disc
-----------+-------+---------+-----------------+-----------------
  127 | WA | 6076.00 | 2044.20 | 1531.00
  787 | WA | 6035.00 | 2044.20 | 1531.00
  381 | WA | 5881.00 | 2044.20 | 1531.00
  777 | WA | 2814.00 | 2044.20 | 1531.00
   33 | WA | 1531.00 | 2044.20 | 1531.00
  800 | WA | 1476.00 | 2044.20 | 1531.00
   1 | WA | 1177.00 | 2044.20 | 1531.00
(7 rows)
```

**PERCENTILE_DISC window function examples**

The following examples uses the WINSALES table. For a description of the WINSALES table, see Window function examples (p. 447).
select sellerid, qty, percentile_disc(0.5) within group (order by qty) over() as median from winsales;

<table>
<thead>
<tr>
<th>sellerid</th>
<th>qty</th>
<th>median</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>3</td>
<td>15</td>
<td>20</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>3</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>1</td>
<td>30</td>
<td>20</td>
</tr>
<tr>
<td>3</td>
<td>30</td>
<td>20</td>
</tr>
<tr>
<td>4</td>
<td>40</td>
<td>20</td>
</tr>
</tbody>
</table>

(11 rows)

select sellerid, qty, percentile_disc(0.5) within group (order by qty) over(partition by sellerid) as median from winsales;

<table>
<thead>
<tr>
<th>sellerid</th>
<th>qty</th>
<th>median</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>40</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>30</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>15</td>
</tr>
<tr>
<td>3</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>3</td>
<td>20</td>
<td>15</td>
</tr>
<tr>
<td>3</td>
<td>30</td>
<td>15</td>
</tr>
</tbody>
</table>

(11 rows)

**RANK window function examples**

**Ranking with ORDER BY**

Order the table by the quantity sold (default ascending), and assign a rank to each row. The results are sorted after the window function results are applied:

select salesid, qty, rank() over (order by qty) as rnk from winsales order by 2,1;

<table>
<thead>
<tr>
<th>salesid</th>
<th>qty</th>
<th>rnk</th>
</tr>
</thead>
<tbody>
<tr>
<td>10001</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>10006</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>30001</td>
<td>10</td>
<td>1</td>
</tr>
</tbody>
</table>
Note that the outer ORDER BY clause in this example includes columns 2 and 1 to make sure that Amazon Redshift returns consistently sorted results each time this query is run. For example, rows with sales IDs 10001 and 10006 have identical QTY and RNK values. Ordering the final result set by column 1 ensures that row 10001 always falls before 10006. For a description of the WINSALES table, see Window function examples (p. 447).

**Ranking with PARTITION BY and ORDER BY**

In this example, the ordering is reversed for the window function (order by qty desc). Now the highest rank value applies to the highest QTY value.

```sql
select salesid, qty,
rank() over (order by qty desc) as rank
from winsales
order by 2,1;
```

```plaintext
salesid | qty | rank
---------+-----+-----
10001   | 10  |  8  
10006   | 10  |  8  
30001   | 10  |  8  
40005   | 10  |  8  
30003   | 15  |  7  
20001   | 20  |  4  
20002   | 20  |  4  
30004   | 20  |  4  
10005   | 30  |  2  
30007   | 30  |  2  
40001   | 40  |  1  
(11 rows)
```

For a description of the WINSALES table, see Window function examples (p. 447).

Partition the table by SELLERID and order each partition by the quantity (in descending order) and assign a rank to each row. The results are sorted after the window function results are applied.

```sql
select salesid, sellerid, qty, rank() over (
partition by sellerid
order by qty desc) as rank
from winsales
order by 2,3,1;
```

```plaintext
salesid | sellerid | qty | rank
---------+----------+-----+-----
10001   |         1| 10  |  2  
10006   |         1| 10  |  2  
(2 rows)
```

For a description of the WINSALES table, see Window function examples (p. 447).
## ROW_NUMBER window function example

The following example partitions the table by SELLERID and orders each partition by QTY (in ascending order), then assigns a row number to each row. The results are sorted after the window function results are applied.

```sql
select salesid, sellerid, qty,
row_number() over
(partition by sellerid
order by qty asc) as row
from winsales
order by 2,4;
```

<table>
<thead>
<tr>
<th>salesid</th>
<th>sellerid</th>
<th>qty</th>
<th>row</th>
</tr>
</thead>
<tbody>
<tr>
<td>10006</td>
<td>1</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>10001</td>
<td>1</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>10005</td>
<td>1</td>
<td>30</td>
<td>3</td>
</tr>
<tr>
<td>20001</td>
<td>2</td>
<td>20</td>
<td>1</td>
</tr>
<tr>
<td>20002</td>
<td>2</td>
<td>20</td>
<td>2</td>
</tr>
<tr>
<td>30001</td>
<td>3</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>30003</td>
<td>3</td>
<td>15</td>
<td>2</td>
</tr>
<tr>
<td>30004</td>
<td>3</td>
<td>20</td>
<td>3</td>
</tr>
<tr>
<td>30007</td>
<td>3</td>
<td>30</td>
<td>4</td>
</tr>
<tr>
<td>40005</td>
<td>4</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>40001</td>
<td>4</td>
<td>40</td>
<td>2</td>
</tr>
</tbody>
</table>

(11 rows)

For a description of the WINSALES table, see Window function examples (p. 447).

## STDDEV_POP and VAR_POP window function examples

The following example shows how to use STDDEV_POP and VAR_POP functions as window functions. The query computes the population variance and population standard deviation for PRICEPAID values in the SALES table.

```sql
select salesid, dateid, pricepaid,
round(stddev_pop(pricepaid) over
(order by dateid, salesid rows unbounded preceding)) as stddevpop,
round(var_pop(pricepaid) over
(order by dateid, salesid rows unbounded preceding)) as varpop
from sales
order by 2,1;
```
The sample standard deviation and variance functions can be used in the same way.

**SUM window function examples**

**Cumulative sums (running totals)**

Create a cumulative (rolling) sum of sales quantities ordered by date and sales ID:

```
select salesid, dateid, sellerid, qty,
sum(qty) over (order by dateid, salesid rows unbounded preceding) as sum
from winsales
order by 2,1;
```

<table>
<thead>
<tr>
<th>salesid</th>
<th>dateid</th>
<th>sellerid</th>
<th>qty</th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>30001</td>
<td>2003-08-02</td>
<td>3</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>10001</td>
<td>2003-12-24</td>
<td>1</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>10005</td>
<td>2003-12-24</td>
<td>1</td>
<td>30</td>
<td>50</td>
</tr>
<tr>
<td>40001</td>
<td>2004-01-09</td>
<td>4</td>
<td>40</td>
<td>90</td>
</tr>
<tr>
<td>10006</td>
<td>2004-01-18</td>
<td>1</td>
<td>10</td>
<td>100</td>
</tr>
<tr>
<td>20001</td>
<td>2004-02-12</td>
<td>2</td>
<td>20</td>
<td>120</td>
</tr>
<tr>
<td>40005</td>
<td>2004-02-12</td>
<td>4</td>
<td>10</td>
<td>130</td>
</tr>
<tr>
<td>20002</td>
<td>2004-02-16</td>
<td>2</td>
<td>20</td>
<td>150</td>
</tr>
<tr>
<td>30003</td>
<td>2004-04-18</td>
<td>3</td>
<td>15</td>
<td>165</td>
</tr>
<tr>
<td>30004</td>
<td>2004-04-18</td>
<td>3</td>
<td>20</td>
<td>185</td>
</tr>
<tr>
<td>30007</td>
<td>2004-09-07</td>
<td>3</td>
<td>30</td>
<td>215</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

For a description of the WINSALES table, see Window function examples (p. 447).

Create a cumulative (rolling) sum of sales quantities by date, partition the results by seller ID, and order the results by date and sales ID within the partition:

```
select salesid, dateid, sellerid, qty,
sum(qty) over (partition by sellerid
order by dateid, salesid rows unbounded preceding) as sum
from winsales
order by 2,1;
```

<table>
<thead>
<tr>
<th>salesid</th>
<th>dateid</th>
<th>sellerid</th>
<th>qty</th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>30001</td>
<td>2003-08-02</td>
<td>3</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>10001</td>
<td>2003-12-24</td>
<td>1</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(11 rows)
Number rows sequentially

Number all of the rows in the result set, ordered by the SELLERID and SALESID columns:

```sql
select salesid, sellerid, qty,
sum(1) over (order by sellerid, salesid rows unbounded preceding) as rownum
from winsales
order by 2,1;
```

<table>
<thead>
<tr>
<th>salesid</th>
<th>sellerid</th>
<th>qty</th>
<th>rownum</th>
</tr>
</thead>
<tbody>
<tr>
<td>10001</td>
<td>1</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>10005</td>
<td>1</td>
<td>30</td>
<td>2</td>
</tr>
<tr>
<td>10006</td>
<td>1</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>20001</td>
<td>2</td>
<td>20</td>
<td>4</td>
</tr>
<tr>
<td>20002</td>
<td>2</td>
<td>20</td>
<td>5</td>
</tr>
<tr>
<td>30001</td>
<td>3</td>
<td>10</td>
<td>6</td>
</tr>
<tr>
<td>30003</td>
<td>3</td>
<td>15</td>
<td>7</td>
</tr>
<tr>
<td>30004</td>
<td>3</td>
<td>20</td>
<td>8</td>
</tr>
<tr>
<td>30007</td>
<td>3</td>
<td>30</td>
<td>9</td>
</tr>
<tr>
<td>40001</td>
<td>4</td>
<td>40</td>
<td>10</td>
</tr>
<tr>
<td>40005</td>
<td>4</td>
<td>10</td>
<td>11</td>
</tr>
</tbody>
</table>

(11 rows)

For a description of the WINSALES table, see Window function examples (p. 447).

Number all rows in the result set, partition the results by SELLERID, and order the results by SELLERID and SALESID within the partition:

```sql
select salesid, sellerid, qty,
sum(1) over (partition by sellerid
order by sellerid, salesid rows unbounded preceding) as rownum
from winsales
order by 2,1;
```

<table>
<thead>
<tr>
<th>salesid</th>
<th>sellerid</th>
<th>qty</th>
<th>rownum</th>
</tr>
</thead>
<tbody>
<tr>
<td>10001</td>
<td>1</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>10005</td>
<td>1</td>
<td>30</td>
<td>2</td>
</tr>
<tr>
<td>10006</td>
<td>1</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>20001</td>
<td>2</td>
<td>20</td>
<td>1</td>
</tr>
<tr>
<td>20002</td>
<td>2</td>
<td>20</td>
<td>2</td>
</tr>
<tr>
<td>30001</td>
<td>3</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>30003</td>
<td>3</td>
<td>15</td>
<td>2</td>
</tr>
<tr>
<td>30004</td>
<td>3</td>
<td>20</td>
<td>3</td>
</tr>
</tbody>
</table>

(11 rows)
Unique ordering of data for window functions

If an ORDER BY clause for a window function does not produce a unique and total ordering of the data, the order of the rows is non-deterministic. If the ORDER BY expression produces duplicate values (a partial ordering), the return order of those rows may vary in multiple runs and window functions may return unexpected or inconsistent results.

For example, the following query returns different results over multiple runs because order by dateid does not produce a unique ordering of the data for the SUM window function.

```sql
select dateid, pricepaid,
sum(pricepaid) over(order by dateid rows unbounded preceding) as sumpaid
from sales
group by dateid, pricepaid;
```

<table>
<thead>
<tr>
<th>dateid</th>
<th>pricepaid</th>
<th>sumpaid</th>
</tr>
</thead>
<tbody>
<tr>
<td>1827</td>
<td>1730.00</td>
<td>1730.00</td>
</tr>
<tr>
<td>1827</td>
<td>708.00</td>
<td>2438.00</td>
</tr>
<tr>
<td>1827</td>
<td>234.00</td>
<td>2672.00</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

```sql
select dateid, pricepaid,
sum(pricepaid) over(order by dateid rows unbounded preceding) as sumpaid
from sales
group by dateid, pricepaid;
```

<table>
<thead>
<tr>
<th>dateid</th>
<th>pricepaid</th>
<th>sumpaid</th>
</tr>
</thead>
<tbody>
<tr>
<td>1827</td>
<td>234.00</td>
<td>234.00</td>
</tr>
<tr>
<td>1827</td>
<td>472.00</td>
<td>706.00</td>
</tr>
<tr>
<td>1827</td>
<td>347.00</td>
<td>1053.00</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

In this case, adding a second ORDER BY column to the window function may solve the problem:

```sql
select dateid, pricepaid,
sum(pricepaid) over(order by dateid, pricepaid rows unbounded preceding) as sumpaid
from sales
group by dateid, pricepaid;
```

<table>
<thead>
<tr>
<th>dateid</th>
<th>pricepaid</th>
<th>sumpaid</th>
</tr>
</thead>
<tbody>
<tr>
<td>1827</td>
<td>234.00</td>
<td>234.00</td>
</tr>
<tr>
<td>1827</td>
<td>337.00</td>
<td>571.00</td>
</tr>
<tr>
<td>1827</td>
<td>347.00</td>
<td>918.00</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Conditional expressions

Topics
- CASE expression (p. 465)
- COALESCE (p. 466)
- DECODE expression (p. 467)
- GREATEST and LEAST (p. 468)
- NVL expression (p. 469)
- NULLIF expression (p. 470)

Amazon Redshift supports some conditional expressions that are extensions to the SQL standard.

CASE expression

Syntax

The CASE expression is a conditional expression, similar to if/then/else statements found in other languages. CASE is used to specify a result when there are multiple conditions.

There are two types of CASE expressions: simple and searched.

In simple CASE expressions, an expression is compared with a value. When a match is found, the specified action in the THEN clause is applied. If no match is found, the action in the ELSE clause is applied.

In searched CASE expressions, each CASE is evaluated based on a Boolean expression, and the CASE statement returns the first matching CASE. If no matching CASEs are found among the WHEN clauses, the action in the ELSE clause is returned.

Simple CASE statement used to match conditions:

```
CASE expression
WHEN value THEN result
[WHEN...]
[ELSE result]
END
```

Searched CASE statement used to evaluate each condition:

```
CASE
WHEN boolean condition THEN result
[WHEN ...]
[ELSE result]
END
```

Arguments

- **expression**
  A column name or any valid expression.
- **value**
  Value that the expression is compared with, such as a numeric constant or a character string.
- **result**
  The target value or expression that is returned when an expression or Boolean condition is evaluated.
### Boolean condition

A Boolean condition is valid or true when the value is equal to the constant. When true, the result specified following the THEN clause is returned. If a condition is false, the result following the ELSE clause is returned. If the ELSE clause is omitted and no condition matches, the result is null.

### Examples

Use a simple CASE expression to replace New York City with Big Apple in a query against the VENUE table. Replace all other city names with other.

```sql
select venuecity,
case venuecity
when 'New York City'
then 'Big Apple'
else 'other'
end from venue
order by venueid desc;
```

<table>
<thead>
<tr>
<th>venuecity</th>
<th>case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Los Angeles</td>
<td>other</td>
</tr>
<tr>
<td>New York City</td>
<td>Big Apple</td>
</tr>
<tr>
<td>San Francisco</td>
<td>other</td>
</tr>
<tr>
<td>Baltimore</td>
<td>other</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>(202 rows)</td>
<td></td>
</tr>
</tbody>
</table>

Use a searched CASE expression to assign group numbers based on the PRICEPAID value for individual ticket sales:

```sql
select pricepaid,
case when pricepaid <10000 then 'group 1'
when pricepaid >10000 then 'group 2'
else 'group 3'
end from sales
order by 1 desc;
```

<table>
<thead>
<tr>
<th>pricepaid</th>
<th>case</th>
</tr>
</thead>
<tbody>
<tr>
<td>12624.00</td>
<td>group 2</td>
</tr>
<tr>
<td>10000.00</td>
<td>group 3</td>
</tr>
<tr>
<td>10000.00</td>
<td>group 3</td>
</tr>
<tr>
<td>9996.00</td>
<td>group 1</td>
</tr>
<tr>
<td>9988.00</td>
<td>group 1</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>(172456 rows)</td>
<td></td>
</tr>
</tbody>
</table>

### COALESCE

Synonym of the NVL expression.

See NVL expression (p. 469).
DECODE expression

A DECODE expression replaces a specific value with either another specific value or a default value, depending on the result of an equality condition. This operation is equivalent to the operation of a simple CASE expression or an IF-THEN-ELSE statement.

Synopsis

```
DECODE ( expression, search, result [, search, result ]... [ ,default ] )
```

This type of expression is useful for replacing abbreviations or codes that are stored in tables with meaningful business values that are needed for reports.

Parameters

- **expression**: Source of the value that you want to compare, such as a column in a table.
- **search**: The target value that is compared against the source expression, such as a numeric value or a character string. The search expression must evaluate to a single fixed value. You cannot specify an expression that evaluates to a range of values, such as `age between 20 and 29`; you need to specify separate search/result pairs for each value that you want to replace.
  
  The data type of all instances of the search expression must be the same or compatible. The `expression` and `search` parameters must also be compatible.
- **result**: The replacement value that query returns when the expression matches the search value. You must include at least one search/result pair in the DECODE expression.
  
  The data types of all instances of the result expression must be the same or compatible. The `result` and `default` parameters must also be compatible.
- **default**: An optional default value that is used for cases when the search condition fails. If you do not specify a default value, the DECODE expression returns NULL.

Usage Notes

If the `expression` value and the `search` value are both NULL, the DECODE result is the corresponding result value. See the Examples section.

Examples

When the value `2008-06-01` exists in the `START_DATE` column of `DATETABLE`, replace it with `June 1st, 2008`. Replace all other `START_DATE` values with NULL.

```
select decode(caldate, '2008-06-01', 'June 1st, 2008')
from date where month='JUN' order by caldate;
```

```
<table>
<thead>
<tr>
<th>case</th>
</tr>
</thead>
<tbody>
<tr>
<td>June 1st, 2008</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>(30 rows)</td>
</tr>
</tbody>
</table>
```
Use a DECODE expression to convert the five abbreviated CATNAME columns in the CATEGORY table to full names. Convert other values in the column to Unknown.

```sql
select catid, decode(catname, 'NHL', 'National Hockey League', 'MLB', 'Major League Baseball', 'MLS', 'Major League Soccer', 'NFL', 'National Football League', 'NBA', 'National Basketball Association', 'Unknown')
from category
order by catid;
```

<table>
<thead>
<tr>
<th>catid</th>
<th>case</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Major League Baseball</td>
</tr>
<tr>
<td>2</td>
<td>National Hockey League</td>
</tr>
<tr>
<td>3</td>
<td>National Football League</td>
</tr>
<tr>
<td>4</td>
<td>National Basketball Association</td>
</tr>
<tr>
<td>5</td>
<td>Major League Soccer</td>
</tr>
<tr>
<td>6</td>
<td>Unknown</td>
</tr>
<tr>
<td>7</td>
<td>Unknown</td>
</tr>
<tr>
<td>8</td>
<td>Unknown</td>
</tr>
<tr>
<td>9</td>
<td>Unknown</td>
</tr>
<tr>
<td>10</td>
<td>Unknown</td>
</tr>
<tr>
<td>11</td>
<td>Unknown</td>
</tr>
<tr>
<td>(11 rows)</td>
<td></td>
</tr>
</tbody>
</table>

Use a DECODE expression to find venues in Colorado and Nevada with NULL in the VENUESEATS column; convert the NULLs to zeroes. If the VENUESEATS column is not NULL, return 1 as the result.

```sql
select venuename, venuestate, decode(venueseats,null,0,1)
from venue
where venuestate in('NV','CO')
order by 2,3,1;
```

<table>
<thead>
<tr>
<th>venuename</th>
<th>venuestate</th>
<th>case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coors Field</td>
<td>CO</td>
<td>1</td>
</tr>
<tr>
<td>Dick's Sporting Goods Park</td>
<td>CO</td>
<td>1</td>
</tr>
<tr>
<td>Ellie Caulkins Opera House</td>
<td>CO</td>
<td>1</td>
</tr>
<tr>
<td>INVEESCO Field</td>
<td>CO</td>
<td>1</td>
</tr>
<tr>
<td>Pepsi Center</td>
<td>CO</td>
<td>1</td>
</tr>
<tr>
<td>Ballys Hotel</td>
<td>NV</td>
<td>0</td>
</tr>
<tr>
<td>Bellagio Hotel</td>
<td>NV</td>
<td>0</td>
</tr>
<tr>
<td>Caesars Palace</td>
<td>NV</td>
<td>0</td>
</tr>
<tr>
<td>Harrahs Hotel</td>
<td>NV</td>
<td>0</td>
</tr>
<tr>
<td>Hilton Hotel</td>
<td>NV</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(20 rows)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**GREATEST and LEAST**

Returns the largest or smallest value from a list of any number of expressions.
Synopsis

GREATEST (value [, ...])
LEAST (value [, ...])

Parameters

expression_list
A comma-separated list of expressions, such as column names. The expressions must all be
convertible to a common data type. NULL values in the list are ignored. If all of the expressions
evaluate to NULL, the result is NULL.

Returns

Returns the data type of the expressions.

Example

The following example returns the highest value alphabetically for firstname or lastname.

```sql
select firstname, lastname, greatest(firstname, lastname) from users
where userid < 10
order by 3;

<table>
<thead>
<tr>
<th>firstname</th>
<th>lastname</th>
<th>greatest</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lars</td>
<td>Ratliff</td>
<td>Ratliff</td>
</tr>
<tr>
<td>Reagan</td>
<td>Hodge</td>
<td>Reagan</td>
</tr>
<tr>
<td>Colton</td>
<td>Roy</td>
<td>Roy</td>
</tr>
<tr>
<td>Barry</td>
<td>Roy</td>
<td>Roy</td>
</tr>
<tr>
<td>Tamekah</td>
<td>Juarez</td>
<td>Tamekah</td>
</tr>
<tr>
<td>Rafael</td>
<td>Taylor</td>
<td>Taylor</td>
</tr>
<tr>
<td>Victor</td>
<td>Hernandez</td>
<td>Victor</td>
</tr>
<tr>
<td>Vladimir</td>
<td>Humphrey</td>
<td>Vladimir</td>
</tr>
<tr>
<td>Mufutau</td>
<td>Watkins</td>
<td>Watkins</td>
</tr>
</tbody>
</table>
```

(9 rows)

NVL expression

An NVL expression is identical to a COALESCE expression. NVL and COALESCE are synonyms.

Synopsis

NVL | COALESCE ( expression, expression, ... )

An NVL or COALESCE expression returns the value of the first expression in the list that is not null. If all
expressions are null, the result is null. When a non-null value is found, the remaining expressions in the
list are not evaluated.

This type of expression is useful when you want to return a backup value for something when the preferred
value is missing or null. For example, a query might return one of three phone numbers (cell, home, or
work, in that order), whichever is found first in the table (not null).
Examples

Create a table with START_DATE and END_DATE columns, insert some rows that include null values, then apply an NVL expression to the two columns.

```sql
create table datetable (start_date date, end_date date);

insert into datetable values ('2008-06-01','2008-12-31');
insert into datetable values (null,'2008-12-31');
insert into datetable values ('2008-12-31',null);

select nvl(start_date, end_date)
from datetable
order by 1;
```

<table>
<thead>
<tr>
<th>Coalesce</th>
<th>-----------</th>
</tr>
</thead>
<tbody>
<tr>
<td>2008-06-01</td>
<td></td>
</tr>
<tr>
<td>2008-12-31</td>
<td></td>
</tr>
<tr>
<td>2008-12-31</td>
<td></td>
</tr>
</tbody>
</table>

The default column name for an NVL expression is COALESCE. The following query would return the same results:

```sql
select coalesce(start_date, end_date)
from datetable
order by 1;
```

If you expect a query to return null values for certain functions or columns, you can use an NVL expression to replace the nulls with some other value. For example, aggregate functions, such as SUM, return null values instead of zeroes when they have no rows to evaluate. You can use an NVL expression to replace these null values with 0.0:

```sql
select nvl(sum(sales), 0.0) as sumresult, ...
```

NULLIF expression

Synopsis

The NULLIF expression compares two arguments and returns null if the arguments are equal. If they are not equal, the first argument is returned. This expression is the inverse of the NVL or COALESCE expression.

```sql
NULLIF ( expression1, expression2 )
```

Arguments

expression1, expression2

The target columns or expressions that are compared. The return type is the same as the type of the first expression. The default column name of the NULLIF result is the column name of the first expression.
Examples

In the following example, the query returns null when the LISTID and SALESID values match:

```
select nullif(listid, salesid), salesid
from sales where salesid<10 order by 1, 2 desc;
```

| listid | salesid |
|--------+---------|
| 4      | 2       |
| 5      | 4       |
| 5      | 3       |
| 6      | 5       |
| 10     | 9       |
| 10     | 8       |
| 10     | 7       |
| 10     | 6       |
|        | 1       |

(9 rows)

You can use NULLIF to ensure that empty strings are always returned as nulls. In the example below, the NULLIF expression returns either a null value or a string that contains at least one character.

```
insert into category
values(0,'','Special','Special');

select nullif(catgroup,'') from category
where catdesc='Special';
```

<table>
<thead>
<tr>
<th>catgroup</th>
</tr>
</thead>
<tbody>
<tr>
<td>null</td>
</tr>
</tbody>
</table>

(1 row)

NULLIF ignores trailing spaces. If a string is not empty but contains spaces, NULLIF still returns null:

```
create table nulliftest(c1 char(2), c2 char(2));

insert into nulliftest values ('a','a ');
insert into nulliftest values ('b','b');

select nullif(c1,c2) from nulliftest;
```

<table>
<thead>
<tr>
<th>c1</th>
</tr>
</thead>
<tbody>
<tr>
<td>null</td>
</tr>
<tr>
<td>null</td>
</tr>
</tbody>
</table>

(2 rows)

Date functions

Topics

- ADD_MONTHS (Oracle compatibility function) (p. 472)
This section contains the date and time scalar functions that Amazon Redshift supports.

The following leader-node only date functions are deprecated.

- AGE. Use DATEDIFF instead.
- CURRENT_TIME. Use GETDATE or SYSDATE instead.
- CURRENT_TIMESTAMP. Use GETDATE or SYSDATE instead.
- LOCALTIME. Use GETDATE or SYSDATE instead.
- LOCALTIMESTAMP. Use GETDATE or SYSDATE instead.
- ISFINITE
- NOW. Use GETDATE or SYSDATE instead.

**ADD_MONTHS** *(Oracle compatibility function)*

The ADD_MONTHS function adds the specified number of months to a date or time stamp value or expression. The DATEADD function provides similar functionality.

### Synopsis

```
ADD_MONTHS ( date, num_months )
```
Arguments

*date*

A date or time stamp expression or any value that implicitly converts to a time stamp. If *date* is the last day of the month, or if the resulting month is shorter, the function returns the last day of the month in the result. For other dates, the result contains the same day number as the *date* expression.

*num_months*

A positive or negative integer or any value that implicitly converts to an integer. Use a negative number to subtract months from dates.

Return type

ADD_MONTHS returns a TIMESTAMP.

Example

The following query uses the ADD_MONTHS function inside a TRUNC function. The TRUNC function removes the time of day from the result of ADD_MONTHS. The ADD_MONTHS function adds 12 months to each value from the CALDATE column.

```sql
select distinct trunc(add_months(caldate, 12)) as calplus12,
       trunc(caldate) as cal
from date
order by 1 asc;
```

<table>
<thead>
<tr>
<th>calplus12</th>
<th>cal</th>
</tr>
</thead>
<tbody>
<tr>
<td>2009-01-01</td>
<td>2008-01-01</td>
</tr>
<tr>
<td>2009-01-02</td>
<td>2008-01-02</td>
</tr>
<tr>
<td>2009-01-03</td>
<td>2008-01-03</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

(365 rows)

The following examples demonstrate the behavior when the ADD_MONTHS function operates on dates with months that have different numbers of days.

```sql
select add_months('2008-03-31',1);
```

```
2008-04-30 00:00:00
```

(1 row)

```sql
select add_months('2008-04-30',1);
```

```
2008-05-31 00:00:00
```

(1 row)

AGE function

AGE is a deprecated leader-node function. Use DATEDIFF function (p. 481) instead.
CONVERT_TIMEZONE function

CONVERT_TIMEZONE converts a time stamp from one time zone to another.

Syntax

CONVERT_TIMEZONE ( ['source_zone',] 'target_zone', 'timestamp')

Arguments

source_timezone
  (Optional) The time zone of the current time stamp. The default is UTC.

target_timezone
  The time zone for the new time stamp.

timestamp
  The time stamp value to be converted.

Either source_timezone or target_timezone can be specified as a time zone name (such as 'Africa/Kampala' or 'Singapore') or as a time zone abbreviation (such as 'UTC' or 'PDT').

If you specify a time zone using a time zone name, CONVERT_TIMEZONE automatically adjusts for Daylight Saving Time (DST), or any other local seasonal protocol, such as Summer Time, Standard Time, or Winter Time, that is in force for that time zone during the date and time specified by timestamp. For example, 'Europe/London' represents UTC in the winter and UTC+1 in the summer.

Time zone abbreviations represent a fixed offset from UTC. If you specify a time zone using a time zone abbreviation, CONVERT_TIMEZONE uses the fixed offset from UTC and does not adjust for any local seasonal protocol. For example, ADT (Atlantic Daylight Time) always represents UTC-03, even in winter.

For a list of supported time zone names and abbreviations, see Appendix: Time Zone Names and Abbreviations (p. 704).

Return type

TIMESTAMP.

Examples

The following example converts the time stamp value in the LISTTIME column from the default UTC time zone to PST. Even though the time stamp is within the daylight time period, it is converted to standard time because the target time zone is specified as an abbreviation (PST).

```
select listtime, convert_timezone('PST', listtime) from listing
where listid = 16;
```

<table>
<thead>
<tr>
<th>listtime</th>
<th>convert_timezone</th>
</tr>
</thead>
<tbody>
<tr>
<td>2008-08-24 09:36:12</td>
<td>2008-08-24 01:36:12</td>
</tr>
</tbody>
</table>

The following example converts a time stamp LISTTIME column from the default UTC time zone to US/Pacific time zone. The target time zone uses a time zone name, and the time stamp is within the daylight time period, so the function returns the daylight time.
select listtime, convert_timezone('US/Pacific', listtime) from listing
where listid = 16;

<table>
<thead>
<tr>
<th>listtime</th>
<th>convert_timezone</th>
</tr>
</thead>
<tbody>
<tr>
<td>2008-08-24 09:36:12</td>
<td>2008-08-24 02:36:12</td>
</tr>
</tbody>
</table>

The following example converts a time stamp string from EST to PST:

select convert_timezone('EST', 'PST', '20080305 12:25:29');

<table>
<thead>
<tr>
<th>convert_timezone</th>
</tr>
</thead>
<tbody>
<tr>
<td>2008-03-05 09:25:29</td>
</tr>
</tbody>
</table>

The following example converts a time stamp to US Eastern Standard Time because the target time zone uses a time zone name (America/New_York) and the time stamp is within the standard time period.

select convert_timezone('America/New_York', '2013-02-01 08:00:00');

<table>
<thead>
<tr>
<th>convert_timezone</th>
</tr>
</thead>
<tbody>
<tr>
<td>2013-02-01 03:00:00</td>
</tr>
</tbody>
</table>

(1 row)

The following example converts the time stamp to US Eastern Daylight Time because the target time zone uses a time zone name (America/New_York) and the time stamp is within the daylight time period.

select convert_timezone('America/New_York', '2013-06-01 08:00:00');

<table>
<thead>
<tr>
<th>convert_timezone</th>
</tr>
</thead>
<tbody>
<tr>
<td>2013-06-01 04:00:00</td>
</tr>
</tbody>
</table>

(1 row)

You can specify an offset from UTC for either source_zone or target_zone by using the format 'name+offset, where name is any string of three or more alphabetic characters, and offset is a time value, using the format hh:mm:ss. The offset can be positive or negative.

CONVERT_TIMEZONE interprets the offset as the time from UTC. For example, an offset of +2 is equivalent to UTC–2, and an offset of –2 is equivalent to UTC+2. CONVERT_TIMEZONE does not use the prefix string when calculating the offset, even if the string represents a valid time zone. For example, 'NEWZONE+2', 'PDT+2', and 'GMT+2' all have the same result. If a string does not include an offset, then it must represent a valid time zone or CONVERT_TIMEZONE returns an error.

The following example demonstrates the use of offsets.

SELECT CONVERT_TIMEZONE('GMT', 'NEWZONE +2', '2014-05-17 12:00:00') as newzone_plus_2,
CONVERT_TIMEZONE('GMT', 'NEWZONE-2:15', '2014-05-17 12:00:00') as newzone_minus_2_15,
CONVERT_TIMEZONE('GMT', 'America/Los_Angeles+2', '2014-05-17 12:00:00') as la_plus_2,
CONVERT_TIMEZONE('GMT', 'GMT+2', '2014-05-17 12:00:00') as gmt_plus_2;
CURRENT_DATE and TIMEOFDAY functions

The CURRENT_DATE and TIMEOFDAY functions are special aliases used to return date/time values.

Syntax

CURRENT_DATE

TIMEOFDAY()

Return type

- CURRENT_DATE returns a date in the default format: YYYY-MM-DD
- TIMEOFDAY() returns a VARCHAR data type and specifies the weekday, date, and time.

Examples

Return the current date:

```
select current_date;
```

```
date
------------
2008-10-01
(1 row)
```

Return the current date and time by using the TIMEOFDAY function:

```
select timeofday();
timeofday
------------
Thu Sep 19 22:53:50.333525 2013 UTC
(1 row)
```

CURRENT_TIME function

CURRENT_TIME is a deprecated leader-node function. Use GETDATE() (p. 485) or SYSDATE (p. 488) instead.

CURRENT_TIMESTAMP function

CURRENT_TIMESTAMP is a deprecated function. Use GETDATE() (p. 485) or SYSDATE (p. 488) instead.
DATE_CMP function

Compares the value of two dates and returns an integer. If the dates are identical, returns 0. If the first date is "greater", returns 1. If the second date is "greater", returns -1.

Synopsis

DATE_CMP(date1, date2)

Arguments

date1
The first input parameter is a DATE.
date2
The second parameter is a DATE.

Return type

The DATE_CMP function returns an integer.

Example

The following query compares the CALDATE column to the date January 4, 2008 and returns whether the value in CALDATE is before (-1), equal to (0), or after (1) January 4, 2008:

<table>
<thead>
<tr>
<th>caldate</th>
<th>'2008-01-04'</th>
<th>date_cmp(caldate,'2008-01-04')</th>
</tr>
</thead>
<tbody>
<tr>
<td>2008-01-01</td>
<td>2008-01-04</td>
<td>-1</td>
</tr>
<tr>
<td>2008-01-02</td>
<td>2008-01-04</td>
<td>-1</td>
</tr>
<tr>
<td>2008-01-03</td>
<td>2008-01-04</td>
<td>-1</td>
</tr>
<tr>
<td>2008-01-04</td>
<td>2008-01-04</td>
<td>0</td>
</tr>
<tr>
<td>2008-01-05</td>
<td>2008-01-04</td>
<td>1</td>
</tr>
<tr>
<td>2008-01-06</td>
<td>2008-01-04</td>
<td>1</td>
</tr>
<tr>
<td>2008-01-07</td>
<td>2008-01-04</td>
<td>1</td>
</tr>
<tr>
<td>2008-01-08</td>
<td>2008-01-04</td>
<td>1</td>
</tr>
<tr>
<td>2008-01-09</td>
<td>2008-01-04</td>
<td>1</td>
</tr>
<tr>
<td>2008-01-10</td>
<td>2008-01-04</td>
<td>1</td>
</tr>
</tbody>
</table>
(10 rows)

DATE_CMP_TIMESTAMP function

Compares the value of a date to a specified time stamp and returns an integer. If the date is "greater" alphabetically, returns 1. If the time stamp is "greater", returns -1. If the date and time stamp are identical, returns a 0.
Synopsis

DATE_CMP_TIMESTAMP (date, timestamp)

Arguments

date
The first input parameter is a DATE.

timestamp
The second parameter is a TIMESTAMP WITHOUT TIMEZONE.

Return type

The DATE_CMP_TIMESTAMP function returns an integer.

Examples

The following example compares the date 2008-06-18 to LISTTIME. Listings made before this date return a 1; listings made after this date return a -1.

```
select listid, '2008-06-18', listtime,
date_cmp_timestamp('2008-06-18', listtime)
from listing
order by 1, 2, 3, 4
limit 10;
```

| listid | ?column? |      listtime       | date_cmp_timestamp |
|--------+----------+---------------------+-------------------|
| 1      | 2008-06-18 | 2008-01-24 06:43:29 |                  1 |
| 2      | 2008-06-18 | 2008-03-05 12:25:29 |                  1 |
| 3      | 2008-06-18 | 2008-11-01 07:35:33 |                 -1 |
| 4      | 2008-06-18 | 2008-05-24 01:18:37 |                  1 |
| 5      | 2008-06-18 | 2008-05-17 02:29:11 |                  1 |
| 6      | 2008-06-18 | 2008-08-15 02:08:13 |                 -1 |
| 7      | 2008-06-18 | 2008-11-15 09:38:15 |                 -1 |
| 8      | 2008-06-18 | 2008-05-09 05:07:30 |                 -1 |
| 9      | 2008-06-18 | 2008-09-09 08:03:36 |                 -1 |
| 10     | 2008-06-18 | 2008-06-17 09:44:54 |                  1 |

(10 rows)

DATE_PART_YEAR function

The DATE_PART_YEAR function extracts the year from a date.

Synopsis

DATE_PART_YEAR (date)

Argument

date
The input parameter is a DATE.
Return type

The DATE_PART_YEAR function returns an integer.

Examples

The following example extracts the year from the CALDATE column:

```sql
select caldate, date_part_year(caldate)
from date
order by
dateid limit 10;
```

<table>
<thead>
<tr>
<th>caldate</th>
<th>date_part_year</th>
</tr>
</thead>
<tbody>
<tr>
<td>2008-01-01</td>
<td>2008</td>
</tr>
<tr>
<td>2008-01-02</td>
<td>2008</td>
</tr>
<tr>
<td>2008-01-03</td>
<td>2008</td>
</tr>
<tr>
<td>2008-01-04</td>
<td>2008</td>
</tr>
<tr>
<td>2008-01-05</td>
<td>2008</td>
</tr>
<tr>
<td>2008-01-06</td>
<td>2008</td>
</tr>
<tr>
<td>2008-01-07</td>
<td>2008</td>
</tr>
<tr>
<td>2008-01-08</td>
<td>2008</td>
</tr>
<tr>
<td>2008-01-09</td>
<td>2008</td>
</tr>
<tr>
<td>2008-01-10</td>
<td>2008</td>
</tr>
</tbody>
</table>
(10 rows)

DATEADD function

Given a datepart and an expression, this function increments date or time stamp values by a specified interval.

Synopsis

```sql
DATEADD ( datepart, interval, expression )
```

This function returns a time stamp data type.

Arguments

- **datepart**
  - The specific part of the date value (year, month, or day, for example) that the function operates on. See Dateparts for date or time stamp functions (p. 491).

- **interval**
  - An integer that defines the increment (how many days, for example) to add to the target expression. A negative integer subtracts the interval from the date value.

- **expression**
  - The target column or expression that the function operates on. The expression must be a date or time stamp expression that contains the specified datepart.

Return type

DATEADD returns a TIMESTAMP WITHOUT TIMEZONE.
Examples

Add 30 days to each date in November that exists in the DATE table:

```sql
select dateadd(day,30,caldate) as novplus30
from date
where month='NOV'
order by dateid;
```

```
novplus30
---------------------
2008-12-01 00:00:00
2008-12-02 00:00:00
2008-12-03 00:00:00
...
(30 rows)
```

Add 18 months to a literal date value:

```sql
select dateadd(month,18,'2008-02-28');
```

```
date_add
---------------------
2009-08-28 00:00:00
(1 row)
```

The default column name for a DATEADD function is DATE_ADD. The default time stamp for a date value is 00:00:00.

Add 30 minutes to a date value that does not specify a time stamp:

```sql
select dateadd(m,30,'2008-02-28');
```

```
date_add
---------------------
2008-02-28 00:30:00
(1 row)
```

You can name dateparts in full or abbreviate them; in this case, \( m \) stands for minutes, not months.

Usage notes

The DATEADD(month, ...) and ADD_MONTHS functions handle dates that fall at the ends of months differently.

- **ADD_MONTHS**: If the date you are adding to is the last day of the month, the result is always the last day of the result month, regardless of the length of the month. For example, April 30th + 1 month is May 31st:

  ```sql
  select add_months('2008-04-30',1);
  ```

  ```
  add_months
  ---------------------
  2008-05-31 00:00:00
  (1 row)
  ```
DATEADD: If there are fewer days in the date you are adding to than in the result month, the result will be the corresponding day of the result month, not the last day of that month. For example, April 30th + 1 month is May 30th:

```sql
select dateadd(month,1,'2008-04-30');

<table>
<thead>
<tr>
<th>date_add</th>
</tr>
</thead>
<tbody>
<tr>
<td>2008-05-30 00:00:00</td>
</tr>
</tbody>
</table>
```

DATEDIFF function

Given two target expressions and a datepart, this function returns the difference between the two expressions.

**Synopsis**

DATEDIFF ( \( datepart \), \( expression \), \( expression \) )

**Arguments**

- **datepart**
  - The specific part of the date value (year, month, or day, for example) that the function operates on. See Dateparts for date or time stamp functions (p. 491). Specifically, DATEDIFF determines the number of datepart boundaries that are crossed between two expressions. For example, if you are calculating the difference in years between two dates, 12-31-2008 and 01-01-2009, the function returns 1 year despite the fact that these dates are only one day apart. If you are finding the difference in hours between two time stamps, 01-01-2009 8:30:00 and 01-01-2009 10:00:00, the result is 2 hours.

- **expression**
  - The target columns or expressions that the function operates on. The expressions must be date or time stamp expressions and they must both contain the specified datepart. If the second date is later than the first date, the result is positive. If the second date is earlier than the first date, the result is negative.

**Return type**

DATEDIFF returns an integer.

**Examples**

Find the difference, in number of weeks, between two literal date values:

```sql
select datediff(week,'2009-01-01','2009-12-31') as numweeks;

<table>
<thead>
<tr>
<th>numweeks</th>
</tr>
</thead>
<tbody>
<tr>
<td>52</td>
</tr>
</tbody>
</table>
```

API Version 2012-12-01
Find the difference, in number of quarters, between a literal value in the past and today's date. This example assumes that the current date is June 5, 2008. You can name dateparts in full or abbreviate them. The default column name for the DATEDIFF function is DATE_DIFF.

```
select datediff(qtr, '1998-07-01', current_date);
```

```
<table>
<thead>
<tr>
<th>date_diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
</tr>
</tbody>
</table>
```

This example joins the SALES and LISTING tables to calculate how many days after they were listed any tickets were sold for listings 1000 through 1005. The longest wait for sales of these listings was 15 days, and the shortest was less than one day (0 days).

```
select priceperticket, 
datediff(day, listtime, saletime) as wait 
from sales, listing where sales.listid = listing.listid 
and sales.listid between 1000 and 1005 
order by wait desc, priceperticket desc;
```

```
<table>
<thead>
<tr>
<th>priceperticket</th>
<th>wait</th>
</tr>
</thead>
<tbody>
<tr>
<td>96.00</td>
<td>15</td>
</tr>
<tr>
<td>123.00</td>
<td>11</td>
</tr>
<tr>
<td>131.00</td>
<td>9</td>
</tr>
<tr>
<td>123.00</td>
<td>6</td>
</tr>
<tr>
<td>129.00</td>
<td>4</td>
</tr>
<tr>
<td>96.00</td>
<td>4</td>
</tr>
<tr>
<td>96.00</td>
<td>0</td>
</tr>
</tbody>
</table>
```

This example calculates the average number of hours sellers waited for any and all ticket sales:

```
select avg(datediff(hours, listtime, saletime)) as avgwait 
from sales, listing 
where sales.listid = listing.listid;
```

```
<table>
<thead>
<tr>
<th>avgwait</th>
</tr>
</thead>
<tbody>
<tr>
<td>465</td>
</tr>
</tbody>
</table>
```

### DATE_PART function

This function extracts datepart values from an expression. Synonym of the PGDATE_PART function.

#### Synopsis

```
DATE_PART ( datepart, expression )
```

API Version 2012-12-01

482
Arguments

datepart
   The specific part of the date value (year, month, or day, for example) that the function operates on. See Dateparts for date or time stamp functions (p. 491).

eexpression
   The target column or expression that the function operates on. The expression must be a date or time stamp expression that contains the specified datepart.

Return type

DATE_PART returns a DOUBLE PRECISION number.

Examples

Apply the DATE_PART function to a column in a table:

```sql
select date_part(w, listtime) as weeks, listtime
from listing where listid=10;
```

<table>
<thead>
<tr>
<th>weeks</th>
<th>listtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>2008-06-17 09:44:54</td>
</tr>
</tbody>
</table>

(1 row)

You can name dateparts in full or abbreviate them; in this case, w stands for weeks.

Use DATE_PART with dow (DAYOFWEEK) to view events on a Saturday. (DOW returns an integer from 0-6):

```sql
select date_part(dow, starttime) as dow, starttime
from event
where date_part(dow, starttime)=6
order by 2,1;
```

<table>
<thead>
<tr>
<th>dow</th>
<th>starttime</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>2008-01-05 14:00:00</td>
</tr>
<tr>
<td>6</td>
<td>2008-01-05 14:00:00</td>
</tr>
<tr>
<td>6</td>
<td>2008-01-05 14:00:00</td>
</tr>
<tr>
<td>6</td>
<td>2008-01-05 14:00:00</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

(1147 rows)

Apply the DATE_PART function to a literal date value:

```sql
select date_part(minute, '2009-01-01 02:08:01');
```

pgdate_part

8

(1 row)

The default column name for the DATE_PART function is PGDATE_PART.
DATE_TRUNC function

The DATE_TRUNC function truncates any time stamp expression or literal based on the time interval that you specify, such as hour, week, or month. Truncates means that the time stamp is reduced to the first of the specified year for year, first of the specified month for month, and so on.

Synopsis

\[
\text{DATE_TRUNC('field', source)}
\]

Arguments

field

The first parameter designates the precision to which to truncate the time stamp value. See Dateparts for date or time stamp functions (p. 491) for valid formats.

source

The second parameter is a time stamp value or expression.

Return type

The DATE_TRUNC function returns a time stamp.

Example

The following query on the sales table uses a saletime column that indicates when a sales transaction happened. The query finds the total sales for all transactions in September 2008 grouped by the week in which the sale occurred:

```
select date_trunc('week', saletime), sum(pricepaid) from sales where
saletime like '2008-09%' group by date_trunc('week', saletime) order by 1;
```

```
date_trunc     |    sum
---------------------+------------
2008-09-01 00:00:00 | 2474899.00
2008-09-08 00:00:00 | 2412354.00
2008-09-15 00:00:00 | 2364707.00
2008-09-22 00:00:00 | 2359351.00
2008-09-29 00:00:00 |  705249.00
(5 rows)
```

EXTRACT function

The EXTRACT function returns a date part, such as a day, month, or year, from a time stamp value or expression.

Synopsis

\[
\text{EXTRACT (datepart FROM}
\]

{ TIMESTAMP 'literal' | timestamp }\)

Arguments

datepart
   See Dateparts for date or time stamp functions (p. 491).

literal
   A time stamp value, enclosed in quotes and preceded by the TIMESTAMP keyword.

timestamp
   A time stamp column or expression.

Return type

EXTRACT returns a DOUBLE PRECISION number.

Examples

Determine the week numbers for sales in which the price paid was $10,000 or more.

```sql
select salesid, extract(week from saletime) as weeknum
from sales where pricepaid > 9999 order by 2;
```

<table>
<thead>
<tr>
<th>salesid</th>
<th>weeknum</th>
</tr>
</thead>
<tbody>
<tr>
<td>159073</td>
<td>6</td>
</tr>
<tr>
<td>160318</td>
<td>8</td>
</tr>
<tr>
<td>161723</td>
<td>26</td>
</tr>
</tbody>
</table>

(3 rows)

Return the minute value from a literal time stamp value:

```sql
select extract(minute from timestamp '2009-09-09 12:08:43');
```

<table>
<thead>
<tr>
<th>date_part</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
</tr>
</tbody>
</table>

(1 row)

GETDATE()

GETDATE returns the current date and time according to the system clock on the leader node.

- The GETDATE() function is similar to the SYSDATE function; however, GETDATE() does not include microseconds in its return value.
- The functions CURRENT_DATE and TRUNC(GETDATE()) produce the same results.

Synopsis

```sql
GETDATE()
```

The parentheses are required.

Return type

GETDATE returns a TIMESTAMP.
Examples

The following example uses the GETDATE() function to return the full time stamp for the current date:

```sql
select getdate();
```

```
timestamp
---------------------
2008-12-04 16:10:43
(1 row)
```

The following example uses the GETDATE() function inside the TRUNC function to return the current date without the time:

```sql
select trunc(getdate());
```

```
trunc
-------
2008-12-04
(1 row)
```

**INTERVAL_CMP function**

Compares two intervals. If the first interval is greater, returns a 1, if the second interval is greater, returns a -1, and if the intervals are equal, returns 0.

**Syntax**

```sql
INTERVAL_CMP(interval1, interval2)
```

**Arguments**

- `interval1`:
  - The first input parameter is an INTERVAL.
- `interval2`:
  - The second parameter is an INTERVAL.

**Return type**

The INTERVAL_CMP function returns an integer.

**Examples**

The following example compares the value of "3 days" to "1 year":

```sql
select interval_cmp('3 days','1 year');
```

```
interval_cmp
-------------
-1
```

This example compares the value "7 days" to "1 week":

```sql
select interval_cmp('7 days','1 week');
```

```
interval_cmp
-------------
0
```
select interval_cmp('7 days','1 week');

interval_cmp
--------------
   0
   (1 row)

**ISFINITE function**

ISFINITE is a deprecated leader-node function.

**LAST_DAY (Oracle compatibility function)**

Returns the date of the last day of the month that contains `date`. The return type is always DATE, regardless of the data type of the `date` argument.

**Synopsis**

```
LAST_DAY (date)
```

**Arguments**

`date`

A date or time stamp expression.

**Return type**

DATE

**Examples**

The following example returns the date of the last day in the current month:

```
select last_day(sysdate);

last_day
---------
2014-01-31
   (1 row)
```

The following example returns the number of tickets sold for each of the last 7 days of the month:

```
select datediff(day, saletime, last_day(saletime)) as "Days Remaining",
       sum(qtysold)
from sales
where datediff(day, saletime, last_day(saletime)) < 7
group by 1
order by 1;

days remaining | sum
---------------|-------
     0          | 10140
     1          | 11187
LOCALTIME function

LOCALTIME is a deprecated leader-node function. Use GETDATE() (p. 485) or SYSDATE (p. 488) instead.

LOCALTIMESTAMP function

LOCALTIMESTAMP is a deprecated leader-node function. Use GETDATE() (p. 485) or SYSDATE (p. 488) instead.

NOW function

NOW is a deprecated leader-node function. Use GETDATE() (p. 485) or SYSDATE (p. 488) instead.

SYSDATE (Oracle compatibility function)

SYSDATE returns the current date and time according to the system clock on the leader node. The functions CURRENT_DATE and TRUNC(SYSDATE) produce the same results.

Synopsis

SYSDATE

This function requires no arguments.

Return type

SYSDATE returns TIMESTAMP.

Examples

The following example uses the SYSDATE function to return the full time stamp for the current date:

```
select sysdate;
timestamp
---------------------
2008-12-04 16:10:43.976353
(1 row)
```

The following example uses the SYSDATE function inside the TRUNC function to return the current date without the time:

```
select trunc(sysdate);
trunc
-------
```
The following query returns sales information for dates that fall between the date when the query is issued and whatever date is 120 days earlier:

```sql
select salesid, pricepaid, trunc(saletime) as saletime, trunc(sysdate) as now
from sales
where saletime between trunc(sysdate)-120 and trunc(sysdate)
order by saletime asc;
```

<table>
<thead>
<tr>
<th>salesid</th>
<th>pricepaid</th>
<th>saletime</th>
<th>now</th>
</tr>
</thead>
<tbody>
<tr>
<td>91535</td>
<td>670.00</td>
<td>2008-08-07</td>
<td>2008-12-05</td>
</tr>
<tr>
<td>91635</td>
<td>365.00</td>
<td>2008-08-07</td>
<td>2008-12-05</td>
</tr>
<tr>
<td>91901</td>
<td>1002.00</td>
<td>2008-08-07</td>
<td>2008-12-05</td>
</tr>
</tbody>
</table>
...      |

**TIMESTAMP_CMP function**

Compares the value of two time stamps and returns an integer. If the timestamps are identical, returns 0. If the first time stamp is "greater", returns 1. If the second time stamp is "greater", returns -1.

**Synopsis**

```sql
TIMESTAMP_CMP(timestamp1, timestamp2)
```

**Arguments**

- `timestamp1` The first input parameter is a TIMESTAMP WITHOUT TIMEZONE.
- `timestamp2` The second parameter is a TIMESTAMP WITHOUT TIMEZONE.

**Return type**

The TIMESTAMP_CMP function returns an integer.

**Examples**

The following example compares the LISTTIME and SALETIME for a listing. Note that the value for TIMESTAMP_CMP is -1 for all listings because the time stamp for the sale is after the time stamp for the listing:

```sql
select listing.listid, listing.listtime,
       sales(saletime, timestamp_cmp(listing.listtime, sales.saletime))
from listing, sales
where listing.listid=sales.listid
order by 1, 2, 3, 4
limit 10;
```

<table>
<thead>
<tr>
<th>listid</th>
<th>listtime</th>
<th>saletime</th>
<th>timestamp_cmp</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
...
This example shows that TIMESTAMP_CMP returns a 0 for identical time stamps:

```
select listid, timestamp_cmp(listtime, listtime)
from listing
order by 1, 2
limit 10;
```

<table>
<thead>
<tr>
<th>listid</th>
<th>timestamp_cmp</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
</tbody>
</table>

(10 rows)

**TIMESTAMP_CVM_DATE function**

Compares the value of a time stamp to a specified date and returns an integer. If the first time stamp is "greater" alphabetically, returns 1. If the second time stamp is "greater", returns -1. If the time stamp and date are identical, returns a 0.

**Syntax**

```
TIMESTAMP_CVM_DATE(timestamp, date)
```

**Arguments**

- `timestamp`
  - The first input parameter is a TIMESTAMP WITHOUT TIMEZONE.
- `date`
  - The second parameter is a DATE.

**Return type**

The TIMESTAMP_CVM_DATE function returns an integer.
Examples

The following example compares LISTTIME to the date 2008-06-18. Listings made after this date return a 1; listings made before this date return a -1.

```
select listid, listtime,
timestamp_cmp_date(listtime, '2008-06-18')
from listing
order by 1, 2, 3
limit 10;
```

<table>
<thead>
<tr>
<th>listid</th>
<th>listtime</th>
<th>timestamp_cmp_date</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2008-01-24 06:43:29</td>
<td>-1</td>
</tr>
<tr>
<td>2</td>
<td>2008-03-05 12:25:29</td>
<td>-1</td>
</tr>
<tr>
<td>3</td>
<td>2008-11-01 07:35:33</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>2008-05-24 01:18:37</td>
<td>-1</td>
</tr>
<tr>
<td>5</td>
<td>2008-05-17 02:29:11</td>
<td>-1</td>
</tr>
<tr>
<td>6</td>
<td>2008-08-15 02:08:13</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>2008-11-15 09:38:15</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>2008-11-09 05:07:30</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>2008-09-09 08:03:36</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>2008-06-17 09:44:54</td>
<td>-1</td>
</tr>
</tbody>
</table>
(10 rows)

TRUNC(timestamp)

See TRUNC function (p. 516).

Dateparts for date or time stamp functions

The following table identifies the datepart and timepart names and abbreviations that are accepted as arguments to the following functions:

- DATEADD
- DATEDIFF
- DATEPART
- DATE_PART
- DATE_TRUNC
- EXTRACT

<table>
<thead>
<tr>
<th>Datepart or timepart</th>
<th>Abbreviations</th>
</tr>
</thead>
<tbody>
<tr>
<td>millennium, millennia</td>
<td>mil, mils</td>
</tr>
<tr>
<td>century, centuries</td>
<td>c, cent, cents</td>
</tr>
<tr>
<td>decade, decades</td>
<td>dec, decs</td>
</tr>
<tr>
<td>epoch</td>
<td>epoch (supported by the DATEPART, DATE_PART, and EXTRACT functions)</td>
</tr>
<tr>
<td>year, years</td>
<td>y, yr, yrs</td>
</tr>
</tbody>
</table>
## Datepart or timepart

<table>
<thead>
<tr>
<th>Datepart or timepart</th>
<th>Abbreviations</th>
</tr>
</thead>
<tbody>
<tr>
<td>quarter, quarters</td>
<td>qtr, qtrs</td>
</tr>
<tr>
<td>month, months</td>
<td>mon, mons</td>
</tr>
<tr>
<td>week, weeks</td>
<td>w</td>
</tr>
<tr>
<td>day of week</td>
<td>dayofweek, dow, dw, weekday (supported by the DATEPART, DATE_PART, and EXTRACT functions)</td>
</tr>
<tr>
<td>day of year</td>
<td>dayofyear, doy, dy, yearday (supported by the DATEPART, DATE_PART, and EXTRACT functions)</td>
</tr>
<tr>
<td>day, days</td>
<td>d</td>
</tr>
<tr>
<td>hour, hours</td>
<td>h, hr, hrs</td>
</tr>
<tr>
<td>minute, minutes</td>
<td>m, min, mins</td>
</tr>
<tr>
<td>second, seconds</td>
<td>s, sec, secs</td>
</tr>
<tr>
<td>millisecond, milliseconds</td>
<td>ms, msec, msecs, msecond, milliseconds, millisecond, millisecs, millisecond</td>
</tr>
<tr>
<td>microsecond, microseconds</td>
<td>microsec, microsecs, microsecond, usecond, useconds, us, usec, usecs</td>
</tr>
</tbody>
</table>

### Unsupported dateparts

Amazon Redshift does not support the time-zone dateparts (such as `timezone` or `timezone_hour`).

### Variations in results with seconds, milliseconds, and microseconds

Minor differences in query results occur when different date functions specify seconds, milliseconds, or microseconds as dateparts:

- The `EXTRACT` and `DATEPART` functions return integers for the specified datepart only, ignoring higher- and lower-level dateparts. If the specified datepart is seconds, milliseconds and microseconds are not included in the result. If the specified datepart is milliseconds, seconds and microseconds are not included. If the specified datepart is microseconds, seconds and milliseconds are not included.
- The `DATE_PART` function returns the complete seconds portion of the time stamp, regardless of the specified datepart, returning either a decimal value or an integer as required.

For example, compare the results of the following queries:

```sql
create table seconds(micro timestamp);
insert into seconds values('2009-09-21 11:10:03.189717');
select extract(sec from micro) from seconds;
date_part
------------
3
(1 row)

select date_part(sec, micro) from seconds;
pdate_part
------------
```
### CENTURY, EPOCH, DECADE, and MIL notes

#### CENTURY or CENTURIES
Amazon Redshift interprets a CENTURY to start with year ###1 and end with year ###0:

```sql
select extract (century from timestamp '2000-12-16 12:21:13');
date_part
-----------
20
(1 row)
```

```sql
select extract (century from timestamp '2001-12-16 12:21:13');
date_part
-----------
21
(1 row)
```

#### EPOCH notes
The Amazon Redshift implementation of EPOCH is relative to 1970-01-01 00:00:00.000000 independent of the time zone where the server resides. You may need to offset the results by the difference in hours depending on the time zone where the server is located.

#### DECADE or DECADES
Amazon Redshift interprets the DECADE or DECADES DATEPART based on the common calendar. For example, because the common calendar starts from the year 1, the first decade (decade 1) is 0001-01-01 through 0009-12-31, and the second decade (decade 2) is 0010-01-01 through 0019-12-31. For example, decade 201 spans from 2000-01-01 to 2009-12-31:

```sql
select extract(decade from timestamp '1999-02-16 20:38:40');
date_part
-----------
200
(1 row)
```

```sql
select extract(decade from timestamp '2000-02-16 20:38:40');
date_part
-----------
201
(1 row)
```

```sql
select extract(decade from timestamp '2010-02-16 20:38:40');
date_part
-----------
202
(1 row)
```

#### MIL or MILS
Amazon Redshift interprets a MIL to start with the first day of year #001 and end with the last day of year #000:

```sql
select extract (mil from timestamp '2000-12-16 12:21:13');
date_part
-----------
001
(1 row)
```

```sql
select extract (mil from timestamp '2001-12-16 12:21:13');
date_part
-----------
002
(1 row)
```

```sql
select extract (mil from timestamp '2002-12-16 12:21:13');
date_part
-----------
003
(1 row)
```
Math functions

Topics

- Mathematical operator symbols (p. 495)
- ABS function (p. 496)
- ACOS function (p. 497)
- ASIN function (p. 498)
- ATAN function (p. 498)
- ATAN2 function (p. 499)
- CBRT function (p. 500)
- CEILING (or CEIL) function (p. 500)
- CHECKSUM function (p. 501)
- COS function (p. 502)
- COT function (p. 502)
- DEGREES function (p. 503)
- DEXP function (p. 504)
- DLOG1 function (p. 504)
- DLOG10 function (p. 504)
- EXP function (p. 505)
- FLOOR function (p. 506)
- LN function (p. 506)
- LOG function (p. 507)
- MOD function (p. 508)
- PI function (p. 509)
- POWER function (p. 509)
- RADIANS function (p. 510)
- RANDOM function (p. 511)
- ROUND function (p. 512)
- SIN function (p. 514)
- SIGN function (p. 514)
- SQRT function (p. 515)
- TAN function (p. 516)
- TRUNC function (p. 516)

This section describes the mathematical operators and functions supported in Amazon Redshift.
Mathematical operator symbols

The following table lists the supported mathematical operators.

**Supported operators**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>addition</td>
<td>2 + 3</td>
<td>5</td>
</tr>
<tr>
<td>-</td>
<td>subtraction</td>
<td>2 - 3</td>
<td>-1</td>
</tr>
<tr>
<td>*</td>
<td>multiplication</td>
<td>2 * 3</td>
<td>6</td>
</tr>
<tr>
<td>/</td>
<td>division</td>
<td>4 / 2</td>
<td>2</td>
</tr>
<tr>
<td>%</td>
<td>modulo</td>
<td>5 % 4</td>
<td>1</td>
</tr>
<tr>
<td>^</td>
<td>exponentiation</td>
<td>2.0 ^ 3.0</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>/</td>
<td>square root</td>
<td>/ 25.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>/</td>
<td>cube root</td>
</tr>
<tr>
<td>@</td>
<td>absolute value</td>
<td>@ -5.0</td>
<td>5</td>
</tr>
<tr>
<td>&lt;&lt;&lt;</td>
<td>bitwise shift left</td>
<td>1 &lt;&lt; 4</td>
<td>16</td>
</tr>
<tr>
<td>&gt;&gt;=</td>
<td>bitwise shift right</td>
<td>8 &gt;&gt; 2</td>
<td>2</td>
</tr>
</tbody>
</table>

**Examples**

Calculate the commission paid plus a $2.00 handling for a given transaction:

```sql
select commission, (commission + 2.00) as comm
from sales where salesid=10000;
```

<table>
<thead>
<tr>
<th>commission</th>
<th>comm</th>
</tr>
</thead>
<tbody>
<tr>
<td>28.05</td>
<td>30.05</td>
</tr>
</tbody>
</table>

(1 row)

Calculate 20% of the sales price for a given transaction:

```sql
select pricepaid, (pricepaid * .20) as twentypct
from sales where salesid=10000;
```

<table>
<thead>
<tr>
<th>pricepaid</th>
<th>twentypct</th>
</tr>
</thead>
<tbody>
<tr>
<td>187.00</td>
<td>37.400</td>
</tr>
</tbody>
</table>

(1 row)

Forecast ticket sales based on a continuous growth pattern. In this example, the subquery returns the number of tickets sold in 2008. That result is multiplied exponentially by a continuous growth rate of 5% over 10 years.
select (select sum(qtysold) from sales, date
where sales.dateid=date.dateid and year=2008)
^ ((5::float/100)*10) as qty10years;

qty10years
----------
587.664019657491
(1 row)

Find the total price paid and commission for sales with a date ID that is greater than or equal to 2000. Then subtract the total commission from the total price paid.

select sum (pricepaid) as sum_price, dateid,
sum (commission) as sum_comm, (sum (pricepaid) - sum (commission)) as value
from sales where dateid >= 2000
group by dateid order by dateid limit 10;

<table>
<thead>
<tr>
<th>sum_price</th>
<th>dateid</th>
<th>sum_comm</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>364445.00</td>
<td>2044</td>
<td>54666.75</td>
<td>309778.25</td>
</tr>
<tr>
<td>349344.00</td>
<td>2112</td>
<td>52401.60</td>
<td>296942.40</td>
</tr>
<tr>
<td>343756.00</td>
<td>2124</td>
<td>51563.40</td>
<td>292192.60</td>
</tr>
<tr>
<td>378595.00</td>
<td>2116</td>
<td>56789.25</td>
<td>321805.75</td>
</tr>
<tr>
<td>328725.00</td>
<td>2080</td>
<td>49308.75</td>
<td>279416.25</td>
</tr>
<tr>
<td>349554.00</td>
<td>2028</td>
<td>52433.10</td>
<td>297120.90</td>
</tr>
<tr>
<td>249207.00</td>
<td>2164</td>
<td>37381.05</td>
<td>211825.95</td>
</tr>
<tr>
<td>285202.00</td>
<td>2064</td>
<td>42780.30</td>
<td>242421.70</td>
</tr>
<tr>
<td>320945.00</td>
<td>2012</td>
<td>48141.75</td>
<td>272803.25</td>
</tr>
<tr>
<td>321096.00</td>
<td>2016</td>
<td>48164.40</td>
<td>272931.60</td>
</tr>
</tbody>
</table>

(10 rows)

**ABS function**

ABS calculates the absolute value of a number, where that number can be a literal or an expression that evaluates to a number.

**Synopsis**

ABS (number)

**Arguments**

*number*

  Number or expression that evaluates to a number.

**Return type**

ABS returns the same data type as its argument.

**Examples**

Calculate the absolute value of -38:
select abs (-38);  
abs  
-------  
38  
(1 row)  

Calculate the absolute value of (14-76):  
select abs (14-76);  
abs  
-------  
62  
(1 row)  

**ACOS function**  
ACOS is a trigonometric function that returns the arc cosine of a number. The return value is in radians and is between PI/2 and -PI/2.  

**Synopsis**  

```
ACOS(number)
```

**Arguments**  

*number*  
The input parameter is a double precision number.  

**Return type**  
The ACOS function returns a double precision number.  

**Examples**  
The following example returns the arc cosine of -1:  

```
select acos(-1);  
acos  
------------------  
3.14159265358979  
(1 row)
```

The following example converts the arc cosine of .5 to the equivalent number of degrees:  

```
select (acos(.5) * 180/(select pi())) as degrees;  
degrees  
---------  
60  
(1 row)
```
ASIN function

ASIN is a trigonometric function that returns the arc sine of a number. The return value is in radians and is between PI/2 and -PI/2.

Synopsis

ASIN(number)

Argument

number
  The input parameter is a double precision number.

Return type

The ASIN function returns a double precision number.

Examples

The following example returns the arc sine of 1 and multiples it by 2:

```sql
select asin(1)*2 as pi;
pi
------------------
3.14159265358979
(1 row)
```

The following example converts the arc sine of .5 to the equivalent number of degrees:

```sql
select (asin(.5) * 180/(select pi())) as degrees;
degrees
--------
30
(1 row)
```

ATAN function

ATAN is a trigonometric function that returns the arc tangent of a number. The return value is in radians and is between PI/2 and -PI/2.

Synopsis

ATAN(number)

Argument

number
  The input parameter is a double precision number.
Return type

The ATAN function returns a double precision number.

Examples

The following example returns the arc tangent of 1 and multiplies it by 4:

```
select atan(1) * 4 as pi;
pi
-------------
3.14159265358979  
(1 row)
```

The following example converts the arc tangent of 1 to the equivalent number of degrees:

```
select (atan(1) * 180/(select pi())) as degrees;
degrees
---------
45
(1 row)
```

ATAN2 function

ATAN2 is a trigonometric function that returns the arc tangent of a one number divided by another number. The return value is in radians and is between PI/2 and -PI/2.

Synopsis

```
ATAN2(number1, number2)
```

Arguments

- `number1`  
  The first input parameter is a double precision number.
- `number2`  
  The second parameter is a double precision number.

Return type

The ATAN2 function returns a double precision number.

Examples

The following example returns the arc tangent of 2/2 and multiplies it by 4:

```
select atan2(2,2) * 4 as pi;
pi
-----------------
3.14159265358979  
(1 row)
```

The following example converts the arc tangent of 1/0 (or 0) to the equivalent number of degrees:
select (atan2(1,0) * 180/(select pi())) as degrees;
degrees
---------
90
(1 row)

**CBRT function**

The CBRT function is a mathematical function that calculates the cube root of a number.

**Synopsis**

CBRT (number)

**Argument**

CBRT takes a DOUBLE PRECISION number as an argument.

**Return type**

CBRT returns a DOUBLE PRECISION number.

**Examples**

Calculate the cube root of the commission paid for a given transaction:

```sql
select cbrt(commission) from sales where salesid=10000;
cbrt
------------------
3.03839539048843
(1 row)
```

**CEILING (or CEIL) function**

The CEILING or CEIL function is used to round a number up to the next whole number. (The FLOOR function (p. 506) rounds a number down to the next whole number.)

**Synopsis**

CEIL | CEILING(number)

**Arguments**

number

DOUBLE PRECISION number to be rounded.

**Return type**

CEILING and CEIL return an integer.
Example

Calculate the ceiling of the commission paid for a given sales transaction:

```
select ceiling(commission) from sales
where salesid=10000;
```

```
   ceiling
----------
      29
(1 row)
```

CHECKSUM function

Computes a checksum value for building a hash index.

Synopsis

```
CHECKSUM(expression)
```

Argument

`expression`

The input expression must be a VARCHAR, INTEGER, or DECIMAL data type.

Return type

The CHECKSUM function returns an integer.

Example

The following example computes a checksum value for the COMMISSION column:

```
select checksum(commission)
from sales
order by salesid
limit 10;
```

```
   checksum
----------
   10920
   1140
   5250
   2625
   2310
   5910
  11820
   2955
  8865
   975
(10 rows)
```
**COS function**

COS is a trigonometric function that returns the cosine of a number. The return value is in radians and is between PI/2 and -PI/2.

**Synopsis**

```
COS(double_precision)
```

**Argument**

`number`

The input parameter is a double precision number.

**Return type**

The COS function returns a double precision number.

**Examples**

The following example returns cosine of 0:

```
select cos(0);
```

```
+-----+
| cos  |
+-----+
| 1    |
+-----+

(1 row)

The following example returns the cosine of PI:

```
select cos(pi());
```

```
+-----+
| cos  |
+-----+
| -1   |
+-----+

(1 row)

**COT function**

COT is a trigonometric function that returns the cotangent of a number. The input parameter must be non-zero.

**Synopsis**

```
COT(number)
```

**Argument**

`number`

The input parameter is a double precision number.
Return type

The COT function returns a double precision number.

Examples

The following example returns the cotangent of 1:

```
select cot(1);
cot
-------------------
0.642092615934331
(1 row)
```

DEGREES function

Converts an angle in radians to its equivalent in degrees.

Synopsis

```
DEGREES(number)
```

Argument

```
number
```

The input parameter is a double precision number.

Return type

The DEGREES function returns a double precision number.

Examples

The following example returns the degree equivalent of .5 radians:

```
select degrees(.5);
degrees
-----------------
28.6478897565412
(1 row)
```

The following example converts PI radians to degrees:

```
select degrees(pi());
degrees
-------
180
(1 row)
```
**DEXP function**

The DEXP function returns the exponential value in scientific notation for a double precision number. The only difference between the DEXP and EXP functions is that the parameter for DEXP must be a double precision.

**Synopsis**

| DEXP(number) |

**Argument**

*number*

The input parameter is a double precision number.

**Return type**

The DEXP function returns a double precision number.

**Example**

Use the DEXP function to forecast ticket sales based on a continuous growth pattern. In this example, the subquery returns the number of tickets sold in 2008. That result is multiplied by the result of the DEXP function, which specifies a continuous growth rate of 7% over 10 years.

```sql
select (select sum(qtysold) from sales, date
where sales.dateid=date.dateid
and year=2008) * dexp((7::float/100)*10) qty2010;
```

<table>
<thead>
<tr>
<th>qty2010</th>
</tr>
</thead>
<tbody>
<tr>
<td>695447.483772222</td>
</tr>
</tbody>
</table>

(1 row)

**DLOG1 function**

The DLOG1 function returns the natural logarithm of the input parameter. Synonym for the LN function.

**Synopsis**

| DLOG1(number) |

**DLOG10 function**

The DLOG10 returns the base 10 logarithm of the input parameter. Synonym of the LOG function.

**Synopsis**

| DLOG10(number) |
**Argument**

*number*

The input parameter is a double precision number.

**Return type**

The DLOG10 function returns a double precision number.

**Example**

The following example returns the base 10 logarithm of the number 100:

```
select dlog10(100);
```

```plaintext
dlog10
--------
 2
(1 row)
```

**EXP function**

The EXP function returns the exponential value in scientific notation for a numeric expression.

**Synopsis**

```plaintext
EXP (expression)
```

**Argument**

*expression*

The expression must be an INTEGER, DECIMAL, or DOUBLE PRECISION data type.

**Return type**

EXP returns a DOUBLE PRECISION number.

**Example**

Use the EXP function to forecast ticket sales based on a continuous growth pattern. In this example, the subquery returns the number of tickets sold in 2008. That result is multiplied by the result of the EXP function, which specifies a continuous growth rate of 7% over 10 years.

```
select (select sum(qtysold) from sales, date
where sales.dateid=date.dateid
and year=2008) * exp((7::float/100)*10) qty2010;
```

```plaintext
qty2010
------------------
 695447.483772222
(1 row)
```
FLOOR function

The FLOOR function rounds a number down to the next whole number.

**Synopsis**

\[ \text{FLOOR} \ (\text{number}) \]

**Argument**

*number*

DOUBLE PRECISION number to be rounded down.

**Return type**

FLOOR returns an integer.

**Example**

Calculate the floor of the commission paid for a given sales transaction:

```
select floor(commission) from sales
where salesid=10000;
```

```
floor
-------
28
(1 row)
```

LN function

Returns the natural logarithm of the input parameter. Synonym of the DLOG1 function.

**Synonym of DLOG1 function (p. 504).**

**Synopsis**

\[ \text{LN} \ (\text{expression}) \]

**Argument**

*expression*

The target column or expression that the function operates on.

**Note**

This function returns an error for some data types if the expression references an Amazon Redshift user-created table or an Amazon Redshift STL or STV system table.

Expressions with the following data types produce an error if they reference a user-created or system table. Expressions with these data types run exclusively on the leader node:

- BOOLEAN
- CHAR
- DATE
• DECIMAL or NUMERIC
• TIMESTAMP
• VARCHAR

Expressions with the following data types run successfully on user-created tables and STL or STV system tables:
• BIGINT
• DOUBLE PRECISION
• INTEGER
• REAL
• SMALLINT

**Return type**

The LN function returns the same type as the expression.

**Example**

The following example returns the natural logarithm, or base e logarithm, of the number 2.718281828:

```
select ln(2.718281828);
```

```
ln
--------------------
0.9999999998311267
(1 row)
```

Note that the answer is nearly equal to 1.

This example returns the natural logarithm of the values in the USERID column in the USERS table:

```
select username, ln(userid) from users order by userid limit 10;
```

```
username |        ln
----------+-------------------
JSG99FHE |                 0
PGL08LJI | 0.693147180559945
IFT66TXU | 1.098612288866811
XDZ38RDD | 1.38629436111989
AEB55QTM | 1.6094379124341
NDQ15VBM | 1.79175946922805
OWY35QYB | 1.94591014905531
A2G78YIP | 2.07944154167984
MSD36KVR | 2.19722457733622
WKW41AIW | 2.30258509299405
(10 rows)
```

**LOG function**

Returns the base 10 logarithm of a number.

Synonym of DLOG10 function (p. 504).
Synopsis  
\texttt{LOG(number)}

Argument  
\textit{number}  
The input parameter is a double precision number.

Return type  
The \texttt{LOG} function returns a double precision number.

Example  
The following example returns the base 10 logarithm of the number 100:

\begin{verbatim}
select log(100);
does10
-----
2
(1 row)
\end{verbatim}

MOD function  
The \texttt{MOD} function returns a numeric result that is the remainder of two numeric parameters. The first parameter is divided by the second parameter.

Synopsis  
\texttt{MOD(number1, number2)}

Arguments  
\textit{number1}  
The first input parameter is an INTEGER, SMALLINT, BIGINT, or DECIMAL number. If either parameter is a DECIMAL type, the other parameter must also be a DECIMAL type. If either parameter is an INTEGER, the other parameter can be an INTEGER, SMALLINT, or BIGINT. Both parameters can also be SMALLINT orBIGINT, but one parameter cannot be a SMALLINT if the other is a BIGINT.

\textit{number2}  
The second parameter is an INTEGER, SMALLINT, BIGINT, or DECIMAL number. The same data type rules apply to \textit{number2} as to \textit{number1}.

Return type  
Valid return types are DECIMAL, INT, SMALLINT, and BIGINT. The return type of the \texttt{MOD} function is the same numeric type as the input parameters, if both input parameters are the same type. If either input parameter is an INTEGER, however, the return type will also be an INTEGER.

Example  
The following example returns information for odd-numbered categories in the \texttt{CATEGORY} table:
select catid, catname
from category
where mod(catid,2)=1
order by 1,2;

<table>
<thead>
<tr>
<th>catid</th>
<th>catname</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>MLB</td>
</tr>
<tr>
<td>3</td>
<td>NFL</td>
</tr>
<tr>
<td>5</td>
<td>MLS</td>
</tr>
<tr>
<td>7</td>
<td>Plays</td>
</tr>
<tr>
<td>9</td>
<td>Pop</td>
</tr>
<tr>
<td>11</td>
<td>Classical</td>
</tr>
</tbody>
</table>

(6 rows)

PI function

The PI function returns the value of PI to 14 decimal places.

Synopsis

PI()

Return type

PI returns a DOUBLE PRECISION number.

Examples

Return the value of pi:

select pi();

pi
------------------
3.14159265358979
(1 row)

POWER function

Synopsis

The POWER function is an exponential function that raises a numeric expression to the power of a second numeric expression. For example, 2 to the third power is calculated as power(2,3), with a result of 8.

| POW | POWER (expression1, expression2) |

POW and POWER are synonyms.

Arguments

expression1
   Numeric expression to be raised. Must be an integer, decimal, or floating-point data type.
expression2
Power to raise expression1. Must be an integer, decimal, or floating-point data type.

Return type
POWER returns a DOUBLE PRECISION number.

Examples
In the following example, the POWER function is used to forecast what ticket sales will look like in the next 10 years, based on the number of tickets sold in 2008 (the result of the subquery). The growth rate is set at 7% per year in this example.

```sql
select (select sum(qtysold) from sales, date where sales.dateid=date.dateid and year=2008) * pow((1+7::float/100),10) qty2010;
```

```
qty2010
------------------
679353.754088594
(1 row)
```

The following example is a variation on the previous example, with the growth rate at 7% per year but the interval set to months (120 months over 10 years):

```sql
select (select sum(qtysold) from sales, date where sales.dateid=date.dateid and year=2008) * pow((1+7::float/100/12),120) qty2010;
```

```
qty2010
-----------------
694034.54678046
(1 row)
```

RADIANS function
Converts an angle in degrees to its equivalent in radians.

Synopsis
```
RADIANS(number)
```

Argument

string
The input parameter is a double precision number.

Return type
The RADIUS function returns a double precision number.
Examples

The following example returns the radian equivalent of 180 degrees:

```sql
select radians(180);
radians
-------------
3.14159265358979
(1 row)
```

RANDOM function

The RANDOM function generates a random value between 0.0 and 1.0.

**Synopsis**

```
RANDOM()
```

**Return type**

RANDOM returns a DOUBLE PRECISION number.

**Usage notes**

Call RANDOM after setting a seed value with the SET (p. 389) command to cause RANDOM to generate numbers in a predictable sequence.

**Examples**

Compute a random value between 0 and 99. If the random number is 0 to 1, this query produces a random number from 0 to 100:

```sql
select cast (random() * 100 as int);
int4
-----
24
(1 row)
```

This example uses the SET (p. 389) command to set a SEED value so that RANDOM generates a predictable sequence of numbers.

First, return three RANDOM integers without setting the SEED value first:

```sql
select cast (random() * 100 as int);
int4
-----
6
(1 row)

select cast (random() * 100 as int);
int4
-----
68
(1 row)
```
select cast (random() * 100 as int);
int4
------
56
(1 row)

Now, set the SEED value to .25, and return three more RANDOM numbers:

set seed to .25;
select cast (random() * 100 as int);
int4
------
21
(1 row)
select cast (random() * 100 as int);
int4
------
79
(1 row)
select cast (random() * 100 as int);
int4
------
12
(1 row)

Finally, reset the SEED value to .25, and verify that RANDOM returns the same results as the previous three calls:

set seed to .25;
select cast (random() * 100 as int);
int4
------
21
(1 row)
select cast (random() * 100 as int);
int4
------
79
(1 row)
select cast (random() * 100 as int);
int4
------
12
(1 row)

**ROUND function**

The ROUND function rounds numbers to the nearest integer or decimal.
The ROUND function can optionally include a second argument: an integer to indicate the number of decimal places for rounding, in either direction. If the second argument is not provided, the function rounds to the nearest whole number; if the second argument \( n \) is specified, the function rounds to the nearest number with \( n \) decimal places of precision.

**Synopsis**

ROUND (number [ , integer ] )

**Argument**

*number*

INTEGER, DECIMAL, and FLOAT data types are supported.

If the first argument is an integer, the parser converts the integer into a decimal data type prior to processing. If the first argument is a decimal number, the parser processes the function without conversion, resulting in better performance.

**Return type**

ROUND returns the same numeric data type as the input argument(s).

**Examples**

Round the commission paid for a given transaction to the nearest whole number.

```
select commission, round(commission) 
from sales where salesid=10000;
```

<table>
<thead>
<tr>
<th>commission</th>
<th>round</th>
</tr>
</thead>
<tbody>
<tr>
<td>28.05</td>
<td>28</td>
</tr>
</tbody>
</table>

(1 row)

Round the commission paid for a given transaction to the first decimal place.

```
select commission, round(commission, 1) 
from sales where salesid=10000;
```

<table>
<thead>
<tr>
<th>commission</th>
<th>round</th>
</tr>
</thead>
<tbody>
<tr>
<td>28.05</td>
<td>28.1</td>
</tr>
</tbody>
</table>

(1 row)

For the same query, extend the precision in the opposite direction.

```
select commission, round(commission, -1) 
from sales where salesid=10000;
```

<table>
<thead>
<tr>
<th>commission</th>
<th>round</th>
</tr>
</thead>
<tbody>
<tr>
<td>28.05</td>
<td>30</td>
</tr>
</tbody>
</table>

(1 row)
**SIN function**

SIN is a trigonometric function that returns the sine of a number. The return value is in radians and is between PI/2 and -PI/2.

**Synopsis**

```
SIN(number)
```

**Argument**

*number*

The input parameter is a double precision number.

**Return type**

The SIN function returns a double precision number.

**Examples**

The following example returns the sine of PI:

```
select sin(-pi());
```

```
<table>
<thead>
<tr>
<th>sin</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1.22464679914735e-16</td>
</tr>
</tbody>
</table>
```

(1 row)

---

**SIGN function**

The SIGN function returns the sign (positive or negative) of a numeric value. The result of the SIGN function will either be a 1 or a -1, indicating the sign of the argument.

**Synopsis**

```
SIGN (numeric)
```

**Argument**

*numeric*

Numeric value to be evaluated.

**Return type**

The SIGN function returns an integer.

**Examples**

Determine the sign of the commission paid for a given transaction:
select commission, sign (commission)
from sales where salesid=10000;

<table>
<thead>
<tr>
<th>commission</th>
<th>sign</th>
</tr>
</thead>
<tbody>
<tr>
<td>28.05</td>
<td>1</td>
</tr>
</tbody>
</table>

(1 row)

**SQRT function**

The SQRT function returns the square root of a numeric value.

**Synopsis**

SQRT (expression)

**Argument**

expression

The expression must have an integer, decimal, or floating-point data type.

**Return type**

SQRT returns a DOUBLE PRECISION number.

**Examples**

The following example returns the square root for some COMMISSION values from the SALES table. The COMMISSION column is a DECIMAL column.

```sql
select sqrt(commission)
from sales where salesid <10 order by salesid;
```

<table>
<thead>
<tr>
<th>sqrt</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.4498803820905</td>
</tr>
<tr>
<td>3.37638860322683</td>
</tr>
<tr>
<td>7.24568837309472</td>
</tr>
<tr>
<td>5.1234753829798</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

The following query returns the rounded square root for the same set of COMMISSION values.

```sql
select salesid, commission, round(sqrt(commission))
from sales where salesid <10 order by salesid;
```

<table>
<thead>
<tr>
<th>salesid</th>
<th>commission</th>
<th>round</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>109.20</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>11.40</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>52.50</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>26.25</td>
<td>5</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
TAN function

TAN is a trigonometric function that returns the tangent of a number. The input parameter must be a non-zero number (in radians).

Synopsis

TAN(number)

Argument

number

The input parameter is a double precision number.

Return type

The TAN function returns a double precision number.

Examples

The following example returns the tangent of 0:

```
select tan(0);
```

<table>
<thead>
<tr>
<th>tan</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

(1 row)

TRUNC function

The TRUNC function truncates a number and right-fills it with zeros from the position specified. This function also truncates a time stamp and returns a date.

Synopsis

TRUNC(number [ , integer ] | timestamp)

Arguments

number

Numeric data type to be truncated. SMALLINT, INTEGER, BIGINT, DECIMAL, REAL, and DOUBLE PRECISION data types are supported.

integer (optional)

An integer that indicates the number of decimal places of precision, in either direction. If no integer is provided, the number is truncated as a whole number; if an integer is specified, the number is truncated to the specified decimal place.

timestamp

The function can also return the date from a time stamp. (To return a time stamp value with 00:00:00 as the time, cast the function result to a time stamp.)
Return type

TRUNC returns the same numeric data type as the first input argument. For time stamps, TRUNC returns a date.

Examples

Truncate the commission paid for a given sales transaction.

```sql
select commission, trunc(commission)
from sales where salesid=784;
```

```
commission | trunc
------------+-------
111.15      |   111
```

(1 row)

Truncate the same commission value to the first decimal place.

```sql
select commission, trunc(commission,1)
from sales where salesid=784;
```

```
commission | trunc
------------+-------
111.15      | 111.1
```

(1 row)

Truncate the commission with a negative value for the second argument; 111.15 is rounded down to 110.

```sql
select commission, trunc(commission,-1)
from sales where salesid=784;
```

```
commission | trunc
------------+-------
111.15      |   110
```

(1 row)

Return the date portion from the result of the SYSDATE function (which returns a time stamp):

```sql
select sysdate;
```

```
timestamp
----------------------------
2011-07-21 10:32:38.248109
```

(1 row)

```sql
select trunc(sysdate);
```

```
trunc
-------
2011-07-21
```

(1 row)
Apply the TRUNC function to a TIMESTAMP column. The return type is a date.

```
select trunc(starttime) from event
order by eventid limit 1;
```

<table>
<thead>
<tr>
<th>trunc</th>
</tr>
</thead>
<tbody>
<tr>
<td>2008-01-25</td>
</tr>
</tbody>
</table>

(String functions)

Topics
- || (Concatenation) operator (p. 519)
- ASCII function (p. 520)
- BPCHARCMP function (p. 520)
- BTRIM function (p. 521)
- BTTEXT_PATTERN_CMP function (p. 522)
- CHAR_LENGTH function (p. 522)
- CHARACTER_LENGTH function (p. 523)
- CHARINDEX function (p. 523)
- CHR function (p. 524)
- CONCAT (Oracle compatibility function) (p. 524)
- CRC32 function (p. 526)
- FUNC_SHA1 function (p. 527)
- GET_BIT function (p. 527)
- GET_BYTE function (p. 527)
- INITCAP function (p. 527)
- LEFT and RIGHT functions (p. 529)
- LEN function (p. 530)
- LENGTH function (p. 531)
- LOWER function (p. 531)
- LPAD and RPAD functions (p. 532)
- LTRIM function (p. 533)
- MD5 function (p. 534)
- OCTET_LENGTH function (p. 534)
- POSITION function (p. 534)
- QUOTE_IDENT function (p. 535)
- QUOTE_LITERAL function (p. 536)
- REGEXP_COUNT function (p. 537)
- REGEXP_INSTR function (p. 538)
- REGEXP_REPLACE function (p. 539)
- REGEXP_SUBSTR function (p. 540)
- REPEAT function (p. 541)
- REPLACE function (p. 542)
- REPLICATE function (p. 543)
- REVERSE function (p. 543)
String functions process and manipulate character strings or expressions that evaluate to character strings. When the *string* argument in these functions is a literal value, it must be enclosed in single quotes. Supported data types include CHAR and VARCHAR.

The following section provides the function names, syntax, and descriptions for supported functions. All offsets into strings are 1-based.

The following leader-node only string functions are deprecated.

- ASCII
- GET_BIT
- GET_BYTE
- OCTET_LENGTH
- SET_BIT
- SET_BYTE
- TO_ASCII

|| (Concatenation) operator

Concatenates two strings on either side of the || symbol and returns the concatenated string.

Similar to CONCAT (Oracle compatibility function) (p. 524).

**Note**

For both the CONCAT function and the concatenation operator, if one or both strings is null, the result of the concatenation is null.

**Synopsis**

```
string1 || string2
```

**Arguments**

`string1, string2`

Both arguments can be fixed-length or variable-length character strings or expressions.
### Return type

The `||` operator returns a string. The type of string is the same as the input arguments.

### Example

The following example concatenates the `FIRSTNAME` and `LASTNAME` fields from the `USERS` table:

```sql
select firstname || ' ' || lastname
from users
order by 1
limit 10;
```

<table>
<thead>
<tr>
<th>column?</th>
</tr>
</thead>
<tbody>
<tr>
<td>--------</td>
</tr>
<tr>
<td>Aaron Banks</td>
</tr>
<tr>
<td>Aaron Booth</td>
</tr>
<tr>
<td>Aaron Browning</td>
</tr>
<tr>
<td>Aaron Burnett</td>
</tr>
<tr>
<td>Aaron Casey</td>
</tr>
<tr>
<td>Aaron Cash</td>
</tr>
<tr>
<td>Aaron Castro</td>
</tr>
<tr>
<td>Aaron Dickerson</td>
</tr>
<tr>
<td>Aaron Dickerson</td>
</tr>
<tr>
<td>Aaron Dotson</td>
</tr>
<tr>
<td>(10 rows)</td>
</tr>
</tbody>
</table>

### ASCII function

ASCII is a deprecated leader-node-only function.

### BPCHARCMP function

Compares the value of two strings and returns an integer. If the strings are identical, returns 0. If the first string is “greater” alphabetically, returns 1. If the second string is “greater”, returns -1.

For multi-byte characters, the comparison is based on the byte encoding.

**Synonym** of `BTTEXT_PATTERN_CMP` function (p. 522).

### Synopsis

```
BPCHARCMP(string1, string2)
```

### Arguments

- `string1`
  The first input parameter is a CHAR or VARCHAR string.
- `string2`
  The second parameter is a CHAR or VARCHAR string.

### Return type

The `BPCHARCMP` function returns an integer.
### Examples

The following example determines whether a user’s first name is alphabetically greater than the user’s last name for the first ten entries in USERS:

```sql
select userid, firstname, lastname,
bpcharcmp(firstname, lastname)
from users
order by 1, 2, 3, 4
limit 10;
```

This example returns the following sample output:

<table>
<thead>
<tr>
<th>userid</th>
<th>firstname</th>
<th>lastname</th>
<th>bpcharcmp</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Rafael</td>
<td>Taylor</td>
<td>-1</td>
</tr>
<tr>
<td>2</td>
<td>Vladimir</td>
<td>Humphrey</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>Lars</td>
<td>Ratliff</td>
<td>-1</td>
</tr>
<tr>
<td>4</td>
<td>Barry</td>
<td>Roy</td>
<td>-1</td>
</tr>
<tr>
<td>5</td>
<td>Reagan</td>
<td>Hodge</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>Victor</td>
<td>Hernandez</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>Tamekah</td>
<td>Juarez</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>Colton</td>
<td>Roy</td>
<td>-1</td>
</tr>
<tr>
<td>9</td>
<td>Mufutau</td>
<td>Watkins</td>
<td>-1</td>
</tr>
<tr>
<td>10</td>
<td>Naida</td>
<td>Calderon</td>
<td>1</td>
</tr>
</tbody>
</table>

(10 rows)

You can see that for entries where the string for the FIRSTNAME is later alphabetically than the LASTNAME, BPCHARCMP returns 1. If the LASTNAME is alphabetically later than FIRSTNAME, BPCHARCMP returns -1.

This example returns all entries in the USER table whose FIRSTNAME is identical to their LASTNAME:

```sql
select userid, firstname, lastname,
bpcharcmp(firstname, lastname)
from users where bpcharcmp(firstname, lastname)=0
order by 1, 2, 3, 4;
```

<table>
<thead>
<tr>
<th>userid</th>
<th>firstname</th>
<th>lastname</th>
<th>bpcharcmp</th>
</tr>
</thead>
<tbody>
<tr>
<td>62</td>
<td>Chase</td>
<td>Chase</td>
<td>0</td>
</tr>
<tr>
<td>4008</td>
<td>Whitney</td>
<td>Whitney</td>
<td>0</td>
</tr>
<tr>
<td>12516</td>
<td>Graham</td>
<td>Graham</td>
<td>0</td>
</tr>
<tr>
<td>13570</td>
<td>Harper</td>
<td>Harper</td>
<td>0</td>
</tr>
<tr>
<td>16712</td>
<td>Cooper</td>
<td>Cooper</td>
<td>0</td>
</tr>
<tr>
<td>18359</td>
<td>Chase</td>
<td>Chase</td>
<td>0</td>
</tr>
<tr>
<td>27530</td>
<td>Bradley</td>
<td>Bradley</td>
<td>0</td>
</tr>
<tr>
<td>31204</td>
<td>Harding</td>
<td>Harding</td>
<td>0</td>
</tr>
</tbody>
</table>

(8 rows)

### BTRIM function

The BTRIM function trims a string by removing leading and trailing blanks or by removing characters that match an optional specified string.
Synopsis

BTRIM(string [, matching_string ])

Arguments

string
The first input parameter is a VARCHAR string.

matching_string
The second parameter, if present, is a VARCHAR string.

Return type

The BTRIM function returns a VARCHAR string.

Examples

The following example trims leading and trailing blanks from the string ' abc ':

```sql
select '     abc    ' as untrim, btrim('     abc    ') as trim;
```

<table>
<thead>
<tr>
<th>untrim</th>
<th>trim</th>
</tr>
</thead>
<tbody>
<tr>
<td>abc</td>
<td>abc</td>
</tr>
</tbody>
</table>

The following example removes the leading and trailing 'xyz' strings from the string 'xyzaxyzbxyzcxyz'

```sql
select 'xyzaxyzbxyzcxyz' as untrim,
btrim('xyzaxyzbxyzcxyz', 'xyz') as trim;
```

<table>
<thead>
<tr>
<th>untrim</th>
<th>trim</th>
</tr>
</thead>
<tbody>
<tr>
<td>xyzaxyzbxyzcxyz</td>
<td>axyzbxyzc</td>
</tr>
</tbody>
</table>

Note that the leading and trailing occurrences of 'xyz' were removed, but that occurrences that were internal within the string were not removed.

BTTEXT_PATTERN_CMP function

Synonym for the BPCHARCMP function.

See BPCHARCMP function (p. 520) for details.

CHAR_LENGTH function

Synonym of the LEN function.

See LEN function (p. 530)
CHARACTER_LENGTH function

Synonym of the LEN function.

See LEN function (p. 530)

CHARINDEX function

Returns the location of the specified substring within a string. Synonym of the STRPOS function.

Synopsis

CHARINDEX(substring, string)

Arguments

substring
    The substring to search for within the string.
string
    The string or column to be searched.

Return type

The CHARINDEX function returns an integer corresponding to the position of the substring (one-based, not zerobased). The position is based on the number of characters, not bytes, so that multi-byte characters are counted as single characters.

Usage notes

CHARINDEX returns 0 if the substring is not found within the string:

select charindex('dog', 'fish');
charindex
--------
0
(1 row)

Examples

The following example shows the position of the string fish within the word dogfish:

select charindex('fish', 'dogfish');
charindex
--------
 4
(1 row)

The following example returns the number of sales transactions with a COMMISSION over 999.00 from the SALES table:
select distinct charindex('.', commission), count (charindex('.', commission))
from sales
where charindex('.', commission) > 4
group by charindex('.', commission)
order by 1,2;

<table>
<thead>
<tr>
<th>charindex</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>629</td>
</tr>
</tbody>
</table>

(1 row)

See STRPOS function (p. 546) for details.

**CHR function**

The CHR function returns the character that matches the ASCII code point value specified by the input parameter.

**Synopsis**

CHR(number)

**Argument**

number

The input parameter is an integer that represents an ASCII code point value.

**Return type**

The CHR function returns a CHAR string if an ASCII character matches the input value. If the input number has no ASCII match, the function returns null.

**Example**

The following example returns event names that begin with a capital A (ASCII code point 65):

select distinct eventname from event
where substring(eventname, 1, 1)=chr(65);

<table>
<thead>
<tr>
<th>eventname</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adriana Lecouvreur</td>
</tr>
<tr>
<td>A Man For All Seasons</td>
</tr>
<tr>
<td>A Bronx Tale</td>
</tr>
<tr>
<td>A Christmas Carol</td>
</tr>
<tr>
<td>Allman Brothers Band</td>
</tr>
</tbody>
</table>

**CONCAT (Oracle compatibility function)**

The CONCAT function concatenates two character strings and returns the resulting string. To concatenate more than two strings, use nested CONCAT functions. The concatenation operator (||) between two strings produces the same results as the CONCAT function.
Note
For both the CONCAT function and the concatenation operator, if one or both strings is null, the result of the concatenation is null.

Synopsis

CONCAT ( string1, string2 )

Arguments

string1, string2
Both arguments can be fixed-length or variable-length character strings or expressions.

Return type

CONCAT returns a string. The data type of the string is the same type as the input arguments.

Examples

The following example concatenates two character literals:

```
select concat('December 25, ', '2008');
```

<table>
<thead>
<tr>
<th>concat</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>December 25, 2008</td>
<td>(1 row)</td>
</tr>
</tbody>
</table>

The following query, using the || operator instead of CONCAT, produces the same result:

```
select 'December 25, '||'2008';
```

<table>
<thead>
<tr>
<th>?column?</th>
<th>---------------</th>
</tr>
</thead>
<tbody>
<tr>
<td>December 25, 2008</td>
<td>(1 row)</td>
</tr>
</tbody>
</table>

The following example uses two CONCAT functions to concatenate three character strings:

```
select concat('Thursday, ', concat('December 25, ', '2008'));
```

<table>
<thead>
<tr>
<th>concat</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Thursday, December 25, 2008</td>
<td>(1 row)</td>
</tr>
</tbody>
</table>

The following query concatenates CITY and STATE values from the VENUE table:

```
select concat(venuecity, venuestate) from venue where venueseats > 75000 order by venueseats;
```
The following query uses nested CONCAT functions. The query concatenates CITY and STATE values from the VENUE table but delimits the resulting string with a comma and a space:

```sql
select concat(concat(venuecity,', '),venuestate)
from venue
where venueseats > 75000
order by venueseats;
```

```plaintext
(4 rows)
```

**CRC32 function**

CRC32 is an error-detecting function that uses a CRC32 algorithm to detect changes between source and target data. The CRC32 function converts a variable-length string into an 8-character string that is a text representation of the hexadecimal value of a 32 bit-binary sequence.

**Syntax**

```sql
CRC32(string)
```

**Arguments**

`string`

A variable-length string.

**Return type**

The CRC32 function returns an 8-character string that is a text representation of the hexadecimal value of a 32-bit binary sequence. The Amazon Redshift CRC32 function is based on the CRC-32C polynomial.

**Example**

The following example shows the 32-bit value for the string 'Amazon Redshift':

```sql
select crc32('Amazon Redshift');
crc32
---------------------
f2726906
(1 row)
```
**FUNC_SHA1 function**

The FUNC_SHA1 function uses the SHA1 cryptographic hash function to convert a variable-length string into a 40-character string that is a text representation of the hexadecimal value of a 160-bit checksum.

**Syntax**

```
FUNC_SHA1(string)
```

**Arguments**

- `string`: A variable-length string.

**Return type**

The FUNC_SHA1 function returns a 40-character string that is a text representation of the hexadecimal value of a 160-bit checksum.

**Example**

The following example returns the 160-bit value for the word 'Amazon Redshift':

```
select func_sha1('Amazon Redshift');
```

---

**GET_BIT function**

GET_BIT is a deprecated leader-node-only function.

**GET_BYTE function**

GET_BYTE is a deprecated leader-node-only function.

---

**INITCAP function**

Capitalize the first letter of each word in a specified string. INITCAP supports UTF-8 multibyte characters, up to a maximum of four bytes per character.

**Synopsis**

```
INITCAP(string)
```

**Argument**

- `string`: The input parameter is a CHAR or VARCHAR string.

**Return type**

The INITCAP function returns a VARCHAR string.
Usage notes

The INITCAP function makes the first letter of each word in a string uppercase, and any subsequent letters are made (or left) lowercase. Therefore, it is important to understand which characters (other than space characters) function as word separators. A word separator character is any non-alphanumeric character, including punctuation marks, symbols, and control characters. All of the following characters are word separators:

```
! " # $ % & ' ( ) * + , - . / : ; < = > ? @ [ \ ] ^ _ ` { | } ~
```

Tabs, newline characters, form feeds, line feeds, and carriage returns are also word separators.

Examples

The following example capitalizes the initials of each word in the CATDESC column:

```sql
select catid, catdesc, initcap(catdesc)
from category
order by 1, 2, 3;
```

```
catid | catdesc                          | initcap
-------+---------------------------------+-------------------------
1       | Major League Baseball            | Major League Baseball   
2       | National Hockey League           | National Hockey League  
3       | National Football League         | National Football League
4       | National Basketball Association  | National Basketball Association
5       | Major League Soccer              | Major League Soccer     
6       | Musical theatre                  | Musical Theatre         
7       | All non-musical theatre          | All Non-Musical Theatre 
8       | All opera and light opera        | All Opera And Light Opera
9       | All rock and pop music concerts  | All Rock And Pop Music Concerts
10      | All jazz singers and bands       | All Jazz Singers And Bands
11      | All symphony, concerto, and choir concerts | All Symphony, Concerto, And Choir Concerts
```

(11 rows)

The following example shows that the INITCAP function does not preserve uppercase characters when they do not begin words. For example, MLB becomes Mlb.

```sql
select initcap(catname)
from category
order by catname;
```

```
initcap
-----------
Classical
Jazz
MLb
Mls
Musicals
Nba
Nfl
Nhl
Opera
Plays
```
The following example shows that non-alphanumeric characters other than spaces function as word separators, causing uppercase characters to be applied to several letters in each string:

```sql
select email, initcap(email)
from users
order by userid desc limit 5;
```

<table>
<thead>
<tr>
<th>email</th>
<th>initcap</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="mailto:urna.Ut@egetdictumplacerat.edu">urna.Ut@egetdictumplacerat.edu</a></td>
<td><a href="mailto:Urna.Ut@Egetdictumplacerat.Edu">Urna.Ut@Egetdictumplacerat.Edu</a></td>
</tr>
<tr>
<td><a href="mailto:nibh.enim@egestas.ca">nibh.enim@egestas.ca</a></td>
<td><a href="mailto:Nibh.Enim@Egestas.Ca">Nibh.Enim@Egestas.Ca</a></td>
</tr>
<tr>
<td><a href="mailto:in@Donecat.ca">in@Donecat.ca</a></td>
<td><a href="mailto:In@Donecat.Ca">In@Donecat.Ca</a></td>
</tr>
<tr>
<td><a href="mailto:sodales@blanditviverradonec.ca">sodales@blanditviverradonec.ca</a></td>
<td><a href="mailto:Sodales@Blanditviverradonec.Ca">Sodales@Blanditviverradonec.Ca</a></td>
</tr>
<tr>
<td><a href="mailto:sociis.natoque.penatibus@vitae.org">sociis.natoque.penatibus@vitae.org</a></td>
<td><a href="mailto:Sociis.Natoque.Penatibus@Vitae.Org">Sociis.Natoque.Penatibus@Vitae.Org</a></td>
</tr>
</tbody>
</table>

(5 rows)

**LEFT and RIGHT functions**

These functions return the specified number of leftmost or rightmost characters from a character string. The number is based on the number of characters, not bytes, so that multi-byte characters are counted as single characters.

**Synopsis**

```sql
LEFT ( string, integer )
RIGHT ( string, integer )
```

**Arguments**

- `string` Any character string or any expression that evaluates to a character string.
- `integer` A positive integer.

**Return type**

LEFT and RIGHT return a VARCHAR string.

**Example**

The following example returns the leftmost 5 and rightmost 5 characters from event names that have IDs between 1000 and 1005:

```sql
select eventid, eventname,
left(eventname,5) as left_5,
right(eventname,5) as right_5
from event
where eventid between 1000 and 1005
```
order by 1;

<table>
<thead>
<tr>
<th>eventid</th>
<th>eventname</th>
<th>left_5</th>
<th>right_5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>Gypsy</td>
<td>Gypsy</td>
<td>Gypsy</td>
</tr>
<tr>
<td>1001</td>
<td>Chicago</td>
<td>Chica</td>
<td>icago</td>
</tr>
<tr>
<td>1002</td>
<td>The King and I</td>
<td>The K</td>
<td>and I</td>
</tr>
<tr>
<td>1003</td>
<td>Pal Joey</td>
<td>Pal J</td>
<td>Joey</td>
</tr>
<tr>
<td>1004</td>
<td>Grease</td>
<td>Greas</td>
<td>rease</td>
</tr>
<tr>
<td>1005</td>
<td>Chicago</td>
<td>Chica</td>
<td>icago</td>
</tr>
</tbody>
</table>

(6 rows)

**LEN function**

Returns the length of the specified string.

**Synopsis**

LEN is a synonym of LENGTH function (p.531), CHAR_LENGTH function (p.522), CHARACTER_LENGTH function (p.523), and TEXTLEN function (p.551).

LEN(expression)

**Argument**

expression

The input parameter is a CHAR or VARCHAR text string.

**Return type**

The LEN function returns an integer indicating the number of characters in the input string. The LEN function returns the actual number of characters in multi-byte strings, not the number of bytes. For example, a VARCHAR(12) column is required to store three four-byte Chinese characters. The LEN function will return 4 for that same string.

**Usage notes**

Length calculations do not count trailing spaces for fixed-length character strings but do count them for variable-length strings.

**Example**

The following example returns the number of characters in the strings `cat` with no trailing spaces and `cat` with three trailing spaces:

```sql
select len('cat'), len('cat   ');
```

<table>
<thead>
<tr>
<th>len</th>
<th>len</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>6</td>
</tr>
</tbody>
</table>

(1 row)

The following example returns the ten longest VENUENAME entries in the VENUE table:
select venuename, len(venuename)
from venue
order by 2 desc, 1
limit 10;

venuename                | len
-----------------------------------------+-----
Saratoga Springs Performing Arts Center |  39
Lincoln Center for the Performing Arts  |  38
Nassau Veterans Memorial Coliseum       |  33
Jacksonville Municipal Stadium          |  30
Rangers BallPark in Arlington           |  29
University of Phoenix Stadium           |  29
Circle in the Square Theatre            |  28
Hubert H. Humphrey Metrodome            |  28
Oriole Park at Camden Yards             |  27
Dick’s Sporting Goods Park              |  26
(10 rows)

**LENGTH function**

Synonym of the LEN function.

See LEN function (p. 530)

**LOWER function**

Converts a string to lower case. LOWER supports UTF-8 multibyte characters, up to a maximum of four bytes per character.

**Synopsis**

```
LOWER(string)
```

**Argument**

`string`

The input parameter is a CHAR or VARCHAR string.

**Return type**

The LOWER function returns a character string that is the same data type as the input string (CHAR or VARCHAR).

**Examples**

The following example converts the CATNAME field to lower case:

```
select catname, lower(catname) from category order by 1,2;
```

<table>
<thead>
<tr>
<th>catname</th>
<th>lower</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classical</td>
<td>classical</td>
</tr>
<tr>
<td>Jazz</td>
<td>jazz</td>
</tr>
</tbody>
</table>
LPAD and RPAD functions

These functions prepend or append characters to a string, based on a specified length.

Synopsis

LPAD (string1, length, [ string2 ])

RPAD (string1, length, [ string2 ])

Arguments

string1
A character string or an expression that evaluates to a character string, such as the name of a character column.

length
An integer that defines the length of the result of the function. The length of a string is based on the number of characters, not bytes, so that multi-byte characters are counted as single characters. If string1 is longer than the specified length, it is truncated (on the right). If length is a negative number, the result of the function is an empty string.

string2
One or more characters that are prepended or appended to string1. This argument is optional; if it is not specified, spaces are used.

Return type

These functions return a VARCHAR data type.

Examples

Truncate a specified set of event names to 20 characters and prepend the shorter names with spaces:

```
select lpad(eventname,20) from event
where eventid between 1 and 5 order by 1;
```

```
lpad
----------------------
Salome
Il Trovatore
Boris Godunov
Gotterdammerung
```
Truncate the same set of event names to 20 characters but append the shorter names with 0123456789.

```
select rpad(eventname,20,'0123456789') from event
where eventid between 1 and 5 order by 1;
```

```
rpad
----------------------
Boris Godunov0123456
Gotterdammerung01234
Il Trovatore01234567
La Cenerentola (Cind
Salome01234567890123
(5 rows)
```

**LTRIM function**

The LTRIM function trims a specified set of characters from the beginning of a string.

**Synopsis**

```
LTRIM( string, 'characters' )
```

**Arguments**

- `characters` 
  The first input parameter is a CHAR or VARCHAR string representing the characters to be trimmed from the beginning of the string.

- `string` 
  The second parameter is the CHAR or VARCHAR string to be trimmed.

**Return type**

The LTRIM function returns a character string that is the same data type as the input string (CHAR or VARCHAR).

**Example**

The following example trims the year from LISTTIME:

```
select listid, listtime, ltrim(listtime, '2008-')
from listing
order by 1, 2, 3
limit 10;
```

```
listid |      listtime       |     ltrim
--------+---------------------+----------------
1 | 2008-01-24 06:43:29 | 1-24 06:43:29
3 | 2008-11-01 07:35:33 | 11-01 07:35:33
```
MD5 function

Uses the MD5 cryptographic hash function to convert a variable-length string into a 32-character string that is a text representation of the hexadecimal value of a 128-bit checksum.

Syntax

```
MD5(string)
```

Arguments

```
string
```
A variable-length string.

Return type

The MD5 function returns a 32-character string that is a text representation of the hexadecimal value of a 128-bit checksum.

Examples

The following example shows the 128-bit value for the string 'Amazon Redshift':

```
select md5('Amazon Redshift');
md5
----------------------------------
f7415e33f972c03abd4f3fed36748f7a
```

OCTET_LENGTH function

OCTET_LENGTH is a deprecated leader-node-only function.

POSITION function

Returns the location of the specified substring within a string.

Synonym of the STRPOS function (p. 546) function.
**Arguments**

*substring*
   - The substring to search for within the *string*.

*string*
   - The string or column to be searched.

**Return type**

The POSITION function returns an integer corresponding to the position of the substring (one-based, not zero-based). The position is based on the number of characters, not bytes, so that multi-byte characters are counted as single characters.

**Usage notes**

POSITION returns 0 if the substring is not found within the string:

```sql
select position('dog' in 'fish');
```

<table>
<thead>
<tr>
<th>position</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

(1 row)

**Examples**

The following example shows the position of the string *fish* within the word *dogfish*:

```sql
select position('fish' in 'dogfish');
```

<table>
<thead>
<tr>
<th>position</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
</tr>
</tbody>
</table>

(1 row)

The following example returns the number of sales transactions with a COMMISSION over 999.00 from the SALES table:

```sql
select distinct position('.' in commission), count (position('.' in commission))
from sales where position('.' in commission) > 4 group by position('.' in commission)
order by 1,2;
```

<table>
<thead>
<tr>
<th>position</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>629</td>
</tr>
</tbody>
</table>

(1 row)

**QUOTE_IDENT function**

The QUOTE_IDENT function returns the specified string as a double quoted string so that it can be used as an identifier in a SQL statement. Appropriately doubles any embedded double quotes.
**Synopsis**

QUOTE_IDENT(string)

**Argument**

string

The input parameter can be a CHAR or VARCHAR string.

**Return type**

The QUOTE_IDENT function returns the same type string as the input parameter.

**Example**

The following example returns the CATNAME column surrounded by quotes:

```sql
select catid, quote_ident(catname)
from category
order by 1,2;
```

<table>
<thead>
<tr>
<th>catid</th>
<th>quote_ident</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>&quot;MLB&quot;</td>
</tr>
<tr>
<td>2</td>
<td>&quot;NHL&quot;</td>
</tr>
<tr>
<td>3</td>
<td>&quot;NFL&quot;</td>
</tr>
<tr>
<td>4</td>
<td>&quot;NBA&quot;</td>
</tr>
<tr>
<td>5</td>
<td>&quot;MLS&quot;</td>
</tr>
<tr>
<td>6</td>
<td>&quot;Musicals&quot;</td>
</tr>
<tr>
<td>7</td>
<td>&quot;Plays&quot;</td>
</tr>
<tr>
<td>8</td>
<td>&quot;Opera&quot;</td>
</tr>
<tr>
<td>9</td>
<td>&quot;Pop&quot;</td>
</tr>
<tr>
<td>10</td>
<td>&quot;Jazz&quot;</td>
</tr>
<tr>
<td>11</td>
<td>&quot;Classical&quot;</td>
</tr>
</tbody>
</table>

(11 rows)

**QUOTE_LITERAL function**

The QUOTE_LITERAL function returns the specified string as a quoted string so that it can be used as a string literal in a SQL statement. If the input parameter is a number, QUOTE_LITERAL treats it as a string. Appropriately doubles any embedded single quotes and backslashes.

**Synopsis**

QUOTE_LITERAL(string)

**Argument**

string

The input parameter is a CHAR or VARCHAR string.
**Return type**

The QUOTE_LITERAL function returns a string that is the same data type as the input string (CHAR or VARCHAR).

**Example**

The following example returns the CATID column surrounded by quotes. Note that the ordering now treats this column as a string:

```sql
select quote_literal(catid), catname
from category
order by 1,2;
```

<table>
<thead>
<tr>
<th>quote_literal</th>
<th>catname</th>
</tr>
</thead>
<tbody>
<tr>
<td>'1'</td>
<td>MLB</td>
</tr>
<tr>
<td>'10'</td>
<td>Jazz</td>
</tr>
<tr>
<td>'11'</td>
<td>Classical</td>
</tr>
<tr>
<td>'2'</td>
<td>NHL</td>
</tr>
<tr>
<td>'3'</td>
<td>NFL</td>
</tr>
<tr>
<td>'4'</td>
<td>NBA</td>
</tr>
<tr>
<td>'5'</td>
<td>MLS</td>
</tr>
<tr>
<td>'6'</td>
<td>Musicals</td>
</tr>
<tr>
<td>'7'</td>
<td>Plays</td>
</tr>
<tr>
<td>'8'</td>
<td>Opera</td>
</tr>
<tr>
<td>'9'</td>
<td>Pop</td>
</tr>
</tbody>
</table>

(11 rows)

**REGEXP_COUNT function**

Searches a string for a regular expression pattern and returns an integer that indicates the number of times the pattern occurs in the string. If no match is found, then the function returns 0. For more information about regular expressions, see POSIX operators (p. 249).

**Synopsis**

```sql
REGEXP_COUNT ( source_string, pattern [, position ] )
```

**Arguments**

- **source_string**
  A string expression, such as a column name, to be searched.

- **pattern**
  A string literal that represents a SQL standard regular expression pattern.

- **position**
  A positive integer that indicates the position within source_string to begin searching. The position is based on the number of characters, not bytes, so that multibyte characters are counted as single characters. The default is 1. If position is less than 1, the search begins at the first character of source_string. If position is greater than the number of characters in source_string, the result is 0.

**Return type**

Integer
Example

The following example counts the number of times a three-letter sequence occurs.

```sql
select regexp_count('abcdefghijklmnopqrstuvwxyz', '[a-z]{3}');
```

<table>
<thead>
<tr>
<th>regexp_count</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
</tr>
<tr>
<td>(1 row)</td>
</tr>
</tbody>
</table>

The following example counts the number of times the top-level domain name is either `org` or `edu`.

```sql
select email, regexp_count(email, '@[^.]*\.(org|edu)')
from users limit 5;
```

<table>
<thead>
<tr>
<th>email</th>
<th>regexp_count</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="mailto:elementum@semperpretiumneque.ca">elementum@semperpretiumneque.ca</a></td>
<td>0</td>
</tr>
<tr>
<td><a href="mailto:Integer.mollis.Integer@tristiquealiquet.org">Integer.mollis.Integer@tristiquealiquet.org</a></td>
<td>1</td>
</tr>
<tr>
<td><a href="mailto:lorem.ipsum@Vestibulumante.com">lorem.ipsum@Vestibulumante.com</a></td>
<td>0</td>
</tr>
<tr>
<td><a href="mailto:euismod@turpis.org">euismod@turpis.org</a></td>
<td>1</td>
</tr>
<tr>
<td><a href="mailto:non.justo.Proin@ametconsectetuer.edu">non.justo.Proin@ametconsectetuer.edu</a></td>
<td>1</td>
</tr>
</tbody>
</table>

**REGEXP_INSTR function**

Searches a string for a regular expression pattern and returns an integer that indicates the beginning position of the matched substring. If no match is found, then the function returns 0. **REGEXP_INSTR** is similar to the **POSITION** function (p. 534), but lets you search a string for a regular expression pattern. For more information about regular expressions, see **POSIX operators** (p. 249).

**Synopsis**

```sql
REGEXP_INSTR ( source_string, pattern [, position ] )
```

**Arguments**

- **source_string**
  A string expression, such as a column name, to be searched.
- **pattern**
  A string literal that represents a SQL standard regular expression pattern.
- **position**
  A positive integer that indicates the position within `source_string` to begin searching. The position is based on the number of characters, not bytes, so that multibyte characters are counted as single characters. The default is 1. If `position` is less than 1, the search begins at the first character of `source_string`. If `position` is greater than the number of characters in `source_string`, the result is 0.

**Return type**

Integer
Example

The following example searches for the @ character that begins a domain name and returns the starting position of the first match.

```sql
select email, regexp_count(email,'@[^\.]\.(org|edu)')
from users limit 5;
```

<table>
<thead>
<tr>
<th>email</th>
<th>regexp_count</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="mailto:elementum@semperpretiumneque.ca">elementum@semperpretiumneque.ca</a></td>
<td>10</td>
</tr>
<tr>
<td><a href="mailto:Integer.mollis.Integer@tristiquealiquet.org">Integer.mollis.Integer@tristiquealiquet.org</a></td>
<td>23</td>
</tr>
<tr>
<td><a href="mailto:lorem.ipsum@Vestibulumante.com">lorem.ipsum@Vestibulumante.com</a></td>
<td>12</td>
</tr>
<tr>
<td><a href="mailto:euismod@turpis.org">euismod@turpis.org</a></td>
<td>8</td>
</tr>
<tr>
<td><a href="mailto:non.justo.Proin@ametconsectetuer.edu">non.justo.Proin@ametconsectetuer.edu</a></td>
<td>16</td>
</tr>
</tbody>
</table>

The following example searches for variants of the word Center and returns the starting position of the first match.

```sql
select venuename, regexp_instr(venuename,'^[cC]ent(er|re)$') from venue limit 5;
```

<table>
<thead>
<tr>
<th>venuename</th>
<th>regexp_instr</th>
</tr>
</thead>
<tbody>
<tr>
<td>RFK Stadium</td>
<td>0</td>
</tr>
<tr>
<td>Air Canada Centre</td>
<td>12</td>
</tr>
<tr>
<td>United Center</td>
<td>8</td>
</tr>
<tr>
<td>Toyota Center</td>
<td>8</td>
</tr>
<tr>
<td>New Orleans Arena</td>
<td>0</td>
</tr>
</tbody>
</table>

(5 rows)

REGEXP_REPLACE function

Searches a string for a regular expression pattern and replaces every occurrence of the pattern with the specified string. REGEXP_REPLACE is similar to the REPLACE function (p. 542), but lets you search a string for a regular expression pattern. For more information about regular expressions, see POSIX operators (p. 249).

Synopsis

```sql
REGEXP_REPLACE ( source_string, pattern [, replace_string [, position ] ] )
```

Arguments

- **source_string**
  - A string expression, such as a column name, to be searched.
- **pattern**
  - A string literal that represents a SQL standard regular expression pattern.
- **replace_string**
  - A string expression, such as a column name, that will replace each occurrence of pattern. The default is an empty string (""").
- **position**
  - A positive integer that indicates the position within source_string to begin searching. The position is based on the number of characters, not bytes, so that multibyte characters are counted as single
characters. The default is 1. If position is less than 1, the search begins at the first character of source_string. If position is greater than the number of characters in source_string, the result is source_string.

Return type

VARCHAR

Example

The following example deletes the @ and domain name from email addresses.

```sql
select email, regexp_replace(email, '@.*\.(org|gov|com)$')
from users limit 5;
```

<table>
<thead>
<tr>
<th>email</th>
<th>regexp_replace</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="mailto:DonecFri@semperpretiumneque.com">DonecFri@semperpretiumneque.com</a></td>
<td>DonecFri</td>
</tr>
<tr>
<td><a href="mailto:mklwait@UniofTech.org">mklwait@UniofTech.org</a></td>
<td>mklwait</td>
</tr>
<tr>
<td><a href="mailto:sed@redshiftemails.com">sed@redshiftemails.com</a></td>
<td>sed</td>
</tr>
<tr>
<td><a href="mailto:bunyung@integermath.gov">bunyung@integermath.gov</a></td>
<td>bunyung</td>
</tr>
<tr>
<td><a href="mailto:tomsupporter@galaticmess.org">tomsupporter@galaticmess.org</a></td>
<td>tomsupporter</td>
</tr>
</tbody>
</table>

The following example selects URLs from the fictional WEBSITES table and replaces the domain names with this value: internal.company.com/

```sql
select url, regexp_replace(url, '^.*\.[[:alpha:]]{3}/', 'internal.company.com/')
from websites limit 4;
```

<table>
<thead>
<tr>
<th>url</th>
<th>regexp_replace</th>
</tr>
</thead>
<tbody>
<tr>
<td>example.com/cuisine/locations/home.html</td>
<td>internal.company.com/cuisine/locations/home.html</td>
</tr>
<tr>
<td>anycompany.employersthere.com/employed/A/index.html</td>
<td>internal.company.com/employed/A/index.html</td>
</tr>
<tr>
<td>example.gov/credentials/keys/public</td>
<td>internal.company.com/credentials/keys/public</td>
</tr>
</tbody>
</table>

REGEXP_SUBSTR function

Returns the characters extracted from a string by searching for a regular expression pattern. REGEXP_SUBSTR is similar to the SUBSTRING function (p. 549) function, but lets you search a string
for a regular expression pattern. For more information about regular expressions, see POSIX operators (p. 249).

### Synopsis

```
REGEXP_SUBSTR ( source_string, pattern [, position ] )
```

### Arguments

- **source_string**
  - A string expression, such as a column name, to be searched.
- **pattern**
  - A string literal that represents a SQL standard regular expression pattern.
- **position**
  - A positive integer that indicates the position within `source_string` to begin searching. The position is based on the number of characters, not bytes, so that multi-byte characters are counted as single characters. The default is 1. If `position` is less than 1, the search begins at the first character of `source_string`. If `position` is greater than the number of characters in `source_string`, the result is an empty string ("").

### Return type

VARCHAR

### Example

The following example returns the portion of an email address between the @ character and the domain extension.

```sql
select email, regexp_substr(email,'@[^.]*')
from users limit 5;
```

<table>
<thead>
<tr>
<th>email</th>
<th>regexp_substr</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="mailto:Suspendisse.tristique@nonnisiAenean.edu">Suspendisse.tristique@nonnisiAenean.edu</a></td>
<td>@nonnisiAenean</td>
</tr>
<tr>
<td><a href="mailto:sed@lacusUtnec.ca">sed@lacusUtnec.ca</a></td>
<td>@lacusUtnec</td>
</tr>
<tr>
<td><a href="mailto:elementum@semperpretiumneque.ca">elementum@semperpretiumneque.ca</a></td>
<td>@semperpretiumneque</td>
</tr>
<tr>
<td><a href="mailto:Integer.mollis.Integer@tristiquealiquet.org">Integer.mollis.Integer@tristiquealiquet.org</a></td>
<td>@tristiquealiquet</td>
</tr>
<tr>
<td><a href="mailto:Donec.fringilla@sodalesat.org">Donec.fringilla@sodalesat.org</a></td>
<td>@sodalesat</td>
</tr>
</tbody>
</table>

### REPEAT function

Repeats a string the specified number of times. If the input parameter is numeric, REPEAT treats it as a string.

**Synonym for REPLICATE function (p. 543).**

### Synopsis

```
REPEAT(string, integer)
```
Arguments

string
The first input parameter is the string to be repeated.

integer
The second parameter is an integer indicating the number of times to repeat the string.

Return type
The REPEAT function returns a string.

Examples
The following example repeats the value of the CATID column in the CATEGORY table three times:

```
select catid, repeat(catid, 3)
from category
order by 1, 2;
```

<table>
<thead>
<tr>
<th>catid</th>
<th>repeat</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>111</td>
</tr>
<tr>
<td>2</td>
<td>222</td>
</tr>
<tr>
<td>3</td>
<td>333</td>
</tr>
<tr>
<td>4</td>
<td>444</td>
</tr>
<tr>
<td>5</td>
<td>555</td>
</tr>
<tr>
<td>6</td>
<td>666</td>
</tr>
<tr>
<td>7</td>
<td>777</td>
</tr>
<tr>
<td>8</td>
<td>888</td>
</tr>
<tr>
<td>9</td>
<td>999</td>
</tr>
<tr>
<td>10</td>
<td>101010</td>
</tr>
<tr>
<td>11</td>
<td>111111</td>
</tr>
</tbody>
</table>

(REPLACE function
Replaces all occurrences of a set of characters within an existing string with other specified characters.

Synopsis

```
REPLACE(string1, old_chars, new_chars)
```

Arguments

string
CHAR or VARCHAR string to be searched

old_chars
CHAR or VARCHAR string to replace.

new_chars
New CHAR or VARCHAR string replacing the old_string.
**Return type**

The REPLACE function returns a VARCHAR string, regardless of the data type of the input string (CHAR or VARCHAR).

**Examples**

The following example converts the string Shows to Theatre in the CATGROUP field:

```sql
select catid, catgroup, replace(catgroup, 'Shows', 'Theatre')
from category
order by 1,2,3;
```

<table>
<thead>
<tr>
<th>catid</th>
<th>catgroup</th>
<th>replace</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Sports</td>
<td>Sports</td>
</tr>
<tr>
<td>2</td>
<td>Sports</td>
<td>Sports</td>
</tr>
<tr>
<td>3</td>
<td>Sports</td>
<td>Sports</td>
</tr>
<tr>
<td>4</td>
<td>Sports</td>
<td>Sports</td>
</tr>
<tr>
<td>5</td>
<td>Sports</td>
<td>Sports</td>
</tr>
<tr>
<td>6</td>
<td>Shows</td>
<td>Theatre</td>
</tr>
<tr>
<td>7</td>
<td>Shows</td>
<td>Theatre</td>
</tr>
<tr>
<td>8</td>
<td>Shows</td>
<td>Theatre</td>
</tr>
<tr>
<td>9</td>
<td>Concerts</td>
<td>Concerts</td>
</tr>
<tr>
<td>10</td>
<td>Concerts</td>
<td>Concerts</td>
</tr>
<tr>
<td>11</td>
<td>Concerts</td>
<td>Concerts</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(11 rows)</td>
</tr>
</tbody>
</table>

**REPLICATE function**

Synonym for the REPEAT function.

See [REPEAT function](#) (p. 541).

**REVERSE function**

The REVERSE function operates on a string and returns the characters in reverse order. For example, reverse('abcde') returns edcba. This function works on numeric and date data types as well as character data types; however, in most cases it has practical value for character strings.

**Synopsis**

```sql
REVERSE ( expression )
```

**Argument**

- **expression**
  
  An expression with a character, date, time stamp, or numeric data type that represents the target of the character reversal. All expressions are implicitly converted to variable-length character strings. Trailing blanks in fixed-width character strings are ignored.

**Return type**

REVERSE returns a VARCHAR.
Examples

Select five distinct city names and their corresponding reversed names from the USERS table:

```sql
SELECT DISTINCT city AS cityname, reverse(cityname)
FROM users
ORDER BY city
LIMIT 5;
```

<table>
<thead>
<tr>
<th>cityname</th>
<th>reverse</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aberdeen</td>
<td>needrebA</td>
</tr>
<tr>
<td>Abilene</td>
<td>enelibA</td>
</tr>
<tr>
<td>Ada</td>
<td>adA</td>
</tr>
<tr>
<td>Agat</td>
<td>tagA</td>
</tr>
<tr>
<td>Agawam</td>
<td>mawagA</td>
</tr>
</tbody>
</table>
(5 rows)

Select five sales IDs and their corresponding reversed IDs cast as character strings:

```sql
SELECT salesid, reverse(salesid)::varchar
FROM sales
ORDER BY salesid DESC
LIMIT 5;
```

<table>
<thead>
<tr>
<th>salesid</th>
<th>reverse</th>
</tr>
</thead>
<tbody>
<tr>
<td>172456</td>
<td>654271</td>
</tr>
<tr>
<td>172455</td>
<td>554271</td>
</tr>
<tr>
<td>172454</td>
<td>454271</td>
</tr>
<tr>
<td>172453</td>
<td>354271</td>
</tr>
<tr>
<td>172452</td>
<td>254271</td>
</tr>
</tbody>
</table>
(5 rows)

RTRIM function

The RTRIM function trims a specified set of characters from the end of a string.

Synopsis

```sql
RTRIM( string, trim_chars )
```

Arguments

`string`

The string column or expression to be trimmed.

`trim_chars`

A string column or expression representing the characters to be trimmed from the end of `string`.

Return type

A string that is the same data type as the `string` argument.

Example

The following example trims the characters ‘Park’ from the end of VENUENAME where present:
```
select venueid, venuename, rtrim(venuename, 'Park')
from venue
order by 1, 2, 3
limit 10;
```

<table>
<thead>
<tr>
<th>venueid</th>
<th>venuename</th>
<th>rtrim</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Toyota Park</td>
<td>Toyota</td>
</tr>
<tr>
<td>2</td>
<td>Columbus Crew Stadium</td>
<td>Columbus Crew Stadium</td>
</tr>
<tr>
<td>3</td>
<td>RFK Stadium</td>
<td>RFK Stadium</td>
</tr>
<tr>
<td>4</td>
<td>CommunityAmerica Ballpark</td>
<td>CommunityAmerica Ballpark</td>
</tr>
<tr>
<td>5</td>
<td>Gillette Stadium</td>
<td>Gillette Stadium</td>
</tr>
<tr>
<td>6</td>
<td>New York Giants Stadium</td>
<td>New York Giants Stadium</td>
</tr>
<tr>
<td>7</td>
<td>BMO Field</td>
<td>BMO Field</td>
</tr>
<tr>
<td>8</td>
<td>The Home Depot Center</td>
<td>The Home Depot Centre</td>
</tr>
<tr>
<td>9</td>
<td>Dick's Sporting Goods Park</td>
<td>Dick's Sporting Goods</td>
</tr>
<tr>
<td>10</td>
<td>Pizza Hut Park</td>
<td>Pizza Hut</td>
</tr>
</tbody>
</table>

(10 rows)

Note that RTRIM removes any of the characters P, a, r, or k when they appear at the end of a VENUENAME.

**SET_BIT function**

SET_BIT is a deprecated leader-node-only function.

**SET_BYTE function**

SET_BYTE is a deprecated leader-node-only function.

**SPLIT_PART function**

Splits a string on the specified delimiter and returns the part at the specified position.

**Synopsis**

```
SPLIT_PART(string, delimiter, part)
```

**Arguments**

- **string**
  The string to be split. The string can be CHAR or VARCHAR.

- **delimiter**
  The delimiter string.
  
    If delimiter is a literal, enclose it in single quotes.

- **part**
  Position of the portion to return (counting from 1). Must be an integer greater than 0. If part is larger than the number of string portions, SPLIT_PART returns an empty string.

**Return type**

A CHAR or VARCHAR string, the same as the string parameter.
Examples

The following example splits the time stamp field LISTTIME into year, month, and date components.

```sql
select listtime, split_part(listtime,'-',1) as year,
       split_part(listtime,'-',2) as month,
       split_part(split_part(listtime,'-',3),' ',1) as date
from listing limit 5;
```

<table>
<thead>
<tr>
<th>listtime</th>
<th>year</th>
<th>month</th>
<th>date</th>
</tr>
</thead>
<tbody>
<tr>
<td>2008-03-05 12:25:29</td>
<td>2008</td>
<td>03</td>
<td>05</td>
</tr>
<tr>
<td>2008-09-09 08:03:36</td>
<td>2008</td>
<td>09</td>
<td>09</td>
</tr>
<tr>
<td>2008-09-26 05:43:12</td>
<td>2008</td>
<td>09</td>
<td>26</td>
</tr>
<tr>
<td>2008-10-04 02:00:30</td>
<td>2008</td>
<td>10</td>
<td>04</td>
</tr>
<tr>
<td>2008-01-06 08:33:11</td>
<td>2008</td>
<td>01</td>
<td>06</td>
</tr>
</tbody>
</table>

(5 rows)

The following example selects the LISTTIME time stamp field and splits it on the ‘-’ character to get the month (the second part of the LISTTIME string), then counts the number of entries for each month:

```sql
select split_part(listtime,'-',2) as month, count(*)
from listing
group by split_part(listtime,'-',2)
order by 1, 2;
```

<table>
<thead>
<tr>
<th>month</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>18543</td>
</tr>
<tr>
<td>02</td>
<td>16620</td>
</tr>
<tr>
<td>03</td>
<td>17594</td>
</tr>
<tr>
<td>04</td>
<td>16822</td>
</tr>
<tr>
<td>05</td>
<td>17618</td>
</tr>
<tr>
<td>06</td>
<td>17158</td>
</tr>
<tr>
<td>07</td>
<td>17626</td>
</tr>
<tr>
<td>08</td>
<td>17881</td>
</tr>
<tr>
<td>09</td>
<td>17378</td>
</tr>
<tr>
<td>10</td>
<td>17756</td>
</tr>
<tr>
<td>11</td>
<td>12912</td>
</tr>
<tr>
<td>12</td>
<td>4589</td>
</tr>
</tbody>
</table>

(12 rows)

**STRPOS function**

Returns the position of a substring within a specified string.

Synopsis of **CHARINDEX function (p. 523)** and **POSITION function (p. 534)**.

**Synopsis**

```
STRPOS(string, substring )
```
Arguments

string
The first input parameter is the string to be searched.

substring
The second parameter is the substring to search for within the string.

Return type

The STRPOS function returns an integer corresponding to the position of the substring (one-based, not zero-based). The position is based on the number of characters, not bytes, so that multi-byte characters are counted as single characters.

Usage notes

STRPOS returns 0 if the substring is not found within the string:

```
select strpos('dogfish', 'fist');
```

| strpos | 0 |
|--------+---|
| (1 row) |   |

Examples

The following example shows the position of the string fish within the word dogfish:

```
select strpos('dogfish', 'fish');
```

| strpos | 4 |
|--------+---|
| (1 row) |   |

The following example returns the number of sales transactions with a COMMISSION over 999.00 from the SALES table:

```
select distinct strpos(commission, '.'),
       count (strpos(commission, '.'))
from sales
where strpos(commission, '.') > 4
group by strpos(commission, '.')
order by 1, 2;
```

| strpos | count |
|--------+------|
| 5      | 629   |
| (1 row) |      |

STRTOL function

Converts a string expression of a number of the specified base to the equivalent integer value. The converted value must be within the signed 64-bit range.
Syntax

```c
STRTOL(num_string, base)
```

Arguments

`num_string`
String expression of a number to be converted. If `num_string` is empty ('') or begins with the null character ('\0'), the converted value is 0. If `num_string` is a column containing a NULL value, `STRTOL` returns NULL. The string can begin with any amount of white space, optionally followed by a single plus '+' or minus '-' sign to indicate positive or negative. The default is '+'. If `base` is 16, the string can optionally begin with '0x'.

`base`
Integer between 2 and 36.

Return type

BIGINT. If `num_string` is null, returns NULL.

Examples

The following examples convert string and base value pairs to integers:

```sql
select strtol('0xf',16);
strtol
--------
   15
(1 row)

select strtol('abcd1234',16);
strtol
----------
  2882343476
(1 row)

select strtol('1234567', 10);
strtol
--------
1234567
(1 row)

select strtol('1234567', 8);
strtol
--------
   342391
(1 row)

select strtol('110101', 2);
strtol
--------
    53

select strtol('\0', 2);
strtol

---

API Version 2012-12-01
548

Amazon Redshift Database Developer Guide
String functions
**SUBSTRING function**

Returns the characters extracted from a string based on the specified character position for a specified number of characters.

The character position and number of characters are based on the number of characters, not bytes, so that multi-byte characters are counted as single characters. You cannot specify a negative length, but you can specify a negative starting position.

**Synopsis**

```
SUBSTRING(string FROM start_position [ FOR number_characters ] )
```

```
SUBSTRING(string, start_position, number_characters )
```

**Arguments**

- **string**
  The first input parameter is the string to be searched. Non-character data types are treated like a string.

- **start_position**
  The second parameter is an integer that is the one-based position within the string to begin the extraction. The `start_position` is based on the number of characters, not bytes, so that multi-byte characters are counted as single characters. This number can be negative.

- **number_characters**
  The third parameter is an integer that is the number of characters to extract (the length of the substring). The `number_characters` is based on the number of characters, not bytes, so that multi-byte characters are counted as single characters. This number cannot be negative.

**Return type**

The SUBSTRING function returns a VARCHAR string.

**Usage notes**

If the `start_position + number_characters` exceeds the length of the `string`, SUBSTRING returns a substring starting from the `start_position` until the end of the string. For example:

```
select substring('caterpillar',6,4);
```

```
substring
-----------
pill
(1 row)
```

If the `start_position` is negative or 0, the SUBSTRING function returns a substring beginning at the first character of string with a length of `start_position + number_characters` -1. For example:
select substring('caterpillar',-2,6);
substring
-----------
cat
(1 row)

If `start_position + number_characters - 1` is less than or equal to zero, SUBSTRING returns an empty string. For example:

select substring('caterpillar',-5,4);
substring
-----------
(1 row)

**Examples**

The following example returns the month from the LISTTIME string in the LISTING table:

```sql
select listid, listtime,
substring(listtime, 6, 2) as month
from listing
order by 1, 2, 3
limit 10;
```

| listid |      listtime       | month |
|--------+---------------------+-------|
| 1      | 2008-01-24 06:43:29 | 01    |
| 2      | 2008-03-05 12:25:29 | 03    |
| 3      | 2008-11-01 07:35:33 | 11    |
| 4      | 2008-05-24 01:18:37 | 05    |
| 5      | 2008-05-17 02:29:11 | 05    |
| 6      | 2008-08-15 02:08:13 | 08    |
| 7      | 2008-11-15 09:38:15 | 11    |
| 8      | 2008-11-09 05:07:30 | 11    |
| 9      | 2008-09-09 08:03:36 | 09    |
| 10     | 2008-06-17 09:44:54 | 06    |
(10 rows)
```

The following example is the same as above, but uses the FROM...FOR option:

```sql
select listid, listtime,
substring(listtime from 6 for 2) as month
from listing
order by 1, 2, 3
limit 10;
```

| listid |      listtime       | month |
|--------+---------------------+-------|
| 1      | 2008-01-24 06:43:29 | 01    |
| 2      | 2008-03-05 12:25:29 | 03    |
| 3      | 2008-11-01 07:35:33 | 11    |
| 4      | 2008-05-24 01:18:37 | 05    |
| 5      | 2008-05-17 02:29:11 | 05    |
| 6      | 2008-08-15 02:08:13 | 08    |
You cannot use SUBSTRING to predictably extract the prefix of a string that might contain multi-byte characters because you need to specify the length of a multi-byte string based on the number of bytes, not the number of characters. To extract the beginning segment of a string based on the length in bytes, you can CAST the string as VARCHAR(byte_length) to truncate the string, where byte_length is the required length. The following example extracts the first 5 bytes from the string 'Fourscore and seven'.

```sql
select cast('Fourscore and seven' as varchar(5));
```

<table>
<thead>
<tr>
<th>varchar</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fours</td>
</tr>
</tbody>
</table>

**TEXTLEN function**

Synonym of LEN function.

See LEN function (p. 530).

**TO_ASCII function**

TO_ASCII is a deprecated leader-node-only function.

**TO_HEX function**

The TO_HEX function converts a number to its equivalent hexadecimal value.

**Syntax**

```sql
TO_HEX(string)
```

**Arguments**

`string`

The input parameter is a number to convert to its hexadecimal value.

**Return type**

The TO_HEX function returns a hexadecimal value.

**Examples**

The following example shows the conversion of a number to its hexadecimal value:

```sql
select to_hex(2147676847);
```

<table>
<thead>
<tr>
<th>to_hex</th>
</tr>
</thead>
<tbody>
<tr>
<td>8EDEEH</td>
</tr>
</tbody>
</table>

Amazon Redshift Database Developer Guide

String functions
TRIM function

The TRIM function trims a specified set of characters from the beginning and end of a string.

Synopsis

\[
\text{TRIM( [ BOTH ] 'characters' FROM string )}
\]

Arguments

- **characters**
  - The first input parameter is a CHAR or VARCHAR string representing the characters to be trimmed from the string.
- **string**
  - The second parameter is the CHAR or VARCHAR string to be trimmed.

Return type

The TRIM function returns a VARCHAR or TEXT string. If you use the TRIM function with a SQL command, Amazon Redshift implicitly converts the results to VARCHAR. If you use the TRIM function in the SELECT list for a SQL function, Amazon Redshift does not implicitly

Example

The following example removes the double quotes that surround the string "dog":

```sql
select trim('"' FROM '"dog"');
```

<table>
<thead>
<tr>
<th>btrim</th>
</tr>
</thead>
<tbody>
<tr>
<td>dog</td>
</tr>
<tr>
<td>(1 row)</td>
</tr>
</tbody>
</table>

UPPER function

Converts a string to uppercase. UPPER supports UTF-8 multibyte characters, up to a maximum of four bytes per character.

Synopsis

\[
\text{UPPER(string)}
\]

Arguments

- **string**
  - The input parameter is a CHAR or VARCHAR string.
Return type

The UPPER function returns a character string that is the same data type as the input string (CHAR or VARCHAR).

Examples

The following example converts the CATNAME field to uppercase:

```sql
select catname, upper(catname) from category order by 1,2;
```

<table>
<thead>
<tr>
<th>catname</th>
<th>upper</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classical</td>
<td>CLASSICAL</td>
</tr>
<tr>
<td>Jazz</td>
<td>JAZZ</td>
</tr>
<tr>
<td>MLB</td>
<td>MLB</td>
</tr>
<tr>
<td>MLS</td>
<td>MLS</td>
</tr>
<tr>
<td>Musicals</td>
<td>MUSICALS</td>
</tr>
<tr>
<td>NBA</td>
<td>NBA</td>
</tr>
<tr>
<td>NFL</td>
<td>NFL</td>
</tr>
<tr>
<td>NHL</td>
<td>NHL</td>
</tr>
<tr>
<td>Opera</td>
<td>OPERA</td>
</tr>
<tr>
<td>Plays</td>
<td>PLAYS</td>
</tr>
<tr>
<td>Pop</td>
<td>POP</td>
</tr>
</tbody>
</table>

(11 rows)

JSON Functions

Topics

- JSON_ARRAY_LENGTH function (p. 554)
- JSON_EXTRACT_ARRAY_ELEMENT_TEXT function (p. 554)
- JSON_EXTRACT_PATH_TEXT function (p. 555)

When you need to store a relatively small set of key-value pairs, you might save space by storing the data in JSON format. Because JSON strings can be stored in a single column, using JSON might be more efficient than storing your data in tabular format. For example, suppose you have a sparse table, where you need to have many columns to fully represent all possible attributes, but most of the column values are NULL for any given row or any given column. By using JSON for storage, you might be able to store the data for a row in key:value pairs in a single JSON string and eliminate the sparsely-populated table columns.

In addition, you can easily modify JSON strings to store additional key:value pairs without needing to add columns to a table.

We recommend using JSON sparingly. JSON is not a good choice for storing larger data sets because, by storing disparate data in a single column, JSON does not leverage Amazon Redshift's column store architecture.

JSON uses UTF-8 encoded text strings, so JSON strings can be stored as CHAR or VARCHAR data types. Use VARCHAR if the strings include multi-byte characters.

JSON strings must be properly formatted JSON, according to the following rules:

- The root level JSON can either be a JSON object or a JSON array. A JSON object is an unordered set of comma-separated key:value pairs enclosed by curly braces.
For example, \{"one" : 1, "two" : 2\}
• A JSON array is an ordered set of comma-separated values enclosed by square brackets.

For example, ["first", {"one" : 1}, "second", 3, null]
• JSON arrays use a zero-based index; the first element in an array is at position 0. In a JSON key:value pair, the key is a double quoted string.
• A JSON value can be any of:
  • JSON object
  • JSON array
  • string (double quoted)
  • number (integer and float)
  • boolean
  • null
• Empty objects and empty arrays are valid JSON values.
• JSON fields are case sensitive.
• White space between JSON structural elements (such as { }, [ ]) is ignored.

**JSON_ARRAY_LENGTH function**

JSON_ARRAY_LENGTH returns the number of elements in the outer array of a JSON string.

For more information, see JSON Functions (p. 553).

**Synopsis**

```
json_array_length('json_array')
```

**Arguments**

*json_array*

A properly formatted JSON array.

**Return type**

An integer representing the number of elements in the outermost array.

**Example**

The following example returns the number of elements in the array:

```
select json_array_length('[11,12,13,{"f1":21,"f2":[25,26]},14]');
```

```
json_array_length
-----------------
5
```

**JSON_EXTRACT_ARRAY_ELEMENT_TEXT function**

This function returns a JSON array element in the outermost array of a JSON string, using a zero-based index. The first element in an array is at position 0. If the index is negative or out of bound, JSON_EXTRACT_ARRAY_ELEMENT_TEXT returns empty string.
For more information, see JSON Functions (p. 553).

**Synopsis**

```sql
json_extract_array_element_text('json string', pos)
```

**Arguments**

- `json_string`:
  A properly formatted JSON string.
- `pos`:
  An integer representing the index of the array element to be returned, using a zero-based array index.

**Return type**

A VARCHAR string representing the JSON array element referenced by `pos`.

**Example**

The following example returns array element at position 2:

```sql
select json_extract_array_element_text('[111,112,113]', 2);
```

```
113
```

**JSON_EXTRACT_PATH_TEXT function**

JSON_EXTRACT_PATH_TEXT returns the value for the key:value pair referenced by a series of path elements in a JSON string. The JSON path can be nested up to five levels deep. Path elements are case-sensitive. If a path element does not exist in the JSON string, JSON_EXTRACT_PATH_TEXT returns an empty string.

For more information, see JSON Functions (p. 553).

**Synopsis**

```sql
json_extract_path_text('json_string', 'path_elem', [',path_elem'][, ...])
```

**Arguments**

- `json_string`:
  A properly formatted JSON string.
- `path_elem`:
  A path element in a JSON string. One `path_elem` is required. Additional path elements can be specified, up to five levels deep.

**Return type**

VARCHAR string representing the JSON value referenced by the path elements.
Amazon Redshift Database Developer Guide
Data type formatting functions

Example
The following example returns the value for the path 'f4', 'f6':
select json_extract_path_text('{"f2":{"f3":1},"f4":{"f5":99,"f6":"star"}}','f4',
'f6');
json_extract_path_text
---------------------star

Data type formatting functions
Topics
• CAST and CONVERT functions (p. 556)
• TO_CHAR (p. 559)
•
•
•
•

TO_DATE (p. 561)
TO_NUMBER (p. 562)
Datetime format strings (p. 563)
Numeric format strings (p. 564)

Data type formatting functions provide an easy way to convert values from one data type to another. For
each of these functions, the first argument is always the value to be formatted and the second argument
contains the template for the new format. Amazon Redshift supports several data type formatting functions.

CAST and CONVERT functions
You can do run-time conversions between compatible data types by using the CAST and CONVERT
functions.
Certain data types require an explicit conversion to other data types using the CAST or CONVERT
function. Other data types can be converted implicitly, as part of another command, without using the
CAST or CONVERT function. See Type compatibility and conversion (p. 232).

CAST
You can use two equivalent syntax forms to cast expressions from one data type to another:
CAST ( expression AS type )
expression :: type

Arguments
expression
An expression that evaluates to one or more values, such as a column name or a literal. Converting
null values returns nulls. The expression cannot contain blank or empty strings.
type
One of the supported Data types (p. 214).

API Version 2012-12-01
556


Return type

CAST returns the data type specified by the `type` argument.

**Note**
Amazon Redshift returns an error if you try to perform a problematic conversion such as the following DECIMAL conversion that loses precision:

```
select 123.456::decimal(2,1);
```

or an INTEGER conversion that causes an overflow:

```
select 12345678::smallint;
```

**CONVERT**

You can also use the CONVERT function to convert values from one data type to another:

```
CONVERT ( type, expression )
```

**Arguments**

- `type` One of the supported Data types (p. 214).
- `expression` An expression that evaluates to one or more values, such as a column name or a literal. Converting null values returns nulls. The expression cannot contain blank or empty strings.

**Return type**

CONVERT returns the data type specified by the `type` argument.

**Examples**

The following two queries are equivalent. They both cast a decimal value to an integer:

```
select cast(pricepaid as integer)
from sales where salesid=100;
```

```
<table>
<thead>
<tr>
<th>pricepaid</th>
</tr>
</thead>
<tbody>
<tr>
<td>162</td>
</tr>
<tr>
<td>(1 row)</td>
</tr>
</tbody>
</table>
```

```
select pricepaid::integer
from sales where salesid=100;
```

```
<table>
<thead>
<tr>
<th>pricepaid</th>
</tr>
</thead>
<tbody>
<tr>
<td>162</td>
</tr>
<tr>
<td>(1 row)</td>
</tr>
</tbody>
</table>
```

The following query uses the CONVERT function to return the same result:
select convert(integer, pricepaid)
from sales where salesid=100;

<table>
<thead>
<tr>
<th>pricepaid</th>
</tr>
</thead>
<tbody>
<tr>
<td>162</td>
</tr>
</tbody>
</table>

(1 row)

In this example, the values in a time stamp column are cast as dates:

select cast(saletime as date), salesid
from sales order by salesid limit 10;

<table>
<thead>
<tr>
<th>saletime</th>
<th>salesid</th>
</tr>
</thead>
<tbody>
<tr>
<td>2008-02-18</td>
<td>1</td>
</tr>
<tr>
<td>2008-06-06</td>
<td>2</td>
</tr>
<tr>
<td>2008-06-06</td>
<td>3</td>
</tr>
<tr>
<td>2008-06-09</td>
<td>4</td>
</tr>
<tr>
<td>2008-08-31</td>
<td>5</td>
</tr>
<tr>
<td>2008-07-16</td>
<td>6</td>
</tr>
<tr>
<td>2008-06-26</td>
<td>7</td>
</tr>
<tr>
<td>2008-07-10</td>
<td>8</td>
</tr>
<tr>
<td>2008-07-22</td>
<td>9</td>
</tr>
<tr>
<td>2008-08-06</td>
<td>10</td>
</tr>
</tbody>
</table>

(10 rows)

In this example, the values in a date column are cast as time stamps:

select cast(caldate as timestamp), dateid
from date order by dateid limit 10;

<table>
<thead>
<tr>
<th>caldate</th>
<th>dateid</th>
</tr>
</thead>
<tbody>
<tr>
<td>2008-01-01 00:00:00</td>
<td>1827</td>
</tr>
<tr>
<td>2008-01-02 00:00:00</td>
<td>1828</td>
</tr>
<tr>
<td>2008-01-03 00:00:00</td>
<td>1829</td>
</tr>
<tr>
<td>2008-01-04 00:00:00</td>
<td>1830</td>
</tr>
<tr>
<td>2008-01-05 00:00:00</td>
<td>1831</td>
</tr>
<tr>
<td>2008-01-06 00:00:00</td>
<td>1832</td>
</tr>
<tr>
<td>2008-01-07 00:00:00</td>
<td>1833</td>
</tr>
<tr>
<td>2008-01-08 00:00:00</td>
<td>1834</td>
</tr>
<tr>
<td>2008-01-09 00:00:00</td>
<td>1835</td>
</tr>
<tr>
<td>2008-01-10 00:00:00</td>
<td>1836</td>
</tr>
</tbody>
</table>

(10 rows)

In this example, an integer is cast as a character string:

select cast(2008 as char(4));

<table>
<thead>
<tr>
<th>bpchar</th>
</tr>
</thead>
<tbody>
<tr>
<td>2008</td>
</tr>
</tbody>
</table>

In this example, a DECIMAL(6,3) value is cast as a DECIMAL(4,1) value:
select cast(109.652 as decimal(4,1));
numeric
--------
109.7

In this example, the PRICEPAID column (a DECIMAL(8,2) column) in the SALES table is converted to a
DECIMAL(38,2) column and the values are multiplied by 100000000000000000000.

select salesid, pricepaid::decimal(38,2)*100000000000000000000 as value from sales where salesid<10 order by salesid;

<table>
<thead>
<tr>
<th>salesid</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>72800000000000000000000.00</td>
</tr>
<tr>
<td>2</td>
<td>76000000000000000000000.00</td>
</tr>
<tr>
<td>3</td>
<td>35000000000000000000000.00</td>
</tr>
<tr>
<td>4</td>
<td>17500000000000000000000.00</td>
</tr>
<tr>
<td>5</td>
<td>15400000000000000000000.00</td>
</tr>
<tr>
<td>6</td>
<td>39400000000000000000000.00</td>
</tr>
<tr>
<td>7</td>
<td>78800000000000000000000.00</td>
</tr>
<tr>
<td>8</td>
<td>19700000000000000000000.00</td>
</tr>
<tr>
<td>9</td>
<td>59100000000000000000000.00</td>
</tr>
</tbody>
</table>

(9 rows)

**TO_CHAR**

TO_CHAR converts a time stamp or numeric expression to a character-string data format.

**Synopsis**

TO_CHAR (timestamp_expression | numeric_expression , 'format')

**Arguments**

timestamp_expression
An expression that results in a time stamp type value or a value that can implicitly be coerced to a
time stamp.

numeric_expression
An expression that results in a numeric data type value or a value that can implicitly be coerced to
a numeric type. See Numeric types (p. 216).

**Note**
TO_CHAR does not support 128-bit DECIMAL values.

format
Format for the new value. See Datetime format strings (p. 563) and Numeric format strings (p. 564)
for valid formats.

**Return type**

TO_CHAR returns a VARCHAR data type.
Examples

The following example converts each STARTTIME value in the EVENT table to a string that consists of hours, minutes, and seconds:

```sql
select to_char(starttime, 'HH12:MI:SS')
from event where eventid between 1 and 5
order by eventid;
```

```
to_char
----------
02:30:00
08:00:00
02:30:00
02:30:00
07:00:00
(5 rows)
```

The following example converts an entire timestamp value into a different format:

```sql
select starttime, to_char(starttime, 'MON-DD-YYYY HH12:MI:PM')
from event where eventid=1;
```

```
| starttime      |       to_char            |
|---------------------+---------------------|
| 2008-01-25 14:30:00 | JAN-25-2008 02:30PM |
```

The following example converts a timestamp literal to a character string:

```sql
select to_char(timestamp '2009-12-31 23:15:59','HH24:MI:SS');
```

```
to_char
----------
23:15:59
```

The following example converts an integer to a character string:

```sql
select to_char(-125.8, '999D99S');
```

```
to_char
---------
125.80-
```

The following example subtracts the commission from the price paid in the sales table. The difference is then rounded up and converted to a roman numeral, shown in the to_char column:

```sql
select salesid, pricepaid, commission, (pricepaid - commission) as difference, to_char(pricepaid - commission, 'rn') from sales
order by salesid limit 10;
```

```
salesid | pricepaid | commission | difference |   to_char
---------+-----------+------------+------------+-----------------|
```
The following example adds the currency symbol to the difference values shown in the to_char column:

```sql
select salesid, pricepaid, commission, (pricepaid - commission) as difference, to_char(pricepaid - commission, 'l99999D99') from sales
order by salesid limit 10;
```

<table>
<thead>
<tr>
<th>salesid</th>
<th>pricepaid</th>
<th>commission</th>
<th>difference</th>
<th>to_char</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>728.00</td>
<td>109.20</td>
<td>618.80</td>
<td>$618.80</td>
</tr>
<tr>
<td>2</td>
<td>76.00</td>
<td>11.40</td>
<td>64.60</td>
<td>$64.60</td>
</tr>
<tr>
<td>3</td>
<td>350.00</td>
<td>52.50</td>
<td>297.50</td>
<td>$297.50</td>
</tr>
<tr>
<td>4</td>
<td>175.00</td>
<td>26.25</td>
<td>148.75</td>
<td>$148.75</td>
</tr>
<tr>
<td>5</td>
<td>154.00</td>
<td>23.10</td>
<td>130.90</td>
<td>$130.90</td>
</tr>
<tr>
<td>6</td>
<td>394.00</td>
<td>59.10</td>
<td>334.90</td>
<td>$334.90</td>
</tr>
<tr>
<td>7</td>
<td>788.00</td>
<td>118.20</td>
<td>669.80</td>
<td>$669.80</td>
</tr>
<tr>
<td>8</td>
<td>197.00</td>
<td>29.55</td>
<td>167.45</td>
<td>$167.45</td>
</tr>
<tr>
<td>9</td>
<td>591.00</td>
<td>88.65</td>
<td>502.35</td>
<td>$502.35</td>
</tr>
<tr>
<td>10</td>
<td>65.00</td>
<td>9.75</td>
<td>55.25</td>
<td>$55.25</td>
</tr>
</tbody>
</table>

(10 rows)

The following example lists the century in which each sale was made.

```sql
select salesid, saletime, to_char(saletime, 'cc') from sales
order by salesid limit 10;
```

<table>
<thead>
<tr>
<th>salesid</th>
<th>saletime</th>
<th>to_char</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2008-02-18 02:36:48</td>
<td>21</td>
</tr>
<tr>
<td>2</td>
<td>2008-06-06 05:00:16</td>
<td>21</td>
</tr>
<tr>
<td>3</td>
<td>2008-06-06 08:26:17</td>
<td>21</td>
</tr>
<tr>
<td>4</td>
<td>2008-06-09 08:38:52</td>
<td>21</td>
</tr>
<tr>
<td>5</td>
<td>2008-08-31 09:17:02</td>
<td>21</td>
</tr>
<tr>
<td>6</td>
<td>2008-07-16 11:59:24</td>
<td>21</td>
</tr>
<tr>
<td>7</td>
<td>2008-06-26 12:56:06</td>
<td>21</td>
</tr>
<tr>
<td>8</td>
<td>2008-07-10 02:12:36</td>
<td>21</td>
</tr>
<tr>
<td>9</td>
<td>2008-07-22 02:23:17</td>
<td>21</td>
</tr>
<tr>
<td>10</td>
<td>2008-08-06 02:51:55</td>
<td>21</td>
</tr>
</tbody>
</table>

(10 rows)

**TO_DATE**

TO_DATE converts a date represented in a character string to a DATE data type.
The second argument is a format string that indicates how the character string should be parsed to create the date value.

**Synopsis**

```
TO_DATE (string, format)
```

**Arguments**

- **string**
  - String to be converted.
- **format**
  - A string literal that defines the format of the string to be converted, in terms of its date parts. For a list of valid formats, see Datetime format strings (p. 563).

**Return type**

TO_DATE returns a DATE, depending on the *format* value.

**Example**

The following command converts the date 02 Oct 2001 into the default date format:

```
select to_date ('02 Oct 2001', 'DD Mon YYYY');
```

```
to_date
------------
2001-10-02
(1 row)
```

**TO_NUMBER**

TO_NUMBER converts a string to a numeric (decimal) value.

**Synopsis**

```
to_number(string, format)
```

**Arguments**

- **string**
  - String to be converted. The format must be a literal value.
- **format**
  - The second argument is a format string that indicates how the character string should be parsed to create the numeric value. For example, the format '"99D999' specifies that the string to be converted consists of five digits with the decimal point in the third position. For example, `to_number('12.345','99D999')` returns 12.345 as a numeric value. For a list of valid formats, see Numeric format strings (p. 564).

**Return type**

TO_NUMBER returns a DECIMAL number.
Examples

The following example converts the string 12,454.8- to a number:

```sql
select to_number('12,454.8-', '99G999D9S');
```

```
to_number
-----------
-12454.8
```

(1 row)

Datetime format strings

This topic provides a reference for datetime format strings.

The following format strings apply to functions such as TO_CHAR. These strings can contain datetime separators (such as '-', '/', or ':') and the following "dateparts" and "timeparts":

<table>
<thead>
<tr>
<th>Datepart/timepart</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>BC or B.C., AD or A.D., b.c. or bc, ad or a.d.</td>
<td>Upper and lowercase era indicators</td>
</tr>
<tr>
<td>CC</td>
<td>Two-digit century</td>
</tr>
<tr>
<td>YYYY, YYY, YY, Y</td>
<td>4-digit, 3-digit, 2-digit, 1-digit year</td>
</tr>
<tr>
<td>Y,YYY</td>
<td>4-digit year with comma</td>
</tr>
<tr>
<td>IYYYY, IYY, IY, I</td>
<td>4-digit, 3-digit, 2-digit, 1-digit ISO year</td>
</tr>
<tr>
<td>Q</td>
<td>Quarter number (1 to 4)</td>
</tr>
<tr>
<td>MONTH, Month, month</td>
<td>Month name (uppercase, mixed-case, lowercase, blank-padded to 9 characters)</td>
</tr>
<tr>
<td>MON, Mon, mon</td>
<td>Abbreviated month name (uppercase, mixed-case, lowercase, blank-padded to 9 characters)</td>
</tr>
<tr>
<td>MM</td>
<td>Month number (01-12)</td>
</tr>
<tr>
<td>RM, rm</td>
<td>Month in Roman numerals (I-XII; I=January) (uppercase or lowercase)</td>
</tr>
<tr>
<td>W</td>
<td>Week of month (1-5) (The first week starts on the first day of the month.)</td>
</tr>
<tr>
<td>WW</td>
<td>Week number of year (1-53) (The first week starts on the first day of the year.)</td>
</tr>
<tr>
<td>IW</td>
<td>ISO week number of year (The first Thursday of the new year is in week 1.)</td>
</tr>
<tr>
<td>DAY, Day, day</td>
<td>Day name (uppercase, mixed-case, lowercase, blank-padded to 9 characters)</td>
</tr>
<tr>
<td>DY, Dy, dy</td>
<td>Abbreviated day name (uppercase, mixed-case, lowercase, blank-padded to 9 characters)</td>
</tr>
<tr>
<td>DDD</td>
<td>Day of year (001-366)</td>
</tr>
</tbody>
</table>
## Datepart/timepart

<table>
<thead>
<tr>
<th>Datepart/timepart</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>DD</td>
<td>Day of month as a number (01-31)</td>
</tr>
<tr>
<td>D</td>
<td>Day of week (1-7; Sunday is 1)</td>
</tr>
<tr>
<td>J</td>
<td>Julian Day (days since January 1, 4712 BC)</td>
</tr>
<tr>
<td>HH24</td>
<td>Hour (24-hour clock, 00-23)</td>
</tr>
<tr>
<td>HH or HH12</td>
<td>Hour (12-hour clock, 01-12)</td>
</tr>
<tr>
<td>MI</td>
<td>Minutes (00-59)</td>
</tr>
<tr>
<td>SS</td>
<td>Seconds (00-59)</td>
</tr>
<tr>
<td>AM or PM, A.M. or P.M., a.m. or p.m., am or pm</td>
<td>Upper and lowercase meridian indicators (for 12-hour clock)</td>
</tr>
</tbody>
</table>

**Note**

Time zone formatting is not supported.

## Numeric format strings

This topic provides a reference for numeric format strings.

The following format strings apply to functions such as `TO_NUMBER` and `TO_CHAR`:

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>Numeric value with the specified number of digits.</td>
</tr>
<tr>
<td>0</td>
<td>Numeric value with leading zeros.</td>
</tr>
<tr>
<td>. (period), D</td>
<td>Decimal point.</td>
</tr>
<tr>
<td>, (comma)</td>
<td>Thousands separator.</td>
</tr>
<tr>
<td>CC</td>
<td>Century code. For example, the 21st century started on 2001-01-01 (supported for <code>TO_CHAR</code> only).</td>
</tr>
<tr>
<td>FM</td>
<td>Fill mode. Suppress padding blanks and zeroes.</td>
</tr>
<tr>
<td>PR</td>
<td>Negative value in angle brackets.</td>
</tr>
<tr>
<td>S</td>
<td>Sign anchored to a number.</td>
</tr>
<tr>
<td>L</td>
<td>Currency symbol in the specified position.</td>
</tr>
<tr>
<td>G</td>
<td>Group separator.</td>
</tr>
<tr>
<td>MI</td>
<td>Minus sign in the specified position for numbers that are less than 0.</td>
</tr>
<tr>
<td>PL</td>
<td>Plus sign in the specified position for numbers that are greater than 0.</td>
</tr>
<tr>
<td>SG</td>
<td>Plus or minus sign in the specified position.</td>
</tr>
</tbody>
</table>
Amazon Redshift supports several system administration functions.

**CURRENT_SETTING**

CURRENT_SETTING returns the current value of the specified configuration parameter.

This function is equivalent to the SHOW (p. 393) command.

**Synopsis**

```
current_setting('parameter')
```

**Argument**

*parameter*

Parameter value to display. For a list of configuration parameters, see [Configuration Reference (p. 693)](#).

**Return type**

Returns a CHAR or VARCHAR string.

**Example**

The following query returns the current setting for the `query_group` parameter:

```
select current_setting('query_group')
```

```
current_setting
-----------
unset
(1 row)
```
PG_CANCEL_BACKEND

Cancels a query. PG_CANCEL_BACKEND is functionally equivalent to the CANCEL (p. 272) command. You can cancel queries currently being run by your user. Superusers can cancel any query.

Synopsis

```
pg_cancel_backend( pid )
```

Arguments

**pid**

The process ID (PID) of the query to be canceled. You cannot cancel a query by specifying a query ID; you must specify the query’s process ID. Requires an integer value.

Return type

None

Usage Notes

If you cannot cancel a query because it is in transaction block (BEGIN ... END), you can terminate the session in which the query is running by using the PG_TERMINATE_BACKEND (p. 566) function.

Examples

To cancel a currently running query, first retrieve the process ID for the query that you want to cancel. To determine the process IDs for all currently running queries, execute the following command:

```
select pid, trim(starttime) as start, duration, trim(user_name) as user, substring (query,1,40) as querytxt
from stv_recents
where status = 'Running';
```

<table>
<thead>
<tr>
<th>pid</th>
<th>starttime</th>
<th>duration</th>
<th>user</th>
<th>querytxt</th>
</tr>
</thead>
<tbody>
<tr>
<td>802</td>
<td>2013-10-14 09:19:03.55</td>
<td>132</td>
<td>dwuser</td>
<td>select venuename from venue</td>
</tr>
<tr>
<td>834</td>
<td>2013-10-14 08:33:49.47</td>
<td>1250414</td>
<td>dwuser</td>
<td>select * from listing;</td>
</tr>
<tr>
<td>964</td>
<td>2013-10-14 08:30:43.29</td>
<td>326179</td>
<td>dwuser</td>
<td>select sellerid from sales</td>
</tr>
</tbody>
</table>

The following statement cancels the query with process ID 802:

```
select pg_cancel_backend(802);
```

PG_TERMINATE_BACKEND

Terminates a session. You can terminate a session owned by your user. A superuser can terminate any session.
Synopsis

```sql
pg_terminate_backend( pid )
```

Arguments

`pid`

The process ID of the session to be terminated. Requires an integer value.

Return type

None

Usage Notes

If you are close to reaching the limit for concurrent connections, use `PG_TERMINATE_BACKEND` to terminate idle sessions and free up the connections. For more information, see Limits in Amazon Redshift.

If queries in multiple sessions hold locks on the same table, you can use `PG_TERMINATE_BACKEND` to terminate one of the sessions, which forces any currently running transactions in the terminated session to release all locks and roll back the transaction. Query the STV_LOCKS system table to view currently held locks.

If a query is not in a transaction block (BEGIN ... END), you can cancel the query by using the CANCEL (p. 272) command or the `PGCANCEL_BACKEND` (p. 566) function.

Examples

The following statement queries the STV_LOCKS table to view all locks in effect for current transactions:

```sql
select table_id, last_update, lock_owner, lock_owner_pid, lock_status
from stv_locks;
```

```
<table>
<thead>
<tr>
<th>table_id</th>
<th>last_update</th>
<th>lock_owner</th>
<th>lock_owner_pid</th>
<th>lock_status</th>
</tr>
</thead>
<tbody>
<tr>
<td>100295</td>
<td>2014-01-06 23:50:56.290917</td>
<td>95402</td>
<td>7723</td>
<td>Holding</td>
</tr>
<tr>
<td>100304</td>
<td>2014-01-06 23:50:57.408457</td>
<td>95402</td>
<td>7723</td>
<td>Holding</td>
</tr>
<tr>
<td>100304</td>
<td>2014-01-06 23:50:57.409986</td>
<td>95402</td>
<td>7723</td>
<td>Holding</td>
</tr>
</tbody>
</table>
```

The following statement terminates the session holding the locks:

```sql
select pg_terminate_backend(7723);
```

SET_CONFIG

Sets a configuration parameter to a new setting.

This function is equivalent to the SET command in SQL.
Synopsis

```python
set_config('parameter', 'new_value', is_local)
```

Arguments

- **parameter**
  - Parameter to set.
- **new_value**
  - New value of the parameter.
- **is_local**
  - If true, parameter value applies only to the current transaction. Valid values are `true` or `1` and `false` or `0`.

Return type

Returns a CHAR or VARCHAR string.

Examples

The following query sets the value of the `query_group` parameter to `test` for the current transaction only:

```sql
select set_config('query_group', 'test', true);
```

```
set_config
-----------------
test
(1 row)
```

System information functions

Topics

- CURRENT_DATABASE (p. 569)
- CURRENT_SCHEMA (p. 569)
- CURRENT_SCHEMAS (p. 570)
- CURRENT_USER (p. 570)
- CURRENT_USER_ID (p. 571)
- HAS_DATABASE_PRIVILEGE (p. 571)
- HAS_SCHEMA_PRIVILEGE (p. 572)
- HAS_TABLE_PRIVILEGE (p. 573)
- PG_BACKEND_PID() (p. 573)
- PG_LAST_COPY_COUNT() (p. 574)
- PG_LAST_COPY_ID() (p. 575)
- PG_LAST_QUERY_ID() (p. 576)
- SESSION_USER (p. 577)
- SLICE_NUM function (p. 577)
- USER (p. 578)
- VERSION() (p. 578)
Amazon Redshift supports numerous system information functions.

**CURRENT_DATABASE**

Returns the name of the database where you are currently connected.

**Synopsis**

```sql
current_database()
```

**Return type**

Returns a CHAR or VARCHAR string.

**Example**

The following query returns the name of the current database:

```sql
select current_database();
```

<table>
<thead>
<tr>
<th>current_database</th>
</tr>
</thead>
<tbody>
<tr>
<td>tickit</td>
</tr>
<tr>
<td>(1 row)</td>
</tr>
</tbody>
</table>

**CURRENT_SCHEMA**

Returns the name of the schema at the front of the search path. This schema will be used for any tables or other named objects that are created without specifying a target schema.

**Synopsis**

**Note**

This is a leader-node function. This function returns an error if it references a user-created table, an STL or STV system table, or an SVV or SVL system view.

```sql
current_schema()
```

**Return type**

CURRENT_SCHEMA returns a CHAR or VARCHAR string.

**Examples**

The following query returns the current schema:

```sql
select current_schema();
```

<table>
<thead>
<tr>
<th>current_schema</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
</tr>
<tr>
<td>(1 row)</td>
</tr>
</tbody>
</table>
CURRENT_SCHEMAS

Returns an array of the names of any schemas in the current search path. The current search path is defined in the search_path parameter.

Synopsis

Note
This is a leader-node function. This function returns an error if it references a user-created table, an STL or STV system table, or an SVV or SVL system view.

```
current_schemas(include_implicit)
```

Argument

include_implicit
If true, specifies that the search path should include any implicitly included system schemas. Valid values are true and false. Typically, if true, this parameter returns the pg_catalog schema in addition to the current schema.

Return type

Returns a CHAR or VARCHAR string.

Examples

The following example returns the names of the schemas in the current search path, not including implicitly included system schemas:

```
select current_schemas(false);
```

```
current_schemas
-----------------
{public}
(1 row)
```

The following example returns the names of the schemas in the current search path, including implicitly included system schemas:

```
select current_schemas(true);
```

```
current_schemas
---------------------
{pg_catalog,public}
(1 row)
```

CURRENT_USER

Returns the user name of the current “effective” user of the database, as applicable to checking permissions. Usually, this user name will be the same as the session user; however, this can occasionally be changed by superusers.

Note
Do not use trailing parentheses when calling CURRENT_USER.
Synopsis

`current_user`

Return type

CURRENT_USER returns a CHAR or VARCHAR string.

Example

The following query returns the name of the current database user:

```
select current_user;
```

```
current_user
--------------
dwuser
   (1 row)
```

CURRENT_USER_ID

Returns the unique identifier for the Amazon Redshift user logged in to the current session.

Synopsis

```
CURRENT_USER_ID
```

Return type

The CURRENT_USER_ID function returns an integer.

Examples

The following example returns the user name and current user ID for this session:

```
select user, current_user_id;
```

```
current_user | current_user_id
--------------|------------------
dwuser       |               1
   (1 row)
```

HAS_DATABASE_PRIVILEGE

Returns `true` if the user has the specified privilege for the specified database.

Synopsis

Note

This is a leader-node function. This function returns an error if it references a user-created table, an STL or STV system table, or an SVV or SVL system view.
has_database_privilege( [ user, ] database, privilege)

Arguments

user
Name of the user to check the database privileges for. Default is to check for the current user.

database
Database associated with the privilege.

privilege
Privilege to check.

Return type
Returns a CHAR or VARCHAR string.

Example

The following query confirms that the GUEST user has the TEMP privilege on the TICKIT database:

```sql
select has_database_privilege('guest', 'tickit', 'temp');
```

<table>
<thead>
<tr>
<th>has_database_privilege</th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
</tr>
</tbody>
</table>

HAS_SCHEMA_PRIVILEGE

Returns true if the user has the specified privilege for the specified schema.

Synopsis

Note
This is a leader-node function. This function returns an error if it references a user-created table, an STL or STV system table, or an SVV or SVL system view.

has_schema_privilege( [ user, ] schema, privilege)

Arguments

user
Name of the user to check the schema privileges for. Default is to check for the current user.

schema
Schema associated with the privilege.

privilege
Privilege to check.

Return type
Returns a CHAR or VARCHAR string.
Example

The following query confirms that the GUEST user has the CREATE privilege on the PUBLIC schema:

```sql
select has_schema_privilege('guest', 'public', 'create');
```

```
has_schema_privilege
----------------------
t
(1 row)
```

**HAS_TABLE_PRIVILEGE**

Returns `true` if the user has the specified privilege for the specified table.

**Synopsis**

**Note**
This is a leader-node function. This function returns an error if it references a user-created table, an STL or STV system table, or an SVV or SVL system view.

```sql
has_table_privilege( [ user, ] table, privilege)
```

**Arguments**

- **user**
  - Name of the user to check the table privileges for. The default is to check for the current user.
- **table**
  - Table associated with the privilege.
- **privilege**
  - Privilege to check.

**Return type**

Returns a CHAR or VARCHAR string.

**Examples**

The following query finds that the GUEST user does not have SELECT privilege on the LISTING table:

```sql
select has_table_privilege('guest', 'listing', 'select');
```

```
has_table_privilege
---------------------
f
(1 row)
```

**PG_BACKEND_PID()**

Returns the process ID (PID) of the server process handling the current session.

**Note**
The PID is not globally unique. It can be reused over time.
**Synopsis**

`pg_backend_pid()`

**Return type**

Returns an integer.

**Example**

You can correlate `PG_BACKEND_PID()` with log tables to retrieve information for the current session. For example, the following query returns the query ID and a portion of the query text for queries executed in the current session.

```sql
select query, substring(text,1,40)
from stl_querytext
where pid =  PG_BACKEND_PID()
order by query desc;
```

<table>
<thead>
<tr>
<th>query</th>
<th>substring</th>
</tr>
</thead>
<tbody>
<tr>
<td>14831</td>
<td>select query, substring(text,1,40) from</td>
</tr>
<tr>
<td>14827</td>
<td>select query, substring(path,0,80) as pa</td>
</tr>
<tr>
<td>14826</td>
<td>copy category from 's3://dw-ticket/manif</td>
</tr>
<tr>
<td>14825</td>
<td>Count rows in target table</td>
</tr>
<tr>
<td>14824</td>
<td>unload ('select * from category') to 's3</td>
</tr>
</tbody>
</table>

You can correlate `PG_BACKEND_PID()` with the pid column in the following log tables (exceptions are noted in parentheses):

- STL_CONNECTION_LOG (p. 591)
- STL_DDLTEXT (p. 591)
- STL_ERROR (p. 596)
- STL_QUERY (p. 620)
- STL_QUERYTEXT (p. 622)
- STL_SESSIONS (p. 631) (process)
- STL_TR_CONFLICT (p. 635)
- STL_UTILITYTEXT (p. 640)
- STL_WARNING (p. 643)
- STV_ACTIVE_CURSORS (p. 649)
- STV_INFLIGHT (p. 654)
- STV_LOCKS (p. 657) (lock_owner_pid)
- STV_RECENTS (p. 659) (process_id)

**PG_LAST_COPY_COUNT()**

Returns the number of rows that were loaded by the last COPY command executed in the current session. `PG_LAST_COPY_COUNT()` is updated with the query ID of the last COPY that began the load process, even if the load failed. If no COPY commands were executed in the current session, or if the last COPY failed during loading, `PG_LAST_COPY_COUNT` returns 0. For more information, see `PG_LAST_COPY_ID()` (p. 575). If the last COPY failed, `PG_LAST_COPY_COUNT` returns 0.
Synopsis

pg_last_copy_count()

Return type

Returns BIGINT.

Example

The following query returns the number of rows loaded by the latest COPY command in the current session.

```
select pg_last_copy_count();
pg_last_copy_count
-------------------
       192497
(1 row)
```

PG_LAST_COPY_ID()

Returns the query ID of the most recently executed COPY command in the current session. If no COPY commands have been executed in the current session, PG_LAST_COPY_ID returns -1. The value for PG_LAST_COPY_ID is updated when the COPY command begins the load process. For example, if the COPY fails because of a syntax error or because of insufficient privileges, the COPY ID is not updated. If the COPY fails because of invalid load data, the COPY ID is updated, so you can use PG_LAST_COPY_ID to query STL_LOAD_ERRORS table. If the COPY transaction is rolled back, the COPY ID is not set back.

Synopsis

pg_last_copy_id()

Return type

Returns an integer.

Example

The following query returns the query ID of the latest COPY command in the current session.

```
select pg_last_copy_id();
pg_last_copy_id
--------------
        5437
(1 row)
```

The following query joins STL_LOAD_ERRORS to STL_LOADERROR_DETAIL to view the details errors that occurred during the most recent load in the current session:
select d.query, substring(d.filename,14,20),
    d.line_number as line,
    substring(d.value,1,16) as value,
    substring(le.err_reason,1,48) as err_reason
from stl_loaderror_detail d, stl_load_errors le
where d.query = le.query
and d.query = pg_last_copy_id();

<table>
<thead>
<tr>
<th>query</th>
<th>substring</th>
<th>line</th>
<th>value</th>
<th>err_reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>558</td>
<td>allusers_pipe.txt</td>
<td>251</td>
<td>251</td>
<td>String contains invalid or unsupported UTF8 code</td>
</tr>
<tr>
<td>558</td>
<td>allusers_pipe.txt</td>
<td>251</td>
<td>ZRU29FGR</td>
<td>String contains invalid or unsupported UTF8 code</td>
</tr>
<tr>
<td>558</td>
<td>allusers_pipe.txt</td>
<td>251</td>
<td>Kaitlin</td>
<td>String contains invalid or unsupported UTF8 code</td>
</tr>
<tr>
<td>558</td>
<td>allusers_pipe.txt</td>
<td>251</td>
<td>Walter</td>
<td>String contains invalid or unsupported UTF8 code</td>
</tr>
</tbody>
</table>

**PG_LAST_QUERY_ID()**

Returns the query ID of the most recently executed query in the current session. If no queries have been executed in the current session, `PG_LAST_QUERY_ID` returns -1. `PG_LAST_QUERY_ID` does not return the query ID for queries that execute exclusively on the leader node. For more information, see Leader-node only functions (p. 414).

**Synopsis**

`pg_last_query_id()`

**Return type**

Returns an integer.

**Example**

The following query returns the ID of the latest query executed in the current session.

```sql
select pg_last_query_id();
```

```
pg_last_query_id
----------------
5437
(1 row)
```

The following query returns the query ID and text of the most recently executed query in the current session.

```sql
select query, trim(querytxt) as sqlquery
from stl_query
where query = pg_last_query_id();
```

<table>
<thead>
<tr>
<th>query</th>
<th>sqlquery</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
SESSION_USER

Returns the name of the user associated with the current session. This is the user who initiated the current
database connection.

**Note**
Do not use trailing parentheses when calling SESSION_USER.

**Synopsis**

```
session_user
```

**Return type**

Returns a CHAR or VARCHAR string.

**Example**

The following example returns the current session user:

```
select session_user;
```

```
session_user
----------------
dwuser
(1 row)
```

SLICE_NUM function

Returns an integer corresponding to the slice number in the cluster where the data for a row is located.
SLICE_NUM takes no parameters.

**Syntax**

```
SLICE_NUM()
```

**Return type**

The SLICE_NUM function returns an integer.

**Examples**

The following example shows which slices contain data for the first ten EVENT rows in the EVENTS table:

```
select distinct eventid, slice_num() from event order by eventid limit 10;
eventid | slice_num
---------+----------
1        | 1
2        | 2
```
The following example returns a code (1012) to show that a query without a FROM statement executes on the leader node:

```sql
select slice_num();
slice_num
----------
1012
(1 row)
```

**USER**

Synonym for CURRENT_USER. See [CURRENT_USER](#) (p. 570).

**VERSION()**

The VERSION() function returns details about the currently installed release, with specific Amazon Redshift version information at the end.

**Note**

This is a leader-node function. This function returns an error if it references a user-created table, an STL or STV system table, or an SVV or SVL system view.

**Synopsis**

```
VERSION()
```

**Return type**

Returns a CHAR or VARCHAR string.

**Reserved words**

The following is a list of Amazon Redshift reserved words. You can use the reserved words with delimited identifiers (double quotes).

For more information, see [Names and identifiers](#) (p. 213).

<table>
<thead>
<tr>
<th>Reserved word</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES128</td>
</tr>
<tr>
<td>AES256</td>
</tr>
<tr>
<td>ALL</td>
</tr>
<tr>
<td>ALLOWOVERWRITE</td>
</tr>
<tr>
<td>ANALYSE</td>
</tr>
</tbody>
</table>

---

**Amazon Redshift Database Developer Guide**

API Version 2012-12-01

578
ANALYZE
AND
ANY
ARRAY
AS
ASC
AUTHORIZATION
BACKUP
BETWEEN
BINARY
BLANKSASNULL
BOTH
BYTEDICT
CASE
CAST
CHECK
COLLATE
COLUMN
CONSTRAINT
CREATE
CREDENTIALS
CROSS
CURRENT_DATE
CURRENT_TIME
CURRENT_TIMESTAMP
CURRENT_USER
CURRENT_USER_ID
DEFAULT
DEFERRABLE
DEFLATE
DEFRAG
DELTA
DELTA32K
DESC
DISABLE
DISTINCT
DO
ELSE
EMPTYASNULL
ENABLE
ENCODE
ENCRYPT
ENCRYPTION
END
EXCEPT
EXPLICIT
FALSE
FOR
FOREIGN
FREEZE
FROM
FULL
GLOBALDICT256
GLOBALDICT64K
GRANT
GROUP
GZIP
HAVING
IDENTITY
IGNORE
ILIKE
IN
INITIALLY
INNER
INTERSECT
INTO
IS
ISNULL
JOIN
LEADING
LEFT
LIKE
LIMIT
LOCALTIME
LOCALTIMESTAMP
LUN
LUNS
LZO
LZOP
MINUS
MOSTLY13
MOSTLY32
MOSTLY8
NATURAL
NEW
NOT
NOTNULL
NULL
NULLS
OFF
OFFLINE
OFFSET
OLD
ON
ONLY
OPEN
OR
ORDER
OUTER
OVERLAPS
PARALLEL
PARTITION
PERCENT
PLACING
PRIMARY
RAW
READRATIO
RECOVER
REFERENCES
REJECTLOG
RESORT
RESTORE
SELECT
SESSION_USER
SIMILAR
SOME
SYSDATE
SYSTEM
TABLE
TAG
TDES
TEXT255
TEXT32K
THEN
TO
TOP
TRAILING
TRUE
TRUNCATECOLUMNS
UNION
UNIQUE
USER
USING
VERBOSE
WALLET
WHEN
WHERE
WITH
WITHOUT
System Tables Reference

Topics
- System tables and views (p. 582)
- Types of system tables and views (p. 583)
- Visibility of data in system tables and views (p. 583)
- STL tables for logging (p. 584)
- STV tables for snapshot data (p. 649)
- System views (p. 671)
- System catalog tables (p. 687)

System tables and views

Amazon Redshift has many system tables and views that contain information about how the system is functioning. You can query these system tables and views the same way that you would query any other database tables. This section shows some sample system table queries and explains:

- How different types of system tables and views are generated
- What types of information you can obtain from these tables
- How to join Amazon Redshift system tables to catalog tables
- How to manage the growth of system table log files

Some system tables can only be used by AWS staff for diagnostic purposes. The following sections discuss the system tables that can be queried for useful information by system administrators or other database users.

Note
System tables are not included in automated or manual cluster backups (snapshots). STL log tables only retain approximately two to five days of log history, depending on log usage and available disk space. If you want to retain the log data, you will need to periodically copy it to other tables or unload it to Amazon S3.
Types of system tables and views

There are two types of system tables: STL and STV tables.

STL tables are generated from logs that have been persisted to disk to provide a history of the system. STV tables are virtual tables that contain snapshots of the current system data. They are based on transient in-memory data and are not persisted to disk-based logs or regular tables. System views that contain any reference to a transient STV table are called SVV views. Views containing only references to STL tables are called SVL views.

System tables and views do not use the same consistency model as regular tables. It is important to be aware of this issue when querying them, especially for STV tables and SVV views. For example, given a regular table t1 with a column c1, you would expect that the following query to return no rows:

```
select * from t1
where c1 > (select max(c1) from t1)
```

However, the following query against a system table might well return rows:

```
select * from stv_exec_state
where currenttime > (select max(currenttime) from stv_exec_state)
```

The reason this query might return rows is that currenttime is transient and the two references in the query might not return the same value when evaluated.

On the other hand, the following query might well return no rows:

```
select * from stv_exec_state
where currenttime = (select max(currenttime) from stv_exec_state)
```

Visibility of data in system tables and views

There are two classes of visibility for data in system tables and views: user visible and superuser visible.

Only users with superuser privileges can see the data in those tables that are in the superuser visible category. Regular users can see data in the user visible tables, but only those rows that were generated by their own activities. Rows generated by another user are invisible to a regular user. A superuser can see all rows in both categories of tables.

Filtering system-generated queries

The query-related system tables and views, such as SVL_QUERY_SUMMARY, SVL_QLOG, and others, usually contain a large number of automatically generated statements that Amazon Redshift uses to monitor the status of the database. These system-generated queries are visible to a superuser, but are seldom useful. To filter them out when selecting from a system table or system view that uses the userid column, add the condition `userid > 1` to the WHERE clause. For example:

```
select * from svl_query_summary where userid > 1
```
STL tables for logging

STL system tables are generated from Amazon Redshift log files to provide a history of the system. These files reside on every node in the data warehouse cluster. The STL tables take the information from the logs and format them into usable tables for system administrators.

To manage disk space, the STL log tables only retain approximately two to five days of log history, depending on log usage and available disk space. If you want to retain the log data, you will need to periodically copy it to other tables or unload it to Amazon S3.

Topics

- STL_AGGR (p. 585)
- STL_ALERT_EVENT_LOG (p. 587)
- STL_BCAST (p. 588)
- STL_COMMIT_STATS (p. 590)
- STL_CONNECTION_LOG (p. 591)
- STL_DDLTEXT (p. 591)
- STL_DIST (p. 593)
- STL_DELETE (p. 594)
- STL_ERROR (p. 596)
- STL_EXPLAIN (p. 597)
- STL_FILE_SCAN (p. 599)
- STL_HASH (p. 600)
- STL_HASHJOIN (p. 602)
- STL_INSERT (p. 603)
- STL_LIMIT (p. 604)
- STL_LOAD_COMMITS (p. 606)
- STL_LOAD_ERRORS (p. 608)
- STL_LOADERROR_DETAIL (p. 610)
- STL_MERGE (p. 612)
- STL_MERGEJOIN (p. 613)
- STL_NESTLOOP (p. 614)
- STL_PARSE (p. 615)
- STL_PLAN_INFO (p. 616)
- STL_PROJECT (p. 618)
- STL_QUERY (p. 620)
- STL_QUERYTEXT (p. 622)
- STL_REPLACEMENTS (p. 623)
- STL_RETURN (p. 624)
- STL_SAVE (p. 625)
- STL_S3CLIENT (p. 626)
- STL_S3CLIENT_ERROR (p. 628)
- STL_SCAN (p. 629)
- STL_SESSIONS (p. 631)
- STL_SORT (p. 632)
- STL_SSHCLIENT_ERROR (p. 633)
- STL_STREAM_SEGS (p. 634)
STL_AGGR

Analyzes aggregate execution steps for queries. These steps occur during execution of aggregate functions and GROUP BY clauses.

This table is visible to all users.

Table columns

<table>
<thead>
<tr>
<th>Column name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>userid</td>
<td>integer</td>
<td>ID of the user who generated the entry.</td>
</tr>
<tr>
<td>query</td>
<td>integer</td>
<td>Query ID. The query column can be used to join other system tables and views.</td>
</tr>
<tr>
<td>slice</td>
<td>integer</td>
<td>Number that identifies the slice where the query was running.</td>
</tr>
<tr>
<td>segment</td>
<td>integer</td>
<td>Number that identifies the query segment.</td>
</tr>
<tr>
<td>step</td>
<td>integer</td>
<td>Query step that executed.</td>
</tr>
<tr>
<td>starttime</td>
<td>timestamp</td>
<td>Time in UTC that the query started executing, with 6 digits of precision for fractional seconds. For example: 2009-06-12 11:29:19.131358.</td>
</tr>
<tr>
<td>endtime</td>
<td>timestamp</td>
<td>Time in UTC that the query finished executing, with 6 digits of precision for fractional seconds. For example: 2009-06-12 11:29:19.131358.</td>
</tr>
<tr>
<td>tasknum</td>
<td>integer</td>
<td>Number of the query task process that was assigned to execute the step.</td>
</tr>
<tr>
<td>rows</td>
<td>bigint</td>
<td>Total number of rows that were processed.</td>
</tr>
<tr>
<td>bytes</td>
<td>bigint</td>
<td>Size, in bytes, of all the output rows for the step.</td>
</tr>
<tr>
<td>slots</td>
<td>integer</td>
<td>Number of hash buckets.</td>
</tr>
<tr>
<td>occupied</td>
<td>integer</td>
<td>Number of slots that contain records.</td>
</tr>
<tr>
<td>maxlength</td>
<td>integer</td>
<td>Size of the largest slot.</td>
</tr>
</tbody>
</table>
### Sample queries

Returns information about aggregate execution steps for SLICE 1 and TBL 239.

```sql
select query, segment, bytes, slots, occupied, maxlen, is_diskbased, workmem, type
from stl_aggr where slice=1 and tbl=239
order by rows
limit 10;
```

<table>
<thead>
<tr>
<th>query</th>
<th>segment</th>
<th>bytes</th>
<th>slots</th>
<th>occupied</th>
<th>maxlen</th>
<th>is_diskbased</th>
<th>workmem</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>562</td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>f</td>
<td></td>
<td>HASHED</td>
</tr>
<tr>
<td></td>
<td>383385600</td>
<td></td>
<td>4194304</td>
<td>0</td>
<td>0</td>
<td>f</td>
<td>0</td>
<td>HASHED</td>
</tr>
<tr>
<td></td>
<td>616</td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>f</td>
<td>0</td>
<td>HASHED</td>
</tr>
<tr>
<td></td>
<td>383385600</td>
<td></td>
<td>4194304</td>
<td>0</td>
<td>0</td>
<td>f</td>
<td>0</td>
<td>HASHED</td>
</tr>
<tr>
<td></td>
<td>546</td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>f</td>
<td>0</td>
<td>HASHED</td>
</tr>
<tr>
<td></td>
<td>383385600</td>
<td></td>
<td>4194304</td>
<td>0</td>
<td>0</td>
<td>f</td>
<td>0</td>
<td>HASHED</td>
</tr>
<tr>
<td></td>
<td>547</td>
<td></td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>f</td>
<td>0</td>
<td>PLAIN</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>685</td>
<td></td>
<td>1</td>
<td>32</td>
<td>1</td>
<td>f</td>
<td>0</td>
<td>HASHED</td>
</tr>
<tr>
<td></td>
<td>383385600</td>
<td></td>
<td>4194304</td>
<td>1</td>
<td>0</td>
<td>f</td>
<td>0</td>
<td>HASHED</td>
</tr>
<tr>
<td></td>
<td>652</td>
<td></td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>f</td>
<td>0</td>
<td>PLAIN</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>680</td>
<td></td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>f</td>
<td>0</td>
<td>PLAIN</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>658</td>
<td></td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>f</td>
<td>0</td>
<td>PLAIN</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>686</td>
<td></td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>f</td>
<td>0</td>
<td>PLAIN</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>695</td>
<td></td>
<td>32</td>
<td>4194304</td>
<td>1</td>
<td>f</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
```
STL_ALERT_EVENT_LOG

Records an alert when the query optimizer identifies conditions that might indicate performance issues. Use the STL_ALERT_EVENT_LOG table to identify opportunities to improve query performance.

A query consists of multiple segments, and each segment consists of one or more steps. For more information, see Query execution steps (p. 12).

STL_ALERT_EVENT_LOG is user visible.

Table columns

<table>
<thead>
<tr>
<th>Column name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>userid</td>
<td>integer</td>
<td>ID of the user who generated the entry.</td>
</tr>
<tr>
<td>query</td>
<td>integer</td>
<td>Query ID. The query column can be used to join other system tables and views.</td>
</tr>
<tr>
<td>slice</td>
<td>integer</td>
<td>Number that identifies the slice where the query was running.</td>
</tr>
<tr>
<td>segment</td>
<td>integer</td>
<td>Number that identifies the query segment.</td>
</tr>
<tr>
<td>step</td>
<td>integer</td>
<td>Query step that executed.</td>
</tr>
<tr>
<td>pid</td>
<td>integer</td>
<td>Process ID associated with the statement.</td>
</tr>
<tr>
<td>xid</td>
<td>bigint</td>
<td>Transaction ID associated with the statement.</td>
</tr>
<tr>
<td>event</td>
<td>character(1024)</td>
<td>Description of the alert event.</td>
</tr>
<tr>
<td>solution</td>
<td>character(1024)</td>
<td>Recommended solution.</td>
</tr>
<tr>
<td>event_time</td>
<td>timestamp</td>
<td>Time in UTC that the query started executing, with 6 digits of precision for fractional seconds. For example: 2009-06-12 11:29:19.131358.</td>
</tr>
</tbody>
</table>

Usage notes

You can use the STL_ALERT_EVENT_LOG to identify potential issues in your queries, then follow the practices in Tuning Query Performance (p. 180) Tuning Query Performance to optimize your database design and rewrite your queries. STL_ALERT_EVENT_LOG records the following alerts:

- **Missing statistics**

  Statistics are missing. Run ANALYZE following data loads or significant updates and use STATUPDATE with COPY operations. For more information, see Maintain up-to-date table statistics (p. 34).

- **Nested Loop**

  A nested loop is usually a Cartesian product. Evaluate your query to ensure that all participating tables are joined efficiently.

- **Very Selective Filter**
The ratio of rows scanned to rows returned is less than 0.05 and the number of rows returned is greater than 2000. Rows scanned is the value of `rows_pre_user_filter` and rows returned is the value of rows in the `STL_SCAN` (p. 629) system table. This can be caused by missing or incorrect sort keys. For more information, see Choosing sort keys (p. 119).

**Excessive Ghost Rows**

A scan skipped a relatively large number of rows that are marked as deleted but not vacuumed, or rows that have been inserted but not committed. For more information, see Vacuuming tables (p. 161).

**Large Distribution**

More than 1,000,000 rows were redistributed for hash join or aggregation. For more information, see Choosing a data distribution style (p. 101).

**Large Broadcast**

More than 1,000,000 rows were broadcast for hash join. For more information, see Choosing a data distribution style (p. 101).

**Serial execution**

A `DS_DIST_ALL_INNER` redistribution style was indicated in the Explain plan, which forces serial execution because the entire inner table was redistributed to a single node. For more information, see Choosing a data distribution style (p. 101).

### Sample queries

The following query shows alert events for four queries.

```sql
SELECT query, substring(event,0,25) as event, substring(solution,0,25) as solution, trim(event_time) as event_time from stl_alert_event_log order by query;
```

<table>
<thead>
<tr>
<th>query</th>
<th>event_time</th>
<th>event</th>
<th>solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>6567</td>
<td>2014-01-03 18:20:58</td>
<td>Missing query planner statistic</td>
<td>Run the ANALYZE command</td>
</tr>
<tr>
<td>7450</td>
<td>2014-01-03 21:19:31</td>
<td>Scanned a large number of del</td>
<td>Run the VACUUM command to rec</td>
</tr>
<tr>
<td>8406</td>
<td>2014-01-04 00:34:22</td>
<td>Nested Loop Join in the query</td>
<td>Review the join predicates to</td>
</tr>
<tr>
<td>29512</td>
<td>2014-01-06 22:00:00</td>
<td>Very selective query filter:r</td>
<td>Review the choice of sort key</td>
</tr>
</tbody>
</table>

(4 rows)

### STL_BCAST

Logs information about network activity during execution of query steps that broadcast data. Network traffic is captured by numbers of rows, bytes, and packets that are sent over the network during a given step on a given slice. The duration of the step is the difference between the logged start and end times.

To identify broadcast steps in a query, look for bcast labels in the QUERY_SUMMARY view or run the EXPLAIN command and then look for step attributes that include bcast.

This table is visible to all users.
Table columns

<table>
<thead>
<tr>
<th>Column name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>userid</td>
<td>integer</td>
<td>ID of the user who generated the entry.</td>
</tr>
<tr>
<td>query</td>
<td>integer</td>
<td>Query ID. The query column can be used to join other system tables and views.</td>
</tr>
<tr>
<td>slice</td>
<td>integer</td>
<td>Number that identifies the slice where the query was running.</td>
</tr>
<tr>
<td>segment</td>
<td>integer</td>
<td>Number that identifies the query segment.</td>
</tr>
<tr>
<td>step</td>
<td>integer</td>
<td>Query step that executed.</td>
</tr>
<tr>
<td>starttime</td>
<td>timestamp</td>
<td>Time in UTC that the query started executing, with 6 digits of precision for fractional seconds. For example: 2009-06-12 11:29:19.131358.</td>
</tr>
<tr>
<td>endtime</td>
<td>timestamp</td>
<td>Time in UTC that the query finished executing, with 6 digits of precision for fractional seconds. For example: 2009-06-12 11:29:19.131358.</td>
</tr>
<tr>
<td>tasknum</td>
<td>integer</td>
<td>Number of the query task process that was assigned to execute the step.</td>
</tr>
<tr>
<td>rows</td>
<td>bigint</td>
<td>Total number of rows that were processed.</td>
</tr>
<tr>
<td>bytes</td>
<td>bigint</td>
<td>Size, in bytes, of all the output rows for the step.</td>
</tr>
<tr>
<td>packets</td>
<td>integer</td>
<td>Total number of packets sent over the network.</td>
</tr>
</tbody>
</table>

Sample queries

The following example returns broadcast information for the queries where there are one or more packets, and the difference between the start and end of the query was one second or more.

```sql
select query, slice, step, rows, bytes, packets, datediff(seconds, starttime, endtime)
from stl_bcast
where packets>0 and datediff(seconds, starttime, endtime)>0;
```

<table>
<thead>
<tr>
<th>query</th>
<th>slice</th>
<th>step</th>
<th>rows</th>
<th>bytes</th>
<th>packets</th>
<th>date_diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>453</td>
<td>2</td>
<td>5</td>
<td>1</td>
<td>264</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>798</td>
<td>2</td>
<td>5</td>
<td>1</td>
<td>264</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1408</td>
<td>2</td>
<td>5</td>
<td>1</td>
<td>264</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2993</td>
<td>0</td>
<td>5</td>
<td>1</td>
<td>264</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5045</td>
<td>3</td>
<td>5</td>
<td>1</td>
<td>264</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>8073</td>
<td>3</td>
<td>5</td>
<td>1</td>
<td>264</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>8163</td>
<td>3</td>
<td>5</td>
<td>1</td>
<td>264</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>9212</td>
<td>1</td>
<td>5</td>
<td>1</td>
<td>264</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>9873</td>
<td>1</td>
<td>5</td>
<td>1</td>
<td>264</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

(9 rows)
STL_COMMIT_STATS

Provides metrics related to commit performance, including the timing of the various stages of commit and the number of blocks committed. Query STL_COMMIT_STATS to determine what portion of a transaction was spent on commit and how much queuing is occurring.

This table is visible to superusers.

**Table columns**

<table>
<thead>
<tr>
<th>Column name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xid</td>
<td>integer</td>
<td>Transaction id being committed.</td>
</tr>
<tr>
<td>node</td>
<td>integer</td>
<td>Node number. -1 is the leader node.</td>
</tr>
<tr>
<td>startqueue</td>
<td>timestamp</td>
<td>Start of queueing for commit.</td>
</tr>
<tr>
<td>startwork</td>
<td>timestamp</td>
<td>Start of commit.</td>
</tr>
<tr>
<td>endflush</td>
<td>timestamp</td>
<td>End of dirty block flush phase.</td>
</tr>
<tr>
<td>endstage</td>
<td>timestamp</td>
<td>End of metadata staging phase.</td>
</tr>
<tr>
<td>endlocal</td>
<td>timestamp</td>
<td>End of local commit phase.</td>
</tr>
<tr>
<td>startglobal</td>
<td>timestamp</td>
<td>Start of global phase.</td>
</tr>
<tr>
<td>endtime</td>
<td>timestamp</td>
<td>End of the commit.</td>
</tr>
<tr>
<td>queuelen</td>
<td>integer</td>
<td>Number of transactions that were ahead of this transaction in the commit queue.</td>
</tr>
<tr>
<td>newblocks</td>
<td>integer</td>
<td>Number of new permanent blocks at the time of this commit.</td>
</tr>
<tr>
<td>dirtyblocks</td>
<td>integer</td>
<td>Number of blocks that had to be written as part of this commit.</td>
</tr>
<tr>
<td>headers</td>
<td>integer</td>
<td>Number of block headers that had to be written as part of this commit.</td>
</tr>
<tr>
<td>numxids</td>
<td>int</td>
<td>The number of active DML transactions.</td>
</tr>
<tr>
<td>oldestxid</td>
<td>long</td>
<td>The XID of the oldest active DML transaction.</td>
</tr>
</tbody>
</table>

**Sample query**

```sql
select node, datediff(ms, startqueue, startwork) as queue_time, datediff(ms, startwork, endtime) as commit_time, queuelen from stl_commit_stats where xid = 2574 order by node;
```

<table>
<thead>
<tr>
<th>node</th>
<th>queue_time</th>
<th>commit_time</th>
<th>queuelen</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>0</td>
<td>617</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>444950725641</td>
<td>616</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>444950725636</td>
<td>616</td>
<td>0</td>
</tr>
</tbody>
</table>
STL_CONNECTION_LOG

Logs authentication attempts and connections and disconnections.

This table is visible to superusers.

Table columns

<table>
<thead>
<tr>
<th>Column name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>event</td>
<td>character(50)</td>
<td>Connection or authentication event.</td>
</tr>
<tr>
<td>recordtime</td>
<td>timestamp</td>
<td>Time the event occurred.</td>
</tr>
<tr>
<td>remotehost</td>
<td>character(32)</td>
<td>Name or IP address of remote host.</td>
</tr>
<tr>
<td>remoteport</td>
<td>character(32)</td>
<td>Port number for remote host.</td>
</tr>
<tr>
<td>pid</td>
<td>integer</td>
<td>Process ID associated with the statement.</td>
</tr>
<tr>
<td>dbname</td>
<td>character(50)</td>
<td>Database name.</td>
</tr>
<tr>
<td>username</td>
<td>character(50)</td>
<td>User name.</td>
</tr>
<tr>
<td>authmethod</td>
<td>character(32)</td>
<td>Authentication method.</td>
</tr>
<tr>
<td>duration</td>
<td>integer</td>
<td>Duration of connection in microseconds.</td>
</tr>
</tbody>
</table>

Sample queries

This example reflects a failed authentication attempt and a successful connection and disconnection.

```
select event, recordtime, remotehost, username
from stl_connection_log order by recordtime;
```

<table>
<thead>
<tr>
<th>event</th>
<th>recordtime</th>
<th>remotehost</th>
<th>username</th>
</tr>
</thead>
<tbody>
<tr>
<td>authentication failure</td>
<td>2012-10-25 14:41:56.96391</td>
<td>10.49.42.138</td>
<td>john</td>
</tr>
<tr>
<td>authenticated</td>
<td>2012-10-25 14:42:10.87613</td>
<td>10.49.42.138</td>
<td>john</td>
</tr>
<tr>
<td>initiating session</td>
<td>2012-10-25 14:42:10.87638</td>
<td>10.49.42.138</td>
<td>john</td>
</tr>
<tr>
<td>disconnecting session</td>
<td>2012-10-25 14:42:19.95992</td>
<td>10.49.42.138</td>
<td>john</td>
</tr>
</tbody>
</table>

(4 rows)

STL_DDLTEXT

Captures the following DDL statements that were run on the system.

These DDL statements include the following queries and objects:
CREATE SCHEMA, TABLE, VIEW
DROP SCHEMA, TABLE, VIEW
ALTER SCHEMA, TABLE

See also STL_QUERYTEXT (p. 622), STL/utilityTEXT (p. 640), and SVL_STATEMENTTEXT (p. 683). These tables provide a timeline of the SQL commands that are executed on the system; this history is useful for troubleshooting purposes and for creating an audit trail of all system activities.

Use the STARTTIME and ENDTIME columns to find out which statements were logged during a given time period. Long blocks of SQL text are broken into lines 200 characters long; the SEQUENCE column identifies fragments of text that belong to a single statement.

This table is visible to all users.

### Table columns

<table>
<thead>
<tr>
<th>Column name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>userid</td>
<td>integer</td>
<td>ID of the user who generated the entry.</td>
</tr>
<tr>
<td>xid</td>
<td>bigint</td>
<td>Transaction ID associated with the statement.</td>
</tr>
<tr>
<td>pid</td>
<td>integer</td>
<td>Process ID associated with the statement.</td>
</tr>
<tr>
<td>label</td>
<td>character(30)</td>
<td>Either the name of the file used to run the query or a label defined with a SET QUERY_GROUP command. If the query is not file-based or the QUERY_GROUP parameter is not set, this field is blank.</td>
</tr>
<tr>
<td>starttime</td>
<td>timestamp</td>
<td>Time in UTC that the query started executing, with 6 digits of precision for fractional seconds. For example: 2009-06-12 11:29:19.131358.</td>
</tr>
<tr>
<td>endtime</td>
<td>timestamp</td>
<td>Time in UTC that the query finished executing, with 6 digits of precision for fractional seconds. For example: 2009-06-12 11:29:19.131358.</td>
</tr>
<tr>
<td>sequence</td>
<td>integer</td>
<td>When a single statement contains more than 200 characters, additional rows are logged for that statement. Sequence 0 is the first row, 1 is the second, and so on.</td>
</tr>
<tr>
<td>text</td>
<td>character(200)</td>
<td>SQL text, in 200-character increments.</td>
</tr>
</tbody>
</table>

### Sample queries

The following query shows the DDL for four CREATE TABLE statements. The DDL text column is truncated for readability.

```sql
select xid, starttime, sequence, substring(text,1,40) as text
from stl_ddltext order by xid desc, sequence;
```

<table>
<thead>
<tr>
<th>xid</th>
<th>starttime</th>
<th>sequence</th>
<th>text</th>
</tr>
</thead>
<tbody>
<tr>
<td>1806</td>
<td>2013-10-23 00:11:14.709851</td>
<td>0</td>
<td>CREATE TABLE supplier {</td>
</tr>
</tbody>
</table>
STL_DIST

Logs information about network activity during execution of query steps that distribute data. Network traffic is captured by numbers of rows, bytes, and packets that are sent over the network during a given step on a given slice. The duration of the step is the difference between the logged start and end times.

To identify distribution steps in a query, look for dist labels in the QUERY_SUMMARY view or run the EXPLAIN command and then look for step attributes that include dist.

This table is visible to all users.

Table columns

<table>
<thead>
<tr>
<th>Column name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>userid</td>
<td>integer</td>
<td>ID of the user who generated the entry.</td>
</tr>
<tr>
<td>query</td>
<td>integer</td>
<td>Query ID. The query column can be used to join other system tables and views.</td>
</tr>
<tr>
<td>slice</td>
<td>integer</td>
<td>Number that identifies the slice where the query was running.</td>
</tr>
<tr>
<td>segment</td>
<td>integer</td>
<td>Number that identifies the query segment.</td>
</tr>
<tr>
<td>step</td>
<td>integer</td>
<td>Query step that executed.</td>
</tr>
<tr>
<td>starttime</td>
<td>timestamp</td>
<td>Time in UTC that the query started executing, with 6 digits of precision for fractional seconds. For example: 2009-06-12 11:29:19.131358.</td>
</tr>
<tr>
<td>endtime</td>
<td>timestamp</td>
<td>Time in UTC that the query finished executing, with 6 digits of precision for fractional seconds. For example: 2009-06-12 11:29:19.131358.</td>
</tr>
<tr>
<td>tasknum</td>
<td>integer</td>
<td>Number of the query task process that was assigned to execute the step.</td>
</tr>
<tr>
<td>rows</td>
<td>bigint</td>
<td>Total number of rows that were processed.</td>
</tr>
<tr>
<td>bytes</td>
<td>bigint</td>
<td>Size, in bytes, of all the output rows for the step.</td>
</tr>
<tr>
<td>packets</td>
<td>integer</td>
<td>Total number of packets sent over the network.</td>
</tr>
</tbody>
</table>
Sample queries

The following example returns distribution information for queries with one or more packets and duration greater than zero.

```
select query, slice, step, rows, bytes, packets, 
datediff(seconds, starttime, endtime) as duration 
from stl_dist 
where packets>0 and datediff(seconds, starttime, endtime)>0 
order by query 
limit 10;
```

<table>
<thead>
<tr>
<th>query</th>
<th>slice</th>
<th>step</th>
<th>rows</th>
<th>bytes</th>
<th>packets</th>
<th>duration</th>
</tr>
</thead>
<tbody>
<tr>
<td>567</td>
<td>1</td>
<td>4</td>
<td>49990</td>
<td>6249564</td>
<td>707</td>
<td>1</td>
</tr>
<tr>
<td>630</td>
<td>0</td>
<td>5</td>
<td>8798</td>
<td>408404</td>
<td>46</td>
<td>2</td>
</tr>
<tr>
<td>645</td>
<td>1</td>
<td>4</td>
<td>8798</td>
<td>408404</td>
<td>46</td>
<td>1</td>
</tr>
<tr>
<td>651</td>
<td>1</td>
<td>5</td>
<td>192497</td>
<td>9226320</td>
<td>1039</td>
<td>6</td>
</tr>
<tr>
<td>669</td>
<td>1</td>
<td>4</td>
<td>192497</td>
<td>9226320</td>
<td>1039</td>
<td>4</td>
</tr>
<tr>
<td>675</td>
<td>1</td>
<td>5</td>
<td>3766</td>
<td>194656</td>
<td>22</td>
<td>1</td>
</tr>
<tr>
<td>696</td>
<td>0</td>
<td>4</td>
<td>3766</td>
<td>194656</td>
<td>22</td>
<td>1</td>
</tr>
<tr>
<td>705</td>
<td>0</td>
<td>4</td>
<td>930</td>
<td>44400</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>111525</td>
<td>0</td>
<td>3</td>
<td>68</td>
<td>17408</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

(9 rows)

STL_DELETE

Analyzes delete execution steps for queries.

This table is visible to all users.

Table columns

<table>
<thead>
<tr>
<th>Column name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>userid</td>
<td>integer</td>
<td>ID of the user who generated the entry.</td>
</tr>
<tr>
<td>query</td>
<td>integer</td>
<td>Query ID. The query column can be used to join other system tables and views.</td>
</tr>
<tr>
<td>slice</td>
<td>integer</td>
<td>Number that identifies the slice where the query was running.</td>
</tr>
<tr>
<td>segment</td>
<td>integer</td>
<td>Number that identifies the query segment.</td>
</tr>
<tr>
<td>step</td>
<td>integer</td>
<td>Query step that executed.</td>
</tr>
<tr>
<td>starttime</td>
<td>timestamp</td>
<td>Time in UTC that the query started executing, with 6 digits of precision for fractional seconds. For example: 2009-06-12 11:29:19.131358.</td>
</tr>
<tr>
<td>endtime</td>
<td>timestamp</td>
<td>Time in UTC that the query finished executing, with 6 digits of precision for fractional seconds. For example: 2009-06-12 11:29:19.131358.</td>
</tr>
</tbody>
</table>
### Sample queries

In order to create a row in STL_DELETE, the following example inserts a row into the EVENT table and then deletes it.

First, insert a row into the EVENT table and verify that it was inserted.

```sql
insert into event(eventid,venueid,catid,dateid,eventname)
values ((select max(eventid)+1 from event),95,9,1857,'Lollapalooza');
```

```sql
select * from event
where eventname='Lollapalooza'
order by eventid;
```

<table>
<thead>
<tr>
<th>eventid</th>
<th>venueid</th>
<th>catid</th>
<th>dateid</th>
<th>eventname</th>
<th>starttime</th>
</tr>
</thead>
<tbody>
<tr>
<td>4274</td>
<td>102</td>
<td>9</td>
<td>1965</td>
<td>Lollapalooza</td>
<td>2008-05-01 19:00:00</td>
</tr>
<tr>
<td>4684</td>
<td>114</td>
<td>9</td>
<td>2105</td>
<td>Lollapalooza</td>
<td>2008-10-06 14:00:00</td>
</tr>
<tr>
<td>5673</td>
<td>128</td>
<td>9</td>
<td>1973</td>
<td>Lollapalooza</td>
<td>2008-05-01 15:00:00</td>
</tr>
<tr>
<td>5740</td>
<td>51</td>
<td>9</td>
<td>1933</td>
<td>Lollapalooza</td>
<td>2008-04-17 15:00:00</td>
</tr>
<tr>
<td>5856</td>
<td>119</td>
<td>9</td>
<td>1831</td>
<td>Lollapalooza</td>
<td>2008-01-05 14:00:00</td>
</tr>
<tr>
<td>6040</td>
<td>126</td>
<td>9</td>
<td>2145</td>
<td>Lollapalooza</td>
<td>2008-11-15 15:00:00</td>
</tr>
<tr>
<td>7972</td>
<td>92</td>
<td>9</td>
<td>2026</td>
<td>Lollapalooza</td>
<td>2008-07-19 19:30:00</td>
</tr>
<tr>
<td>8046</td>
<td>65</td>
<td>9</td>
<td>1840</td>
<td>Lollapalooza</td>
<td>2008-01-14 15:00:00</td>
</tr>
<tr>
<td>8518</td>
<td>48</td>
<td>9</td>
<td>1904</td>
<td>Lollapalooza</td>
<td>2008-03-19 15:00:00</td>
</tr>
<tr>
<td>8799</td>
<td>95</td>
<td>9</td>
<td>1857</td>
<td>Lollapalooza</td>
<td></td>
</tr>
</tbody>
</table>

(10 rows)

Now, delete the row that you added to the EVENT table and verify that it was deleted.

```sql
delete from event
where eventname='Lollapalooza' and eventid=(select max(eventid) from event);
```

```sql
select * from event
where eventname='Lollapalooza'
order by eventid;
```

<table>
<thead>
<tr>
<th>eventid</th>
<th>venueid</th>
<th>catid</th>
<th>dateid</th>
<th>eventname</th>
<th>starttime</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Then query stl_delete to see the execution steps for the deletion. In this example, the query returned over 300 rows, so the output below is shortened for display purposes.

```sql
select query, slice, segment, step, tasknum, rows, tbl from stl_delete order by query;
```

### STL_ERROR

Records all errors that occur while running queries.

Use the STL_ERROR table to investigate exceptions and signals that occurred in Amazon Redshift. Although queries display error messages when they fail, you can use this table to investigate past errors or for situations where a user did not capture the query error information.

This table is visible to all users.

#### Table columns

<table>
<thead>
<tr>
<th>Column name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>userid</td>
<td>integer</td>
<td>ID of the user who generated the entry.</td>
</tr>
<tr>
<td>Column name</td>
<td>Data type</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>-----------</td>
<td>-------------</td>
</tr>
<tr>
<td>process</td>
<td>character(12)</td>
<td>Process that threw the exception.</td>
</tr>
<tr>
<td>recordtime</td>
<td>timestamp</td>
<td>Time that the error occurred.</td>
</tr>
<tr>
<td>pid</td>
<td>integer</td>
<td>Process ID. The STL_QUERY (p. 620) table contains process IDs and unique query IDs for executed queries.</td>
</tr>
<tr>
<td>errcode</td>
<td>integer</td>
<td>Error code corresponding to the error category.</td>
</tr>
<tr>
<td>file</td>
<td>character(90)</td>
<td>Name of the source file where the error occurred.</td>
</tr>
<tr>
<td>linenum</td>
<td>integer</td>
<td>Line number in the source file where the error occurred.</td>
</tr>
<tr>
<td>context</td>
<td>character(100)</td>
<td>Cause of the error.</td>
</tr>
<tr>
<td>error</td>
<td>character(512)</td>
<td>Error message.</td>
</tr>
</tbody>
</table>

**Sample queries**

The following example retrieves the error information for queries during the current session:

```sql
select process, errcode, file, linenum as line, trim(error) as err
from stl_error
where pid=pg_backend_pid();
```

This query returns the following example output:

```
<table>
<thead>
<tr>
<th>process</th>
<th>errcode</th>
<th>file</th>
<th>line</th>
<th>err</th>
</tr>
</thead>
<tbody>
<tr>
<td>diskman2</td>
<td>9</td>
<td>fdisk.cpp</td>
<td>1599</td>
<td>aio_error():Bad file descriptor</td>
</tr>
<tr>
<td>diskman2</td>
<td>1001</td>
<td>fdisk.cpp</td>
<td>135</td>
<td>'false' - need to handle redshift exceptions:Not implemented</td>
</tr>
<tr>
<td>diskman2</td>
<td>1001</td>
<td>resman.cpp</td>
<td>128</td>
<td>'false' - need to handle redshift exceptions:Not implemented</td>
</tr>
<tr>
<td>diskman2</td>
<td>1008</td>
<td>shutdown.cpp</td>
<td>111</td>
<td>uncaught exception from an unknown source</td>
</tr>
</tbody>
</table>
```

(4 rows)

**STL_EXPLAIN**

Displays the EXPLAIN plan for a query that has been submitted for execution.

This table is visible to all users.

**Table columns**

<table>
<thead>
<tr>
<th>Column name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>userid</td>
<td>integer</td>
<td>ID of the user who generated the entry.</td>
</tr>
<tr>
<td>Column name</td>
<td>Data type</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>-------------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>query</td>
<td>integer</td>
<td>Query ID. The query column can be used to join other system tables and views.</td>
</tr>
<tr>
<td>nodeid</td>
<td>integer</td>
<td>Plan node identifier, where a node maps to one or more steps in the execution of the query.</td>
</tr>
<tr>
<td>parentid</td>
<td>integer</td>
<td>Plan node identifier for a parent node. A parent node has some number of child nodes. For example, a merge join is the parent of the scans on the joined tables.</td>
</tr>
<tr>
<td>plannode</td>
<td>character(400)</td>
<td>The node text from the EXPLAIN output. Plan nodes that refer to execution on compute nodes are prefixed with XN in the EXPLAIN output.</td>
</tr>
<tr>
<td>info</td>
<td>character(400)</td>
<td>Qualifier and filter information for the plan node. For example, join conditions and WHERE clause restrictions are included in this column.</td>
</tr>
</tbody>
</table>

### Sample queries

Consider the following EXPLAIN output for an aggregate join query:

```sql
explain select avg(datediff(day, listtime, saletime)) as avgwait
from sales, listing where sales.listid = listing.listid;
QUERY PLAN

| XN Aggregate  (cost=6350.30..6350.31 rows=1 width=16) |
|---------------|-----------------|
|   -> XN Hash Join DS_DIST_NONE  (cost=47.08..6340.89 rows=3766 width=16) |
|       Hash Cond: ("outer".listid = "inner".listid) |
|       -> XN Seq Scan on listing  (cost=0.00..1924.97 rows=192497 width=12) |
|       -> XN Hash  (cost=37.66..37.66 rows=3766 width=12) |
|       -> XN Seq Scan on sales  (cost=0.00..37.66 rows=3766 width=12) |
```

If you run this query and its query ID is 10, you can use the STL_EXPLAIN table to see the same kind of information that the EXPLAIN command returns:

```sql
select query,nodeid,parentid,substring(plannode from 1 for 30),
substring(info from 1 for 20) from stl_explain
where query=10 order by 1,2;

<table>
<thead>
<tr>
<th>query</th>
<th>nodeid</th>
<th>parentid</th>
<th>substring</th>
<th>substring</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1</td>
<td>0</td>
<td>XN Aggregate  (cost=6717.61..6</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>2</td>
<td>1</td>
<td></td>
<td>-&gt; XN Merge Join DS_DIST_NO Merge Cond:&quot;outer&quot;</td>
</tr>
<tr>
<td>10</td>
<td>3</td>
<td>2</td>
<td></td>
<td>-&gt; XN Seq Scan on lis</td>
</tr>
<tr>
<td>10</td>
<td>4</td>
<td>2</td>
<td></td>
<td>-&gt; XN Seq Scan on sal</td>
</tr>
</tbody>
</table>
```

Consider the following query:
select event.eventid, sum(pricepaid) 
from event, sales 
where event.eventid=sales.eventid 
group by event.eventid order by 2 desc;

<table>
<thead>
<tr>
<th>eventid</th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>289</td>
<td>51846.00</td>
</tr>
<tr>
<td>7895</td>
<td>51049.00</td>
</tr>
<tr>
<td>1602</td>
<td>50301.00</td>
</tr>
<tr>
<td>851</td>
<td>49956.00</td>
</tr>
<tr>
<td>7315</td>
<td>49823.00</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

If this query's ID is 15, the following system table query returns the plan nodes that were executed. In this case, the order of the nodes is reversed to show the actual order of execution:

select query,nodeid,parentid,substring(plannode from 1 for 56) 
from stl_explain where query=15 order by 1, 2 desc;

<table>
<thead>
<tr>
<th>query</th>
<th>nodeid</th>
<th>parentid</th>
<th>substring</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>8</td>
<td>7</td>
<td>-&gt; XN Seq Scan on event</td>
</tr>
<tr>
<td>15</td>
<td>7</td>
<td>5</td>
<td>-&gt; XN Hash(cost=87.98..87.9)</td>
</tr>
<tr>
<td>15</td>
<td>6</td>
<td>5</td>
<td>-&gt; XN Seq Scan on sales(cost)</td>
</tr>
<tr>
<td>15</td>
<td>5</td>
<td>4</td>
<td>-&gt; XN Hash Join DS_DIST_OUTER(cost)</td>
</tr>
<tr>
<td>15</td>
<td>4</td>
<td>3</td>
<td>-&gt; XN HashAggregate(cost=862286577.07..</td>
</tr>
<tr>
<td>15</td>
<td>3</td>
<td>2</td>
<td>-&gt; XN Sort(cost=1000862287175.47..1000862287175.47..1000862287197.</td>
</tr>
<tr>
<td>15</td>
<td>2</td>
<td>1</td>
<td>-&gt; XN Network(cost=1000862287175.47..1000862287197.</td>
</tr>
<tr>
<td>15</td>
<td>1</td>
<td>0</td>
<td>XN Merge(cost=1000862287175.47..1000862287197.46 rows=87</td>
</tr>
</tbody>
</table>

(8 rows)

The following query retrieves the query IDs for any query plans that contain a window function:

select query, trim(plannode) from stl_explain 
where plannode like '%Window%';

<table>
<thead>
<tr>
<th>query</th>
<th>btrim</th>
</tr>
</thead>
<tbody>
<tr>
<td>26</td>
<td>-&gt; XN Window(cost=1000985348268.57..1000985351256.98 rows=170 width=33)</td>
</tr>
<tr>
<td>27</td>
<td>-&gt; XN Window(cost=1000985348268.57..1000985351256.98 rows=170 width=33)</td>
</tr>
</tbody>
</table>

(2 rows)

**STL_FILE_SCAN**

Returns the files that Amazon Redshift read while loading data via the COPY command.

Querying this table can help troubleshoot data load errors. STL_FILE_SCAN can be particularly helpful with pinpointing issues in parallel data loads because parallel data loads typically load many files with a single COPY command.

This table is visible to all users.
Table columns

<table>
<thead>
<tr>
<th>Column name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>userid</td>
<td>integer</td>
<td>ID of the user who generated the entry.</td>
</tr>
<tr>
<td>query</td>
<td>integer</td>
<td>Query ID. The query column can be used to join other system tables and views.</td>
</tr>
<tr>
<td>slice</td>
<td>integer</td>
<td>Number that identifies the slice where the query was running.</td>
</tr>
<tr>
<td>name</td>
<td>character(90)</td>
<td>Full path and name of the file that was loaded.</td>
</tr>
<tr>
<td>lines</td>
<td>bigint</td>
<td>Number of lines read from the file.</td>
</tr>
<tr>
<td>bytes</td>
<td>bigint</td>
<td>Number of bytes read from the file.</td>
</tr>
<tr>
<td>loadtime</td>
<td>bigint</td>
<td>Amount of time spent loading the file (in microseconds).</td>
</tr>
<tr>
<td>curtime</td>
<td>bigint</td>
<td>Timestamp representing the time that Amazon Redshift started processing the file.</td>
</tr>
</tbody>
</table>

Sample queries

The following query retrieves the names and load times of any files that took over 1000000 microseconds for Amazon Redshift to read:

```sql
select trim(name) as name, loadtime from stl_file_scan
where loadtime > 1000000;
```

This query returns the following example output:

<table>
<thead>
<tr>
<th>name</th>
<th>loadtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>listings_pipe.txt</td>
<td>9458354</td>
</tr>
<tr>
<td>allusers_pipe.txt</td>
<td>2963761</td>
</tr>
<tr>
<td>allevents_pipe.txt</td>
<td>1409135</td>
</tr>
<tr>
<td>tickit/listings_pipe.txt</td>
<td>7071087</td>
</tr>
<tr>
<td>tickit/allevents_pipe.txt</td>
<td>1237364</td>
</tr>
<tr>
<td>tickit/allusers_pipe.txt</td>
<td>2535138</td>
</tr>
<tr>
<td>listings_pipe.txt</td>
<td>6706370</td>
</tr>
<tr>
<td>allusers_pipe.txt</td>
<td>3579461</td>
</tr>
<tr>
<td>allevents_pipe.txt</td>
<td>1313195</td>
</tr>
<tr>
<td>tickit/allusers_pipe.txt</td>
<td>3236060</td>
</tr>
<tr>
<td>tickit/listings_pipe.txt</td>
<td>4980108</td>
</tr>
</tbody>
</table>

(11 rows)

STL_HASH

Analyzes hash execution steps for queries.

This table is visible to all users.
### Table columns

<table>
<thead>
<tr>
<th>Column name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>userid</td>
<td>integer</td>
<td>ID of the user who generated the entry.</td>
</tr>
<tr>
<td>query</td>
<td>integer</td>
<td>Query ID. The query column can be used to join other system tables and views.</td>
</tr>
<tr>
<td>slice</td>
<td>integer</td>
<td>Number that identifies the slice where the query was running.</td>
</tr>
<tr>
<td>segment</td>
<td>integer</td>
<td>Number that identifies the query segment.</td>
</tr>
<tr>
<td>step</td>
<td>integer</td>
<td>Query step that executed.</td>
</tr>
<tr>
<td>starttime</td>
<td>timestamp</td>
<td>Time in UTC that the query started executing, with 6 digits of precision for fractional seconds. For example: 2009-06-12 11:29:19.131358.</td>
</tr>
<tr>
<td>endtime</td>
<td>timestamp</td>
<td>Time in UTC that the query finished executing, with 6 digits of precision for fractional seconds. For example: 2009-06-12 11:29:19.131358.</td>
</tr>
<tr>
<td>tasknum</td>
<td>integer</td>
<td>Number of the query task process that was assigned to execute the step.</td>
</tr>
<tr>
<td>rows</td>
<td>bigint</td>
<td>Total number of rows that were processed.</td>
</tr>
<tr>
<td>bytes</td>
<td>bigint</td>
<td>Size, in bytes, of all the output rows for the step.</td>
</tr>
<tr>
<td>slots</td>
<td>integer</td>
<td>Total number of hash buckets.</td>
</tr>
<tr>
<td>occupied</td>
<td>integer</td>
<td>Total number of slots that contain records.</td>
</tr>
<tr>
<td>maxlength</td>
<td>integer</td>
<td>Size of the largest slot.</td>
</tr>
<tr>
<td>tbl</td>
<td>integer</td>
<td>Table ID.</td>
</tr>
<tr>
<td>is_diskbased</td>
<td>character(1)</td>
<td>If true (t), the query was executed as a disk-based operation. If false (f), the query was executed in memory.</td>
</tr>
<tr>
<td>workmem</td>
<td>bigint</td>
<td>Total number of bytes of working memory assigned to the step.</td>
</tr>
<tr>
<td>num_parts</td>
<td>integer</td>
<td>Total number of partitions that a hash table was divided into during a hash step. A hash table is partitioned when it is estimated that the entire hash table might not fit into memory.</td>
</tr>
<tr>
<td>est_rows</td>
<td>bigint</td>
<td>Estimated number of rows to be hashed.</td>
</tr>
<tr>
<td>num_blocks_pert</td>
<td>integer</td>
<td>This information is for internal use only.</td>
</tr>
<tr>
<td>row_dist_variance</td>
<td>integer</td>
<td>This information is for internal use only.</td>
</tr>
</tbody>
</table>

### Sample queries

The following example returns information about the number of partitions that were used in a hash for query 720, and indicates that none of the steps ran on disk.
STL_HASHJOIN

Analyzes hash join execution steps for queries.

This table is visible to all users.

Table columns

<table>
<thead>
<tr>
<th>Column name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>userid</td>
<td>integer</td>
<td>ID of the user who generated the entry.</td>
</tr>
<tr>
<td>query</td>
<td>integer</td>
<td>Query ID. The query column can be used to join other system tables and views.</td>
</tr>
<tr>
<td>slice</td>
<td>integer</td>
<td>Number that identifies the slice where the query was running.</td>
</tr>
<tr>
<td>segment</td>
<td>integer</td>
<td>Number that identifies the query segment.</td>
</tr>
<tr>
<td>step</td>
<td>integer</td>
<td>Query step that executed.</td>
</tr>
<tr>
<td>starttime</td>
<td>timestamp</td>
<td>Time in UTC that the query started executing, with 6 digits of precision for fractional seconds. For example: 2009-06-12 11:29:19.131358.</td>
</tr>
<tr>
<td>endpoint</td>
<td>timestamp</td>
<td>Time in UTC that the query finished executing, with 6 digits of precision for fractional seconds. For example: 2009-06-12 11:29:19.131358.</td>
</tr>
<tr>
<td>tasknum</td>
<td>integer</td>
<td>Number of the query task process that was assigned to execute the step.</td>
</tr>
<tr>
<td>rows</td>
<td>bigint</td>
<td>Total number of rows that were processed.</td>
</tr>
<tr>
<td>tbl</td>
<td>integer</td>
<td>Table ID.</td>
</tr>
<tr>
<td>num_parts</td>
<td>integer</td>
<td>Total number of partitions that a hash table was divided into during a hash step. A hash table is partitioned when it is estimated that the entire hash table might not fit into memory.</td>
</tr>
</tbody>
</table>
### Column name | Data type | Description
---|---|---
join_type | integer | The type of join for the step:
• 0. The query used an inner join.
• 1. The query used a left outer join.
• 2. The query used a full outer join.
• 3. The query used a right outer join.
• 4. The query used a UNION operator.
• 5. The query used an IN condition.
• 6. This information is for internal use only.
• 7. This information is for internal use only.
• 8. This information is for internal use only.
• 9. This information is for internal use only.
• 10. This information is for internal use only.
• 11. This information is for internal use only.
• 12. This information is for internal use only.

hash_looped | character(1) | This information is for internal use only.
switched_parts | character(1) | Indicates whether the build (or outer) and probe (or inner) sides have switched.

### Sample queries

The following example returns the number of partitions used in a hash join for query 720.

```sql
select query, slice, tbl, num_parts
from stl_hashjoin
where query=720 limit 10;
```

<table>
<thead>
<tr>
<th>query</th>
<th>slice</th>
<th>tbl</th>
<th>num_parts</th>
</tr>
</thead>
<tbody>
<tr>
<td>720</td>
<td>0</td>
<td>243</td>
<td>1</td>
</tr>
<tr>
<td>720</td>
<td>1</td>
<td>243</td>
<td>1</td>
</tr>
</tbody>
</table>

(2 rows)

### STL_INSERT

Analyzes insert execution steps for queries.

This table is visible to all users.
Table columns

<table>
<thead>
<tr>
<th>Column name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>userid</td>
<td>integer</td>
<td>ID of the user who generated the entry.</td>
</tr>
<tr>
<td>query</td>
<td>integer</td>
<td>Query ID. The query column can be used to join other system tables and views.</td>
</tr>
<tr>
<td>slice</td>
<td>integer</td>
<td>Number that identifies the slice where the query was running.</td>
</tr>
<tr>
<td>segment</td>
<td>integer</td>
<td>Number that identifies the query segment.</td>
</tr>
<tr>
<td>step</td>
<td>integer</td>
<td>Query step that executed.</td>
</tr>
<tr>
<td>starttime</td>
<td>timestamp</td>
<td>Time in UTC that the query started executing, with 6 digits of precision for fractional seconds. For example: 2009-06-12 11:29:19.131358.</td>
</tr>
<tr>
<td>endtime</td>
<td>timestamp</td>
<td>Time in UTC that the query finished executing, with 6 digits of precision for fractional seconds. For example: 2009-06-12 11:29:19.131358.</td>
</tr>
<tr>
<td>tasknum</td>
<td>integer</td>
<td>Number of the query task process that was assigned to execute the step.</td>
</tr>
<tr>
<td>rows</td>
<td>bigint</td>
<td>Total number of rows that were processed.</td>
</tr>
<tr>
<td>tbl</td>
<td>integer</td>
<td>Table ID.</td>
</tr>
</tbody>
</table>

Sample queries

The following example returns insert execution steps for the most recent query.

```
select slice, segment, step, tasknum, rows, tbl
from stl_insert
where query=pg_last_query_id();
```

<table>
<thead>
<tr>
<th>slice</th>
<th>segment</th>
<th>step</th>
<th>tasknum</th>
<th>rows</th>
<th>tbl</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>2</td>
<td>15</td>
<td>24958</td>
<td>100548</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td>15</td>
<td>25032</td>
<td>100548</td>
</tr>
</tbody>
</table>

(2 rows)

**STL_LIMIT**

Analyzes the execution steps that occur when a LIMIT clause is used in a SELECT query.

This table is visible to all users.
Table columns

<table>
<thead>
<tr>
<th>Column name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>userid</td>
<td>integer</td>
<td>ID of the user who generated the entry.</td>
</tr>
<tr>
<td>query</td>
<td>integer</td>
<td>Query ID. The query column can be used to join other system tables and views.</td>
</tr>
<tr>
<td>slice</td>
<td>integer</td>
<td>Number that identifies the slice where the query was running.</td>
</tr>
<tr>
<td>segment</td>
<td>integer</td>
<td>Number that identifies the query segment.</td>
</tr>
<tr>
<td>step</td>
<td>integer</td>
<td>Query step that executed.</td>
</tr>
<tr>
<td>starttime</td>
<td>timestamp</td>
<td>Time in UTC that the query started executing, with 6 digits of precision for fractional seconds. For example: 2009-06-12 11:29:19.131358.</td>
</tr>
<tr>
<td>endtime</td>
<td>timestamp</td>
<td>Time in UTC that the query finished executing, with 6 digits of precision for fractional seconds. For example: 2009-06-12 11:29:19.131358.</td>
</tr>
<tr>
<td>tasknum</td>
<td>integer</td>
<td>Number of the query task process that was assigned to execute the step.</td>
</tr>
<tr>
<td>rows</td>
<td>bigint</td>
<td>Total number of rows that were processed.</td>
</tr>
</tbody>
</table>

Sample queries

In order to generate a row in STL_LIMIT, this example first runs the following query against the VENUE table using the LIMIT clause.

```sql
select * from venue
order by 1
limit 10;
```

<table>
<thead>
<tr>
<th>venueid</th>
<th>venuename</th>
<th>venuecity</th>
<th>venuestate</th>
<th>venueseats</th>
</tr>
</thead>
<tbody>
<tr>
<td>68756</td>
<td>Toyota Park</td>
<td>Bridgeview</td>
<td>IL</td>
<td></td>
</tr>
<tr>
<td>80242</td>
<td>Columbus Crew Stadium</td>
<td>Columbus</td>
<td>OH</td>
<td></td>
</tr>
<tr>
<td></td>
<td>RFK Stadium</td>
<td>Washington</td>
<td>DC</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>CommunityAmerica Ballpark</td>
<td>Kansas City</td>
<td>KS</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Gillette Stadium</td>
<td>Foxborough</td>
<td>MA</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>New York Giants Stadium</td>
<td>East Rutherford</td>
<td>NJ</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>BMO Field</td>
<td>Toronto</td>
<td>ON</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>The Home Depot Center</td>
<td>Carson</td>
<td>CA</td>
<td></td>
</tr>
</tbody>
</table>
Next, run the following query to find the query ID of the last query you ran against the VENUE table.

```sql
select max(query) from stl_query;
```

```
max
--------
127128
(1 row)
```

Optionally, you can run the following query to verify that the query ID corresponds to the LIMIT query you previously ran.

```sql
select query, trim(querytxt) from stl_query where query=127128;
```

```
query |                  btrim
--------+------------------------------------------
127128 | select * from venue order by 1 limit 10;
(1 row)
```

Finally, run the following query to return information about the LIMIT query from the STL_LIMIT table.

```sql
select slice, segment, step, starttime, endtime, tasknum from stl_limit where query=127128 order by starttime, endtime;
```

```
slice | segment | step | starttime | endtime
-------+---------+------+-----------+----------
1 |       1 |    3 | 2013-09-06 22:56:43.608114 | 2013-09-06 22:56:43.609383
0 |       1 |    3 | 2013-09-06 22:56:43.608708 | 2013-09-06 22:56:43.609521
10000 |       2 |    2 | 2013-09-06 22:56:43.612506 | 2013-09-06 22:56:43.612668
(3 rows)
```

**STL_LOAD_COMMITS**

Returns information to track or troubleshoot a data load.
This table records the progress of each data file as it is loaded into a database table.

This table is visible to all users.

## Table columns

<table>
<thead>
<tr>
<th>Column name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>userid</td>
<td>integer</td>
<td>ID of the user who generated the entry.</td>
</tr>
<tr>
<td>query</td>
<td>integer</td>
<td>Query ID. The query column can be used to join other system tables and views.</td>
</tr>
<tr>
<td>slice</td>
<td>integer</td>
<td>Slice loaded for this entry.</td>
</tr>
<tr>
<td>name</td>
<td>character(256)</td>
<td>System-defined value.</td>
</tr>
<tr>
<td>filename</td>
<td>character(256)</td>
<td>Name of file being tracked.</td>
</tr>
<tr>
<td>byte_offset</td>
<td>integer</td>
<td>This information is for internal use only.</td>
</tr>
<tr>
<td>lines_scanned</td>
<td>integer</td>
<td>Number of lines scanned from the load file. This number may not match the number of rows that are actually loaded. For example, the load may scan but tolerate a number of bad records, based on the MAXERROR option in the COPY command.</td>
</tr>
<tr>
<td>errors</td>
<td>integer</td>
<td>This information is for internal use only.</td>
</tr>
<tr>
<td>curtime</td>
<td>timestamp</td>
<td>Time that this entry was last updated.</td>
</tr>
<tr>
<td>status</td>
<td>integer</td>
<td>This information is for internal use only.</td>
</tr>
</tbody>
</table>

## Sample queries

The following example returns details for the last COPY operation.

```
select query, trim(filename) as file, curtime as updated
from stl_load_commits
where query = pg_last_copy_id();
```

```
query |               file               |          updated
-------+----------------------------------+----------------------------
28554 | s3://dw-ticket/category_pipe.txt | 2013-11-01 17:14:52.648486
```

(1 row)

The following query contains entries for a fresh load of the tables in the TICKIT database:

```
select query, trim(filename), curtime
from stl_load_commits
where filename like '%tickit%'
order by query;
```

```
query |            btrim            |          curtime
-------+---------------------------+----------------------------
22475 | ticket/allusers_pipe.txt  | 2013-02-08 20:58:23.274186
22478 | ticket/venue_pipe.txt     | 2013-02-08 20:58:25.070604
```

(1 row)
The fact that a record is written to the log file for this system table does not mean that the load committed successfully as part of its containing transaction. To verify load commits, query the STL_UTILITYTEXT table and look for the COMMIT record that corresponds with a COPY transaction. For example, this query joins STL_LOAD_COMMITS and STL_QUERY based on a subquery against STL_UTILITYTEXT:

```sql
select l.query, rtrim(l.filename), q.xid
from stl_load_commits l, stl_query q
where l.query=q.query
and exists
(select xid from stl_utilitytext where xid=q.xid and rtrim("text")='COMMIT');
```

<table>
<thead>
<tr>
<th>query</th>
<th>rtrim</th>
<th>xid</th>
</tr>
</thead>
<tbody>
<tr>
<td>tickit/date2008_pipe.txt</td>
<td>68066</td>
<td>23365</td>
</tr>
<tr>
<td>tickit/category_pipe.txt</td>
<td>68066</td>
<td>23365</td>
</tr>
<tr>
<td>allusers_pipe.txt</td>
<td>23365</td>
<td>23365</td>
</tr>
<tr>
<td>venue_pipe.txt</td>
<td>23390</td>
<td>23390</td>
</tr>
<tr>
<td>listings_pipe.txt</td>
<td>23445</td>
<td>23445</td>
</tr>
<tr>
<td>tickit/venue_pipe.txt</td>
<td>68065</td>
<td>68065</td>
</tr>
<tr>
<td>allusers_pipe.txt</td>
<td>23365</td>
<td>23365</td>
</tr>
<tr>
<td>venue_pipe.txt</td>
<td>23390</td>
<td>23390</td>
</tr>
<tr>
<td>listings_pipe.txt</td>
<td>23445</td>
<td>23445</td>
</tr>
</tbody>
</table>

STL_LOAD_ERRORS

Displays the records of all Amazon Redshift load errors.
STL_LOAD_ERRORS contains a history of all Amazon Redshift load errors. See Load error reference (p. 149) for a comprehensive list of possible load errors and explanations.

This table is visible to all users.

**Table columns**

<table>
<thead>
<tr>
<th>Column name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>userid</td>
<td>integer</td>
<td>ID of the user who generated the entry.</td>
</tr>
<tr>
<td>slice</td>
<td>integer</td>
<td>Slice where the error occurred.</td>
</tr>
<tr>
<td>tbl</td>
<td>integer</td>
<td>Table ID.</td>
</tr>
<tr>
<td>starttime</td>
<td>timestamp</td>
<td>Start time in UTC for the load.</td>
</tr>
<tr>
<td>session</td>
<td>integer</td>
<td>Session ID for the session performing the load.</td>
</tr>
<tr>
<td>query</td>
<td>integer</td>
<td>Query ID. The query column can be used to join other system tables and views.</td>
</tr>
<tr>
<td>filename</td>
<td>character(256)</td>
<td>Complete path to the input file for the load.</td>
</tr>
<tr>
<td>line_number</td>
<td>bigint</td>
<td>Line number in the load file with the error. For COPY from JSON, the line number of the last line of the JSON object with the error.</td>
</tr>
<tr>
<td>colname</td>
<td>character(127)</td>
<td>Field with the error.</td>
</tr>
<tr>
<td>type</td>
<td>character(10)</td>
<td>Data type of the field.</td>
</tr>
<tr>
<td>col_length</td>
<td>character(10)</td>
<td>Column length, if applicable. This field is populated when the data type has a limit length. For example, for a column with a data type of “character(3)”, this column will contain the value &quot;3&quot;.</td>
</tr>
<tr>
<td>position</td>
<td>integer</td>
<td>Position of the error in the field.</td>
</tr>
<tr>
<td>raw_line</td>
<td>character(1024)</td>
<td>Raw load data that contains the error.</td>
</tr>
<tr>
<td>raw_field_value</td>
<td>char(1024)</td>
<td>The pre-parsing value for the field “colname” that lead to the parsing error.</td>
</tr>
<tr>
<td>err_code</td>
<td>integer</td>
<td>Error code.</td>
</tr>
<tr>
<td>err_reason</td>
<td>character(100)</td>
<td>Explanation for the error.</td>
</tr>
</tbody>
</table>

**Sample queries**

The following query joins STL_LOAD_ERRORS to STL_LOADERROR_DETAIL to view the details errors that occurred during the most recent load.

```sql
select d.query, substring(d.filename,14,20),
       d.line_number as line,
       substring(d.value,1,16) as value,
       substring(le.err_reason,1,48) as err_reason
from stl_loaderror_detail d, stl_load_errors le
where d.query = le.query
  and d.query = pg_last_copy_id();
```
The following example uses `STL_LOAD_ERRORS` with `STV_TBL_PERM` to create a new view, and then uses that view to determine what errors occurred while loading data into the EVENT table:

```sql
create view loadview as
(select distinct tbl, trim(name) as table_name, query, startime,
trim(filename) as input, line_number, colname, err_code,
trim(err_reason) as reason
from stl_load_errors sl, stv_tbl_perm sp
where sl.tbl = sp.id);
```

Next, the following query actually returns the last error that occurred while loading the EVENT table:

```sql
select table_name, query, line_number, colname, startime,
trim(reason) as error
from loadview
where table_name = 'event'
order by line_number limit 1;
```

The query returns the last load error that occurred for the EVENT table. If no load errors occurred, the query returns zero rows. In this example, the query returns a single error:

```
<table>
<thead>
<tr>
<th>table_name</th>
<th>query</th>
<th>line_number</th>
<th>colname</th>
<th>error</th>
<th>startime</th>
</tr>
</thead>
<tbody>
<tr>
<td>event</td>
<td>309</td>
<td>0</td>
<td>5</td>
<td>Error in Timestamp value or format [%Y-%m-%d %H:%M:%S]</td>
<td>2014-04-22 15:12:44</td>
</tr>
</tbody>
</table>
```

**STL_LOADERROR_DETAIL**

Displays a log of data parse errors that occurred while using a COPY command to load tables.

A parse error occurs when Amazon Redshift cannot parse a field in a data row while loading it into a table. For example, if a table column is expecting an integer data type and the data file contains a string of letters in that field, it causes a parse error.

Query `STL_LOADERROR_DETAIL` for additional details, such as the exact data row and column where a parse error occurred, after you query `STL_LOAD_ERRORS` to find out general information about the error.
The STL_LOADERROR_DETAIL table contains all data columns including and prior to the column where the parse error occurred. Use the VALUE field to see the data value that was actually parsed in this column, including the columns that parsed correctly up to the error.

This table is visible to all users.

### Table columns

<table>
<thead>
<tr>
<th>Column name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>userid</td>
<td>integer</td>
<td>ID of the user who generated the entry.</td>
</tr>
<tr>
<td>slice</td>
<td>integer</td>
<td>Slice where the error occurred.</td>
</tr>
<tr>
<td>session</td>
<td>integer</td>
<td>Session ID for the session performing the load.</td>
</tr>
<tr>
<td>query</td>
<td>integer</td>
<td>Query ID. The query column can be used to join other system tables and views.</td>
</tr>
<tr>
<td>filename</td>
<td>character(256)</td>
<td>Complete path to the input file for the load.</td>
</tr>
<tr>
<td>line_number</td>
<td>bigint</td>
<td>Line number in the load file with the error.</td>
</tr>
<tr>
<td>field</td>
<td>integer</td>
<td>Field with the error.</td>
</tr>
<tr>
<td>colname</td>
<td>character(1024)</td>
<td>Column name.</td>
</tr>
<tr>
<td>value</td>
<td>character(1024)</td>
<td>Parsed data value of the field. (May be truncated.)</td>
</tr>
<tr>
<td>is_null</td>
<td>integer</td>
<td>Whether or not the parsed value is null.</td>
</tr>
<tr>
<td>type</td>
<td>character(10)</td>
<td>Data type of the field.</td>
</tr>
<tr>
<td>col_length</td>
<td>character(10)</td>
<td>Column length, if applicable. This field is populated when the data type has a limit length. For example, for a column with a data type of &quot;character(3)&quot;, this column will contain the value &quot;3&quot;.</td>
</tr>
</tbody>
</table>

### Sample query

The following query joins STL_LOAD_ERRORS to STL_LOADERROR_DETAIL to view the details of a parse error that occurred while loading the EVENT table, which has a table ID of 100133:

```sql
select d.query, d.line_number, d.value, le.raw_line, le.err_reason
from stl_loaderror_detail d, stl_load_errors le
where d.query = le.query
and tbl = 100133;
```

The following sample output shows the columns that loaded successfully, including the column with the error. In this example, two columns successfully loaded before the parse error occurred in the third column, where a character string was incorrectly parsed for a field expecting an integer. Because the field expected an integer, it parsed the string "aaa", which is uninitialized data, as a null and generated a parse error. The output shows the raw value, parsed value, and error reason:
When a query joins STL_LOAD_ERRORS and STL_LOADERROR_DETAIL, it displays an error reason for each column in the data row, which simply means that an error occurred in that row. The last row in the results is the actual column where the parse error occurred.

**STL_MERGE**

Analyzes merge execution steps for queries. These steps occur when the results of parallel operations (such as sorts and joins) are merged for subsequent processing.

This table is visible to all users.

**Table columns**

<table>
<thead>
<tr>
<th>Column name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>userid</td>
<td>integer</td>
<td>ID of the user who generated the entry.</td>
</tr>
<tr>
<td>query</td>
<td>integer</td>
<td>Query ID. The query column can be used to join other system tables and views.</td>
</tr>
<tr>
<td>slice</td>
<td>integer</td>
<td>Number that identifies the slice where the query was running.</td>
</tr>
<tr>
<td>segment</td>
<td>integer</td>
<td>Number that identifies the query segment.</td>
</tr>
<tr>
<td>step</td>
<td>integer</td>
<td>Query step that executed.</td>
</tr>
<tr>
<td>starttime</td>
<td>timestamp</td>
<td>Time in UTC that the query started executing, with 6 digits of precision for fractional seconds. For example: 2009-06-12 11:29:19.131358.</td>
</tr>
<tr>
<td>endtime</td>
<td>timestamp</td>
<td>Time in UTC that the query finished executing, with 6 digits of precision for fractional seconds. For example: 2009-06-12 11:29:19.131358.</td>
</tr>
<tr>
<td>tasknum</td>
<td>integer</td>
<td>Number of the query task process that was assigned to execute the step.</td>
</tr>
<tr>
<td>rows</td>
<td>bigint</td>
<td>Total number of rows that were processed.</td>
</tr>
</tbody>
</table>

**Sample queries**

The following example returns 10 merge execution results.

```
select query, step, starttime, endtime, tasknum, rows
from stl_merge
limit 10;
```
### STL_MERGEJOIN

Analyzes merge join execution steps for queries.

This table is visible to all users.

#### Table columns

<table>
<thead>
<tr>
<th>Column name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>userid</td>
<td>integer</td>
<td>ID of the user who generated the entry.</td>
</tr>
<tr>
<td>query</td>
<td>integer</td>
<td>Query ID. The query column can be used to join other system tables and views.</td>
</tr>
<tr>
<td>slice</td>
<td>integer</td>
<td>Number that identifies the slice where the query was running.</td>
</tr>
<tr>
<td>segment</td>
<td>integer</td>
<td>Number that identifies the query segment.</td>
</tr>
<tr>
<td>step</td>
<td>integer</td>
<td>Query step that executed.</td>
</tr>
<tr>
<td>starttime</td>
<td>timestamp</td>
<td>Time in UTC that the query started executing, with 6 digits of precision for fractional seconds. For example: 2009-06-12 11:29:19.131358.</td>
</tr>
<tr>
<td>endtime</td>
<td>timestamp</td>
<td>Time in UTC that the query finished executing, with 6 digits of precision for fractional seconds. For example: 2009-06-12 11:29:19.131358.</td>
</tr>
<tr>
<td>tasknum</td>
<td>integer</td>
<td>Number of the query task process that was assigned to execute the step.</td>
</tr>
<tr>
<td>rows</td>
<td>bigint</td>
<td>Total number of rows that were processed.</td>
</tr>
<tr>
<td>tbl</td>
<td>integer</td>
<td>Table ID. This is the ID for the inner table that was used in the merge join.</td>
</tr>
</tbody>
</table>

#### Sample queries

The following example returns merge join results for the most recent query.
select sum(s.qtysold), e.eventname
from event e, listing l, sales s
where e.eventid=l.eventid
and l.listid= s.listid
group by e.eventname;

select * from stl_mergejoin where query=pg_last_query_id();

<table>
<thead>
<tr>
<th>userid</th>
<th>query</th>
<th>slice</th>
<th>segment</th>
<th>step</th>
<th>starttime</th>
<th>endtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>27399</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>2013-10-02 16:30:41</td>
<td>2013-10-02 16:30:41</td>
</tr>
<tr>
<td>100</td>
<td>27399</td>
<td>0</td>
<td>4</td>
<td>4</td>
<td>2013-10-02 16:30:41</td>
<td>2013-10-02 16:30:41</td>
</tr>
<tr>
<td>100</td>
<td>27399</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>2013-10-02 16:30:41</td>
<td>2013-10-02 16:30:41</td>
</tr>
<tr>
<td>100</td>
<td>27399</td>
<td>1</td>
<td>4</td>
<td>4</td>
<td>2013-10-02 16:30:41</td>
<td>2013-10-02 16:30:41</td>
</tr>
</tbody>
</table>

**STL_NESTLOOP**

Analyzes nested-loop join execution steps for queries.

This table is visible to all users.

**Table columns**

<table>
<thead>
<tr>
<th>Column name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>userid</td>
<td>integer</td>
<td>ID of the user who generated the entry.</td>
</tr>
<tr>
<td>query</td>
<td>integer</td>
<td>Query ID. The query column can be used to join other system tables and views.</td>
</tr>
<tr>
<td>slice</td>
<td>integer</td>
<td>Number that identifies the slice where the query was running.</td>
</tr>
<tr>
<td>segment</td>
<td>integer</td>
<td>Number that identifies the query segment.</td>
</tr>
<tr>
<td>step</td>
<td>integer</td>
<td>Query step that executed.</td>
</tr>
<tr>
<td>starttime</td>
<td>timestamp</td>
<td>Time in UTC that the query started executing, with 6 digits of precision for fractional seconds. For example: 2009-06-12 11:29:19.131358.</td>
</tr>
<tr>
<td>endtime</td>
<td>timestamp</td>
<td>Time in UTC that the query finished executing, with 6 digits of precision for fractional seconds. For example: 2009-06-12 11:29:19.131358.</td>
</tr>
<tr>
<td>tasknum</td>
<td>integer</td>
<td>Number of the query task process that was assigned to execute the step.</td>
</tr>
<tr>
<td>rows</td>
<td>bigint</td>
<td>Total number of rows that were processed.</td>
</tr>
<tr>
<td>tbl</td>
<td>integer</td>
<td>Table ID.</td>
</tr>
</tbody>
</table>
Sample queries

Because the following query neglects to join the CATEGORY table, it produces a partial Cartesian product, which is not recommended. It is shown here to illustrate a nested loop.

```sql
select count(event.eventname), event.eventname, category.catname, date.caldate
from event, category, date
where event.dateid = date.dateid
group by event.eventname, category.catname, date.caldate;
```

The following query shows the results from the previous query in the STL_NESTLOOP table.

```sql
select query, slice, segment as seg, step,
datediff(msec, starttime, endtime) as duration, tasknum, rows, tbl
from stl_nestloop
where query = pg_last_query_id();
```

<table>
<thead>
<tr>
<th>query</th>
<th>slice</th>
<th>seg</th>
<th>step</th>
<th>duration</th>
<th>tasknum</th>
<th>rows</th>
<th>tbl</th>
</tr>
</thead>
<tbody>
<tr>
<td>6028</td>
<td>0</td>
<td>4</td>
<td>5</td>
<td>41</td>
<td>22</td>
<td>24277</td>
<td>240</td>
</tr>
<tr>
<td>6028</td>
<td>1</td>
<td>4</td>
<td>5</td>
<td>26</td>
<td>23</td>
<td>24189</td>
<td>240</td>
</tr>
<tr>
<td>6028</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>25</td>
<td>23</td>
<td>24376</td>
<td>240</td>
</tr>
<tr>
<td>6028</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>54</td>
<td>22</td>
<td>23936</td>
<td>240</td>
</tr>
</tbody>
</table>

STL_PARSE

Analyzes query steps that parse strings into binary values for loading.

This table is visible to all users.

Table columns

<table>
<thead>
<tr>
<th>Column name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>userid</td>
<td>integer</td>
<td>ID of the user who generated the entry.</td>
</tr>
<tr>
<td>query</td>
<td>integer</td>
<td>Query ID. The query column can be used to join other system tables and views.</td>
</tr>
<tr>
<td>slice</td>
<td>integer</td>
<td>Number that identifies the slice where the query was running.</td>
</tr>
<tr>
<td>segment</td>
<td>integer</td>
<td>Number that identifies the query segment.</td>
</tr>
<tr>
<td>step</td>
<td>integer</td>
<td>Query step that executed.</td>
</tr>
<tr>
<td>starttime</td>
<td>timestamp</td>
<td>Time in UTC that the query started executing, with 6 digits of precision for fractional seconds. For example: 2009-06-12 11:29:19.131358.</td>
</tr>
<tr>
<td>endtime</td>
<td>timestamp</td>
<td>Time in UTC that the query finished executing, with 6 digits of precision for fractional seconds. For example: 2009-06-12 11:29:19.131358.</td>
</tr>
</tbody>
</table>
Sample queries

The following example returns all query step results for slice 1 and segment 0 where strings were parsed into binary values.

```sql
select query, step, starttime, endtime, tasknum, rows
from stl_parse
where slice=1 and segment=0;
```

<table>
<thead>
<tr>
<th>query</th>
<th>step</th>
<th>starttime</th>
<th>endtime</th>
<th>tasknum</th>
<th>rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>669</td>
<td>1</td>
<td>2013-08-12 22:35:13</td>
<td>2013-08-12 22:35:17</td>
<td>32</td>
<td>192497</td>
</tr>
<tr>
<td>696</td>
<td>1</td>
<td>2013-08-12 22:35:49</td>
<td>2013-08-12 22:35:49</td>
<td>32</td>
<td>0</td>
</tr>
<tr>
<td>525</td>
<td>1</td>
<td>2013-08-12 22:35:03</td>
<td>2013-08-12 22:35:03</td>
<td>13</td>
<td>49990</td>
</tr>
<tr>
<td>621</td>
<td>1</td>
<td>2013-08-12 22:34:03</td>
<td>2013-08-12 22:34:03</td>
<td>27</td>
<td>365</td>
</tr>
<tr>
<td>651</td>
<td>1</td>
<td>2013-08-12 22:34:47</td>
<td>2013-08-12 22:34:53</td>
<td>35</td>
<td>192497</td>
</tr>
<tr>
<td>590</td>
<td>1</td>
<td>2013-08-12 22:33:28</td>
<td>2013-08-12 22:33:28</td>
<td>19</td>
<td>0</td>
</tr>
<tr>
<td>675</td>
<td>1</td>
<td>2013-08-12 22:35:26</td>
<td>2013-08-12 22:35:27</td>
<td>38</td>
<td>3766</td>
</tr>
<tr>
<td>630</td>
<td>1</td>
<td>2013-08-12 22:34:17</td>
<td>2013-08-12 22:34:17</td>
<td>36</td>
<td>0</td>
</tr>
<tr>
<td>572</td>
<td>1</td>
<td>2013-08-12 22:33:04</td>
<td>2013-08-12 22:33:04</td>
<td>29</td>
<td>0</td>
</tr>
<tr>
<td>645</td>
<td>1</td>
<td>2013-08-12 22:34:37</td>
<td>2013-08-12 22:34:38</td>
<td>29</td>
<td>8798</td>
</tr>
<tr>
<td>604</td>
<td>1</td>
<td>2013-08-12 22:33:47</td>
<td>2013-08-12 22:33:47</td>
<td>37</td>
<td>0</td>
</tr>
</tbody>
</table>

STL_PLAN_INFO

Use the STL_PLAN_INFO table to look at the EXPLAIN output for a query in terms of a set of rows. This is an alternative way to look at query plans.

This table is visible to all users.

Table columns

<table>
<thead>
<tr>
<th>Column name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>userdid</td>
<td>integer</td>
<td>ID of the user who generated the entry.</td>
</tr>
<tr>
<td>query</td>
<td>integer</td>
<td>Query ID. The query column can be used to join other system tables and views.</td>
</tr>
<tr>
<td>nodeid</td>
<td>integer</td>
<td>Plan node identifier, where a node maps to one or more steps in the execution of the query.</td>
</tr>
<tr>
<td>Column name</td>
<td>Data type</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>-------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>segment</td>
<td>integer</td>
<td>Number that identifies the query segment.</td>
</tr>
<tr>
<td>step</td>
<td>integer</td>
<td>Number that identifies the query step.</td>
</tr>
<tr>
<td>locus</td>
<td>integer</td>
<td>Location where the step executes. 0 if on a compute node and 1 if on the leader node.</td>
</tr>
<tr>
<td>plannode</td>
<td>integer</td>
<td>Enumerated value of the plan node. See the following table for enums for plannode. (The PLANNODE column in STL_EXPLAIN (p. 597) contains the plan node text.)</td>
</tr>
<tr>
<td>startupcost</td>
<td>double precision</td>
<td>The estimated relative cost of returning the first row for this step.</td>
</tr>
<tr>
<td>totalcost</td>
<td>double precision</td>
<td>The estimated relative cost of executing the step.</td>
</tr>
<tr>
<td>rows</td>
<td>bigint</td>
<td>The estimated number of rows that will be produced by the step.</td>
</tr>
<tr>
<td>bytes</td>
<td>bigint</td>
<td>The estimated number of bytes that will be produced by the step.</td>
</tr>
</tbody>
</table>

### Sample queries

The following examples compare the query plans for a simple SELECT query returned by using the EXPLAIN command and by querying the STL_PLAN_INFO table.

```
explain select * from category;
QUERY PLAN
-------------------------------------------------------------
XN Seq Scan on category (cost=0.00..0.11 rows=11 width=49)
(1 row)
```

```
select * from category;
catid | catgroup | catname | catdesc
1     | Sports   | MLB     | Major League Baseball
3     | Sports   | NFL     | National Football League
5     | Sports   | MLS     | Major League Soccer
...
```

```
select * from stl_plan_info where query=256;
```

```
<table>
<thead>
<tr>
<th>query</th>
<th>nodeid</th>
<th>segment</th>
<th>step</th>
<th>locus</th>
<th>plannode</th>
<th>startupcost</th>
<th>totalcost</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>256</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>104</td>
<td>0</td>
<td>0.11</td>
</tr>
<tr>
<td></td>
<td>256</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>104</td>
<td>0</td>
<td>0.11</td>
</tr>
</tbody>
</table>
(2 rows)
```

In this example, PLANNODE 104 refers to the sequential scan of the CATEGORY table.

```
select distinct eventname from event order by 1;
```

```
eventname
```
explain select distinct eventname from event order by 1;

QUERY PLAN

- XN Merge (cost=1000000000136.38..1000000000137.82 rows=576 width=17)
  Merge Key: eventname
  -> XN Network (cost=1000000000136.38..1000000000137.82 rows=576 width=17)
  Send to leader
  -> XN Sort (cost=1000000000136.38..1000000000137.82 rows=576 width=17)
  Sort Key: eventname
  -> XN Unique (cost=0.00..109.98 rows=576 width=17)
  -> XN Seq Scan on event (cost=0.00..87.98 rows=8798 width=17)
(8 rows)

select * from stl_plan_info where query=240 order by nodeid desc;

| query | nodeid | segment | step | locus | plannode | startupcost | totalcost | rows | bytes |
|-------+--------+---------+------|-------+----------+------------+-----------|------|-------|
| 240   | 5      | 0       | 0    | 0     | 104       | 0          | 87.98    | 8798 | 149566|
| 240   | 5      | 0       | 1    | 0     | 104       | 0          | 87.98    | 8798 | 149566|
| 240   | 4      | 0       | 2    | 0     | 117       | 0          | 109.975  | 576  | 9792  |
| 240   | 4      | 0       | 3    | 0     | 117       | 0          | 109.975  | 576  | 9792  |
| 240   | 4      | 1       | 0    | 0     | 117       | 0          | 109.975  | 576  | 9792  |
| 240   | 4      | 1       | 1    | 0     | 117       | 0          | 109.975  | 576  | 9792  |
| 240   | 3      | 1       | 2    | 0     | 114       | 1000000000136.38 | 1000000000137.82 | 576 | 9792 |
| 240   | 3      | 2       | 0    | 0     | 114       | 1000000000136.38 | 1000000000137.82 | 576 | 9792 |
| 240   | 2      | 2       | 1    | 0     | 123       | 1000000000136.38 | 1000000000137.82 | 576 | 9792 |
| 240   | 1      | 3       | 0    | 0     | 122       | 1000000000136.38 | 1000000000137.82 | 576 | 9792 |
(10 rows)

STL_PROJECT

Contains rows for query steps that are used to evaluate expressions.

This table is visible to all users.

Table columns

<table>
<thead>
<tr>
<th>Column name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>userid</td>
<td>integer</td>
<td>ID of the user who generated the entry.</td>
</tr>
</tbody>
</table>
Sample queries

The following example returns all rows for query steps that were used to evaluate expressions for slice 0 and segment 1.

```
select query, step, starttime, endtime, tasknum, rows
from stl_project
where slice=0 and segment=1;
```

<table>
<thead>
<tr>
<th>query</th>
<th>step</th>
<th>starttime</th>
<th>endtime</th>
<th>tasknum</th>
<th>rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>719</td>
<td>1</td>
<td>2013-08-12 22:38:33</td>
<td>2013-08-12 22:38:33</td>
<td>7</td>
<td>-1</td>
</tr>
<tr>
<td>86383</td>
<td>1</td>
<td>2013-08-29 21:58:35</td>
<td>2013-08-29 21:58:35</td>
<td>7</td>
<td>-1</td>
</tr>
<tr>
<td>714</td>
<td>1</td>
<td>2013-08-12 22:38:17</td>
<td>2013-08-12 22:38:17</td>
<td>2</td>
<td>-1</td>
</tr>
<tr>
<td>86397</td>
<td>2</td>
<td>2013-08-29 22:01:20</td>
<td>2013-08-29 22:01:20</td>
<td>19</td>
<td>-1</td>
</tr>
<tr>
<td>627</td>
<td>1</td>
<td>2013-08-12 22:34:13</td>
<td>2013-08-12 22:34:13</td>
<td>34</td>
<td>-1</td>
</tr>
<tr>
<td>86326</td>
<td>2</td>
<td>2013-08-29 21:45:28</td>
<td>2013-08-29 21:45:28</td>
<td>34</td>
<td>-1</td>
</tr>
<tr>
<td>86326</td>
<td>3</td>
<td>2013-08-29 21:45:28</td>
<td>2013-08-29 21:45:28</td>
<td>34</td>
<td>-1</td>
</tr>
<tr>
<td>86371</td>
<td>1</td>
<td>2013-08-29 21:57:42</td>
<td>2013-08-29 21:57:42</td>
<td>4</td>
<td>-1</td>
</tr>
<tr>
<td>111100</td>
<td>2</td>
<td>2013-09-03 19:04:45</td>
<td>2013-09-03 19:04:45</td>
<td>12</td>
<td>-1</td>
</tr>
<tr>
<td>704</td>
<td>2</td>
<td>2013-08-12 22:36:34</td>
<td>2013-08-12 22:36:34</td>
<td>37</td>
<td>-1</td>
</tr>
<tr>
<td>649</td>
<td>2</td>
<td>2013-08-12 22:34:47</td>
<td>2013-08-12 22:34:47</td>
<td>38</td>
<td>-1</td>
</tr>
<tr>
<td>649</td>
<td>3</td>
<td>2013-08-12 22:34:47</td>
<td>2013-08-12 22:34:47</td>
<td>38</td>
<td>-1</td>
</tr>
<tr>
<td>632</td>
<td>2</td>
<td>2013-08-12 22:34:22</td>
<td>2013-08-12 22:34:22</td>
<td>13</td>
<td>-1</td>
</tr>
<tr>
<td>705</td>
<td>2</td>
<td>2013-08-12 22:36:48</td>
<td>2013-08-12 22:36:48</td>
<td>13</td>
<td>-1</td>
</tr>
<tr>
<td>705</td>
<td>3</td>
<td>2013-08-12 22:36:48</td>
<td>2013-08-12 22:36:48</td>
<td>13</td>
<td>-1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>2013-08-12 20:07:40</td>
<td>2013-08-12 20:07:40</td>
<td>3</td>
<td>-1</td>
</tr>
</tbody>
</table>
STL_QUERY

Returns execution information about a database query.

Note
The STL_QUERY and STL_QUERYTEXT tables only contain information about queries, not other utility and DDL commands. For a listing and information on all statements executed by Amazon Redshift, you can also query the STL_DDLTEXT and STL_UTILITYTEXT tables. For a complete listing of all statements executed by Amazon Redshift, you can query the SVL_STATEMENTTEXT view.

To manage disk space, the STL log tables only retain approximately two to five days of log history, depending on log usage and available disk space. If you want to retain the log data, you will need to periodically copy it to other tables or unload it to Amazon S3.

This table is visible to all users.

Table columns

<table>
<thead>
<tr>
<th>Column name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>userid</td>
<td>integer</td>
<td>ID of the user who generated the entry.</td>
</tr>
<tr>
<td>query</td>
<td>integer</td>
<td>Query ID. The query column can be used to join other system tables and views.</td>
</tr>
<tr>
<td>label</td>
<td>character(15)</td>
<td>Either the name of the file used to run the query or a label defined with a SET QUERY_GROUP command. If the query is not file-based or the QUERY_GROUP parameter is not set, this field is blank.</td>
</tr>
<tr>
<td>xid</td>
<td>bigint</td>
<td>Transaction ID.</td>
</tr>
<tr>
<td>pid</td>
<td>integer</td>
<td>Process ID. All of the queries in a session are run in the same process, so this value remains constant if you run a series of queries in the same session. You can use this column to join to the STL_ERROR (p. 596) table.</td>
</tr>
<tr>
<td>database</td>
<td>character(32)</td>
<td>The name of the database the user was connected to when the query was issued.</td>
</tr>
<tr>
<td>querytxt</td>
<td>character(4000)</td>
<td>Actual query text for the query.</td>
</tr>
<tr>
<td>starttime</td>
<td>timestamp</td>
<td>Time in UTC that the query started executing, with 6 digits of precision for fractional seconds. For example: 2009-06-12 11:29:19.131358.</td>
</tr>
</tbody>
</table>
### Sample queries

The following query lists the five most recent queries.

```sql
select query, trim(querytxt) as sqlquery
from stl_query
order by query desc limit 5;
```

<table>
<thead>
<tr>
<th>query</th>
<th>sqlquery</th>
</tr>
</thead>
<tbody>
<tr>
<td>129</td>
<td>select query, trim(querytxt) from stl_query order by query;</td>
</tr>
<tr>
<td>128</td>
<td>select node from stv_disk_read_speeds;</td>
</tr>
<tr>
<td>127</td>
<td>select system_status from stv_gui_status</td>
</tr>
<tr>
<td>126</td>
<td>select * from systable_topology order by slice</td>
</tr>
<tr>
<td>125</td>
<td>load global dict registry</td>
</tr>
</tbody>
</table>

(5 rows)

The following query returns the time elapsed in descending order for queries that ran on February 15, 2013.

```sql
select query, datediff(seconds, starttime, endtime),
trim(querytxt) as sqlquery
from stl_query
where starttime >= '2013-02-15 00:00' and endtime < '2013-02-15 23:59'
order by date_diff desc;
```

<table>
<thead>
<tr>
<th>query</th>
<th>date_diff</th>
<th>sqlquery</th>
</tr>
</thead>
<tbody>
<tr>
<td>55</td>
<td>119</td>
<td>padb_fetch_sample: select count(*) from category</td>
</tr>
<tr>
<td>121</td>
<td>9</td>
<td>select * from svl_query_summary;</td>
</tr>
<tr>
<td>181</td>
<td>6</td>
<td>select * from svl_query_summary where query in(179,178);</td>
</tr>
<tr>
<td>172</td>
<td>5</td>
<td>select * from svl_query_summary where query=148;</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(189 rows)
STL_QUERYTEXT

Captures the query text for SQL commands.

Query the STL_QUERYTEXT table to capture the SQL that was logged for the following statements:

- SELECT, SELECT INTO
- INSERT, UPDATE, DELETE
- COPY
- VACUUM, ANALYZE
- CREATE TABLE AS (CTAS)

To query activity for these statements over a given time period, join the STL_QUERYTEXT and
STL_QUERY tables.

**Note**
The STL_QUERY and STL_QUERYTEXT tables only contain information about queries, not
other utility and DDL commands. For a listing and information on all statements executed by
Amazon Redshift, you can also query the STLDDLTEXT and STLUTILITYTEXT tables. For
a complete listing of all statements executed by Amazon Redshift, you can query the
SVLSTATEMENTTEXT view.

See also STLDDLTEXT (p. 591), STLUTILITYTEXT (p. 640), and SVLSTATEMENTTEXT (p. 683).

This table is visible to all users.

**Table columns**

<table>
<thead>
<tr>
<th>Column name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>userid</td>
<td>integer</td>
<td>ID of the user who generated the entry.</td>
</tr>
<tr>
<td>query</td>
<td>integer</td>
<td>Query ID. The query column can be used to join other system tables and views.</td>
</tr>
<tr>
<td>xid</td>
<td>bigint</td>
<td>Transaction ID.</td>
</tr>
<tr>
<td>pid</td>
<td>integer</td>
<td>Process ID. All of the queries in a session are run in the same process, so this value remains constant if you run a series of queries in the same session. You can use this column to join to the STL_ERROR (p. 596) table.</td>
</tr>
<tr>
<td>sequence</td>
<td>integer</td>
<td>When a single statement contains more than 200 characters, additional rows are logged for that statement. Sequence 0 is the first row, 1 is the second, and so on.</td>
</tr>
<tr>
<td>text</td>
<td>character(200)</td>
<td>SQL text, in 200-character increments.</td>
</tr>
</tbody>
</table>

**Sample queries**

You can use the PG_BACKEND_PID() function to retrieve information for the current session. For example, the following query returns the query ID and a portion of the query text for queries executed in the current session.
select query, substring(text,1,60)
from stl_querytext
where pid = pg_backend_pid()
order by query desc;

<table>
<thead>
<tr>
<th>query</th>
<th>substring</th>
</tr>
</thead>
<tbody>
<tr>
<td>select query, substring(text,1,80) from stl_querytext where</td>
<td></td>
</tr>
<tr>
<td>select query, substring(path,0,80) as path from stl_unload_l</td>
<td></td>
</tr>
<tr>
<td>copy category from 's3://dw-ticket/manifest/category/1030_ma</td>
<td></td>
</tr>
<tr>
<td>Count rows in target table</td>
<td></td>
</tr>
<tr>
<td>unload ('select * from category') to 's3://dw-ticket/manifes</td>
<td></td>
</tr>
<tr>
<td>select query, substring(text,1,40) from stl_querytext where</td>
<td></td>
</tr>
</tbody>
</table>

(6 rows)

STL_REPLACEMENTS

Displays a log that records when invalid UTF-8 characters were replaced by the COPY (p. 276) command with the ACCEPTINVCHARS option. A log entry is added to STL_REPLACEMENTS for each of the first 100 rows on each node slice that required at least one replacement.

This table is visible to all users.

Table columns

<table>
<thead>
<tr>
<th>Column name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>userid</td>
<td>integer</td>
<td>ID of the user who generated the entry.</td>
</tr>
<tr>
<td>query</td>
<td>integer</td>
<td>Query ID. The query column can be used to join other system tables and views.</td>
</tr>
<tr>
<td>slice</td>
<td>integer</td>
<td>Node slice number where the replacement occurred.</td>
</tr>
<tr>
<td>tbl</td>
<td>integer</td>
<td>Table ID.</td>
</tr>
<tr>
<td>starttime</td>
<td>timestamp</td>
<td>Start time in UTC for the COPY command.</td>
</tr>
<tr>
<td>session</td>
<td>integer</td>
<td>Session ID for the session performing the COPY command.</td>
</tr>
<tr>
<td>filename</td>
<td>character(256)</td>
<td>Complete path to the input file for the COPY command.</td>
</tr>
<tr>
<td>line_number</td>
<td>bigint</td>
<td>Line number in the input data file that contained an invalid UTF-8 character.</td>
</tr>
<tr>
<td>colname</td>
<td>character(127)</td>
<td>First field that contained an invalid UTF-8 character.</td>
</tr>
<tr>
<td>raw_line</td>
<td>character(1024)</td>
<td>Raw load data that contained an invalid UTF-8 character.</td>
</tr>
</tbody>
</table>

Sample queries

The following example returns replacements for the most recent COPY operation.

```sql
select query, session, filename, line_number, colname
from stl replacements
where query = pg_last_copy_id();
```
STL_RETURN

Contains details for return steps in queries. A return step returns the results of queries executed on the compute nodes to the leader node. The leader node then merges the data and returns the results to the requesting client. For queries executed on the leader node, a return step returns results to the client.

A query consists of multiple segments, and each segment consists of one or more steps. For more information, see Query execution steps (p. 12).

STL_RETURN is user visible.

### Table columns

<table>
<thead>
<tr>
<th>Column name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>userid</td>
<td>integer</td>
<td>ID of the user who generated the entry.</td>
</tr>
<tr>
<td>query</td>
<td>integer</td>
<td>Query ID. The query column can be used to join other system tables and views.</td>
</tr>
<tr>
<td>slice</td>
<td>integer</td>
<td>Number that identifies the slice where the query was running.</td>
</tr>
<tr>
<td>segment</td>
<td>integer</td>
<td>Number that identifies the query segment.</td>
</tr>
<tr>
<td>step</td>
<td>integer</td>
<td>Query step that executed.</td>
</tr>
<tr>
<td>starttime</td>
<td>timestamp</td>
<td>Time in UTC that the query started executing, with 6 digits of precision for fractional seconds. For example: 2009-06-12 11:29:19.131358.</td>
</tr>
<tr>
<td>endtime</td>
<td>timestamp</td>
<td>Time in UTC that the query finished executing, with 6 digits of precision for fractional seconds. For example: 2009-06-12 11:29:19.131358.</td>
</tr>
<tr>
<td>tasknum</td>
<td>integer</td>
<td>Number of the query task process that was assigned to execute the step.</td>
</tr>
<tr>
<td>rows</td>
<td>bigint</td>
<td>Total number of rows that were processed.</td>
</tr>
<tr>
<td>bytes</td>
<td>bigint</td>
<td>Size, in bytes, of all the output rows for the step.</td>
</tr>
<tr>
<td>packets</td>
<td>integer</td>
<td>Total number of packets sent over the network.</td>
</tr>
</tbody>
</table>

### Sample queries

The following query shows which steps in the most recent query were executed on each slice. (Slice 10000 is on the leader node.)
SELECT query, slice, segment, step, endtime, rows, packets
from stl_return where query = pg_last_query_id();

<table>
<thead>
<tr>
<th>query</th>
<th>slice</th>
<th>segment</th>
<th>step</th>
<th>endtime</th>
<th>rows</th>
<th>packets</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>2013-12-27 01:43:21.469043</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2013-12-27 01:43:21.473321</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>2013-12-27 01:43:21.469118</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>2013-12-27 01:43:21.474196</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>2013-12-27 01:43:21.47704</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>2013-12-27 01:43:21.478593</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>10000</td>
<td>4</td>
<td>1</td>
<td>2013-12-27 01:43:21.480755</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

(7 rows)

**STL_SAVE**

Contains details for save steps in queries. A save step saves the input stream to a transient table. A transient table is a temporary row set that stores intermediate results while a query runs.

A query consists of multiple segments, and each segment consists of one or more steps. For more information, see Query execution steps (p. 12).

STL_SAVE is user visible.

**Table columns**

<table>
<thead>
<tr>
<th>Column name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>userid</td>
<td>integer</td>
<td>ID of the user who generated the entry.</td>
</tr>
<tr>
<td>query</td>
<td>integer</td>
<td>Query ID. The query column can be used to join other system tables and views.</td>
</tr>
<tr>
<td>slice</td>
<td>integer</td>
<td>Number that identifies the slice where the query was running.</td>
</tr>
<tr>
<td>segment</td>
<td>integer</td>
<td>Number that identifies the query segment.</td>
</tr>
<tr>
<td>step</td>
<td>integer</td>
<td>Query step that executed.</td>
</tr>
<tr>
<td>starttime</td>
<td>timestamp</td>
<td>Time in UTC that the query started executing, with 6 digits of precision for fractional seconds. For example: 2009-06-12 11:29:19.131358.</td>
</tr>
<tr>
<td>endtime</td>
<td>timestamp</td>
<td>Time in UTC that the query finished executing, with 6 digits of precision for fractional seconds. For example: 2009-06-12 11:29:19.131358.</td>
</tr>
<tr>
<td>tasknum</td>
<td>integer</td>
<td>Number of the query task process that was assigned to execute the step.</td>
</tr>
<tr>
<td>rows</td>
<td>bigint</td>
<td>Total number of rows that were processed.</td>
</tr>
</tbody>
</table>
### Sample queries

The following query shows which save steps in the most recent query were executed on each slice.

```sql
select query, slice, segment, step, tasknum, rows, tbl
from stl_save where query = pg_last_query_id();
```

<table>
<thead>
<tr>
<th>query</th>
<th>slice</th>
<th>segment</th>
<th>step</th>
<th>tasknum</th>
<th>rows</th>
<th>tbl</th>
</tr>
</thead>
<tbody>
<tr>
<td>52236</td>
<td>3</td>
<td>0</td>
<td>2</td>
<td>21</td>
<td>0</td>
<td>239</td>
</tr>
<tr>
<td>52236</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>20</td>
<td>0</td>
<td>239</td>
</tr>
<tr>
<td>52236</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>20</td>
<td>0</td>
<td>239</td>
</tr>
<tr>
<td>52236</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>21</td>
<td>0</td>
<td>239</td>
</tr>
<tr>
<td>52236</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>21</td>
<td>0</td>
<td>239</td>
</tr>
<tr>
<td>52236</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>20</td>
<td>0</td>
<td>239</td>
</tr>
<tr>
<td>52236</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>20</td>
<td>0</td>
<td>239</td>
</tr>
<tr>
<td>52236</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>21</td>
<td>0</td>
<td>239</td>
</tr>
</tbody>
</table>

(8 rows)

### STL_S3CLIENT

Records transfer time and other performance metrics.

Use the STL_S3CLIENT table to find the time spent transferring data from Amazon S3 as part of a COPY command.

This table is visible to all users.

### Table columns

<table>
<thead>
<tr>
<th>Column name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>userid</td>
<td>integer</td>
<td>ID of the user who generated the entry.</td>
</tr>
<tr>
<td>query</td>
<td>integer</td>
<td>Query ID. The query column can be used to join other system tables and views.</td>
</tr>
<tr>
<td>slice</td>
<td>integer</td>
<td>Number that identifies the slice where the query was running.</td>
</tr>
<tr>
<td>recordtime</td>
<td>timestamp</td>
<td>Time the record is logged.</td>
</tr>
<tr>
<td>pid</td>
<td>integer</td>
<td>Process ID. All of the queries in a session are run in the same process, so this value remains constant if you run a series of queries in the same session.</td>
</tr>
</tbody>
</table>
### Column name | Data type | Description
---|---|---
http_method | character(64) | HTTP method name corresponding to the Amazon S3 request.
bucket | character(64) | S3 bucket name.
key | character(256) | Key corresponding to the Amazon S3 object.
transfer_size | bigint | Number of bytes transferred.
data_size | bigint | Number of bytes of data. This value is the same as transfer_size for uncompressed data. If compression was used, this is the size of the uncompressed data.
start_time | bigint | Time when the transfer began (in microseconds).
end_time | bigint | Time when the transfer ended (in microseconds).
transfer_time | bigint | Time taken by the transfer (in microseconds).
compression_time | bigint | Portion of the transfer time that was spent uncompressing data (in microseconds).
connect_time | bigint | Time from the start until the connect to the remote server was completed (in microseconds).
app_connect_time | bigint | Time from the start until the SSL connect/handshake with the remote host was completed (in microseconds).
retries | bigint | Number of times the transfer was retried.
request_id | char(32) | Request ID from Amazon S3 HTTP response header.
extended_request_id | char(128) | Extended request ID from Amazon S3 HTTP header response (x-amz-id-2).
ip_address | char(64) | IP address of the server (ip V4 or V6).

### Sample query

The following query returns the time taken to load files using a COPY command.

```sql
select slice, key, transfer_time
from stl_s3client
where query = pg_last_copy_id();
```

<table>
<thead>
<tr>
<th>slice</th>
<th>key</th>
<th>transfer_time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>listing10M0003_part_00</td>
<td>16626716</td>
</tr>
<tr>
<td>1</td>
<td>listing10M0001_part_00</td>
<td>12894494</td>
</tr>
<tr>
<td>2</td>
<td>listing10M0002_part_00</td>
<td>14320978</td>
</tr>
<tr>
<td>3</td>
<td>listing10M0000_part_00</td>
<td>11293439</td>
</tr>
<tr>
<td>3371</td>
<td>prefix=listing10M;marker=</td>
<td>99395</td>
</tr>
</tbody>
</table>
STL_S3CLIENT_ERROR

Records errors encountered by a slice while loading a file from Amazon S3.

Use the STL_S3CLIENT_ERROR to find details for errors encountered while transferring data from Amazon S3 as part of a COPY command.

This table is visible to all users.

Table columns

<table>
<thead>
<tr>
<th>Column name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>userid</td>
<td>integer</td>
<td>ID of the user who generated the entry.</td>
</tr>
<tr>
<td>query</td>
<td>integer</td>
<td>Query ID. The query column can be used to join other system tables and views.</td>
</tr>
<tr>
<td>slice</td>
<td>integer</td>
<td>Number that identifies the slice where the query was running.</td>
</tr>
<tr>
<td>recordtime</td>
<td>timestamp</td>
<td>Time the record is logged.</td>
</tr>
<tr>
<td>pid</td>
<td>integer</td>
<td>Process ID. All of the queries in a session are run in the same process, so this value remains constant if you run a series of queries in the same session.</td>
</tr>
<tr>
<td>http_method</td>
<td>character(64)</td>
<td>HTTP method name corresponding to the Amazon S3 request.</td>
</tr>
<tr>
<td>bucket</td>
<td>character(64)</td>
<td>Amazon S3 bucket name.</td>
</tr>
<tr>
<td>key</td>
<td>character(256)</td>
<td>Key corresponding to the Amazon S3 object.</td>
</tr>
<tr>
<td>error</td>
<td>character(1024)</td>
<td>Error message.</td>
</tr>
</tbody>
</table>

Sample query

The following query returns the errors from COPY commands executed during the current session.

```sql
select query, sliceid, substring(key from 1 for 20) as file, substring(error from 1 for 35) as error
from stl_s3client_error
where pid = pg_backend_pid()
order by query desc;
```

Result

<table>
<thead>
<tr>
<th>query</th>
<th>sliceid</th>
<th>file</th>
<th>error</th>
</tr>
</thead>
<tbody>
<tr>
<td>362228</td>
<td>12</td>
<td>part.tbl.25.159.gz</td>
<td>transfer closed with 1947655 bytes</td>
</tr>
<tr>
<td>362228</td>
<td>24</td>
<td>part.tbl.15.577.gz</td>
<td>transfer closed with 1881910 bytes</td>
</tr>
<tr>
<td>362228</td>
<td>7</td>
<td>part.tbl.22.600.gz</td>
<td>transfer closed with 700143 bytes</td>
</tr>
<tr>
<td>362228</td>
<td>22</td>
<td>part.tbl.3.34.gz</td>
<td>transfer closed with 2334528 bytes</td>
</tr>
<tr>
<td>362228</td>
<td>11</td>
<td>part.tbl.30.274.gz</td>
<td>transfer closed with 699031 bytes</td>
</tr>
<tr>
<td>362228</td>
<td>30</td>
<td>part.tbl.5.509.gz</td>
<td>Unknown SSL protocol error in conne</td>
</tr>
</tbody>
</table>
STL_SCAN

Analyzes table scan steps for queries. The step number for rows in this table is always 0 because a scan is the first step in a segment.

This table is visible to all users.

Table columns

<table>
<thead>
<tr>
<th>Column name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>userid</td>
<td>integer</td>
<td>ID of the user who generated the entry.</td>
</tr>
<tr>
<td>query</td>
<td>integer</td>
<td>Query ID. The query column can be used to join other system tables and views.</td>
</tr>
<tr>
<td>slice</td>
<td>integer</td>
<td>Number that identifies the slice where the query was running.</td>
</tr>
<tr>
<td>segment</td>
<td>integer</td>
<td>Number that identifies the query segment.</td>
</tr>
<tr>
<td>step</td>
<td>integer</td>
<td>Query step that executed.</td>
</tr>
<tr>
<td>starttime</td>
<td>timestamp</td>
<td>Time in UTC that the query started executing, with 6 digits of precision for fractional seconds. For example: 2009-06-12 11:29:19.131358.</td>
</tr>
<tr>
<td>endtime</td>
<td>timestamp</td>
<td>Time in UTC that the query finished executing, with 6 digits of precision for fractional seconds. For example: 2009-06-12 11:29:19.131358.</td>
</tr>
<tr>
<td>tasknum</td>
<td>integer</td>
<td>Number of the query task process that was assigned to execute the step.</td>
</tr>
<tr>
<td>rows</td>
<td>bigint</td>
<td>Total number of rows that were processed.</td>
</tr>
<tr>
<td>bytes</td>
<td>bigint</td>
<td>Size, in bytes, of all the output rows for the step.</td>
</tr>
<tr>
<td>fetches</td>
<td>bigint</td>
<td>This information is for internal use only.</td>
</tr>
<tr>
<td>type</td>
<td>integer</td>
<td>ID of the scan type. For a list of valid values, see the following table.</td>
</tr>
<tr>
<td>tbl</td>
<td>integer</td>
<td>Table ID.</td>
</tr>
<tr>
<td>is_rrscan</td>
<td>character(1)</td>
<td>If true (t), indicates that range-restricted scan was used on the step.</td>
</tr>
<tr>
<td>is_delayed_scan</td>
<td>character(1)</td>
<td>This information is for internal use only.</td>
</tr>
<tr>
<td>rows_pre_filter</td>
<td>bigint</td>
<td>For scans of permanent tables, the total number of rows emitted before filtering rows marked for deletion (ghost rows) and before applying user-defined query filters.</td>
</tr>
</tbody>
</table>
### Scan types

<table>
<thead>
<tr>
<th>Type ID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Data from the network.</td>
</tr>
<tr>
<td>2</td>
<td>Permanent user tables in compressed shared memory.</td>
</tr>
<tr>
<td>3</td>
<td>Transient row-wise tables.</td>
</tr>
<tr>
<td>21</td>
<td>Load files from Amazon S3.</td>
</tr>
<tr>
<td>22</td>
<td>Load tables from Amazon DynamoDB.</td>
</tr>
<tr>
<td>23</td>
<td>Load data from a remote SSH connection.</td>
</tr>
<tr>
<td>24</td>
<td>Load data from remote cluster (sorted region). This is used for resizing.</td>
</tr>
<tr>
<td>25</td>
<td>Load data from remote cluster (unsorted region). This is used for resizing.</td>
</tr>
</tbody>
</table>

### Usage notes

Ideally rows should be relatively close to rows_pre_filter. A large difference between rows and rows_pre_filter is an indication that the execution engine is scanning rows that are later discarded, which is inefficient. The difference between rows_pre_filter and rows_pre_user_filter is the number of ghost rows in the scan. Run a VACUUM to remove rows marked for deletion. The difference between rows and rows_pre_user_filter is the number of rows filtered by the query. If a lot of rows are discarded by the user filter, review your choice of sort column or, if this is due to a large unsorted region, run a vacuum.

### Sample queries

The following example shows that rows_pre_filter is larger than rows_pre_user_filter because the table has deleted rows that have not been vacuumed (ghost rows).

```sql
SELECT slice, segment, step, rows, rows_pre_filter, rows_pre_user_filter
FROM stl_scan WHERE query = pg_last_query_id();
```

<table>
<thead>
<tr>
<th>query</th>
<th>slice</th>
<th>segment</th>
<th>step</th>
<th>rows</th>
<th>rows_pre_filter</th>
<th>rows_pre_user_filter</th>
</tr>
</thead>
<tbody>
<tr>
<td>------</td>
<td>-------</td>
<td>---------</td>
<td>------</td>
<td>-------</td>
<td>-----------------</td>
<td>----------------------</td>
</tr>
<tr>
<td>42915</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>43159</td>
<td>86318</td>
<td></td>
</tr>
<tr>
<td>43159</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>42915</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
```
STL_SESSIONS

Returns information about user session history.

STL_SESSIONS differs from STV_SESSIONS in that STL_SESSIONS contains session history, where STV_SESSIONS contains the current active sessions.

This table is visible to all users.

Table columns

<table>
<thead>
<tr>
<th>Column name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>userid</td>
<td>integer</td>
<td>ID of the user who generated the entry.</td>
</tr>
<tr>
<td>starttime</td>
<td>timestamp</td>
<td>Time in UTC that the session started.</td>
</tr>
<tr>
<td>endtime</td>
<td>timestamp</td>
<td>Time in UTC that the session ended.</td>
</tr>
<tr>
<td>process</td>
<td>integer</td>
<td>Process ID for the session.</td>
</tr>
<tr>
<td>user_name</td>
<td>character(50)</td>
<td>User name associated with the session.</td>
</tr>
<tr>
<td>db_name</td>
<td>character(50)</td>
<td>Name of the database associated with the session.</td>
</tr>
</tbody>
</table>

Sample queries

To view session history for the TICKIT database, type the following query:

```sql
SELECT starttime, process, user_name
FROM stl_sessions
WHERE db_name='tickit' ORDER BY starttime;
```

This query returns the following sample output:
### STL_SORT

Displays sort execution steps for queries, such as steps that use ORDER BY processing.

This table is visible to all users.

#### Table columns

<table>
<thead>
<tr>
<th>Column name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>userid</td>
<td>integer</td>
<td>ID of the user who generated the entry.</td>
</tr>
<tr>
<td>query</td>
<td>integer</td>
<td>Query ID. The query column can be used to join other system tables and views.</td>
</tr>
<tr>
<td>slice</td>
<td>integer</td>
<td>Number that identifies the slice where the query was running.</td>
</tr>
<tr>
<td>segment</td>
<td>integer</td>
<td>Number that identifies the query segment.</td>
</tr>
<tr>
<td>step</td>
<td>integer</td>
<td>Query step that executed.</td>
</tr>
<tr>
<td>starttime</td>
<td>timestamp</td>
<td>Time in UTC that the query started executing, with 6 digits of precision for fractional seconds. For example: <strong>2009-06-12 11:29:19.131358</strong>.</td>
</tr>
<tr>
<td>endtime</td>
<td>timestamp</td>
<td>Time in UTC that the query finished executing, with 6 digits of precision for fractional seconds. For example: <strong>2009-06-12 11:29:19.131358</strong>.</td>
</tr>
<tr>
<td>tasknum</td>
<td>integer</td>
<td>Number of the query task process that was assigned to execute the step.</td>
</tr>
<tr>
<td>rows</td>
<td>bigint</td>
<td>Total number of rows that were processed.</td>
</tr>
<tr>
<td>bytes</td>
<td>bigint</td>
<td>Size, in bytes, of all the output rows for the step.</td>
</tr>
</tbody>
</table>
### Sample queries

The following example returns sort results for slice 0 and segment 1.

```sql
select query, bytes, tbl, is_diskbased, workmem
from stl_sort
where slice=0 and segment=1;
```

<table>
<thead>
<tr>
<th>query</th>
<th>bytes</th>
<th>tbl</th>
<th>is_diskbased</th>
<th>workmem</th>
</tr>
</thead>
<tbody>
<tr>
<td>567</td>
<td>3126968</td>
<td>241</td>
<td>f</td>
<td>383385600</td>
</tr>
<tr>
<td>604</td>
<td>5292</td>
<td>242</td>
<td>f</td>
<td>383385600</td>
</tr>
<tr>
<td>675</td>
<td>104776</td>
<td>251</td>
<td>f</td>
<td>383385600</td>
</tr>
<tr>
<td>525</td>
<td>3126968</td>
<td>251</td>
<td>f</td>
<td>383385600</td>
</tr>
<tr>
<td>585</td>
<td>5068</td>
<td>241</td>
<td>f</td>
<td>383385600</td>
</tr>
<tr>
<td>630</td>
<td>204808</td>
<td>266</td>
<td>f</td>
<td>383385600</td>
</tr>
<tr>
<td>704</td>
<td>0</td>
<td>242</td>
<td>f</td>
<td>0</td>
</tr>
<tr>
<td>669</td>
<td>4606416</td>
<td>241</td>
<td>f</td>
<td>383385600</td>
</tr>
<tr>
<td>696</td>
<td>104776</td>
<td>241</td>
<td>f</td>
<td>383385600</td>
</tr>
<tr>
<td>651</td>
<td>4606416</td>
<td>254</td>
<td>f</td>
<td>383385600</td>
</tr>
<tr>
<td>632</td>
<td>0</td>
<td>256</td>
<td>f</td>
<td>0</td>
</tr>
<tr>
<td>599</td>
<td>396</td>
<td>241</td>
<td>f</td>
<td>383385600</td>
</tr>
<tr>
<td>86397</td>
<td>0</td>
<td>242</td>
<td>f</td>
<td>0</td>
</tr>
<tr>
<td>621</td>
<td>5292</td>
<td>241</td>
<td>f</td>
<td>383385600</td>
</tr>
<tr>
<td>86325</td>
<td>0</td>
<td>242</td>
<td>f</td>
<td>0</td>
</tr>
<tr>
<td>572</td>
<td>5068</td>
<td>242</td>
<td>f</td>
<td>383385600</td>
</tr>
<tr>
<td>645</td>
<td>204808</td>
<td>241</td>
<td>f</td>
<td>383385600</td>
</tr>
<tr>
<td>590</td>
<td>396</td>
<td>242</td>
<td>f</td>
<td>383385600</td>
</tr>
</tbody>
</table>

(18 rows)

### STL_SSHCLIENT_ERROR

Records all errors seen by the SSH client.

This table is visible to all users.

#### Table columns

<table>
<thead>
<tr>
<th>Column name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>userid</td>
<td>integer</td>
<td>ID of the user who generated the entry.</td>
</tr>
</tbody>
</table>
### STL_STREAM_SEGS

Lists the relationship between streams and concurrent segments.

This table is visible to all users.

#### Table columns

<table>
<thead>
<tr>
<th>Column name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>userid</td>
<td>integer</td>
<td>ID of the user who generated the entry.</td>
</tr>
<tr>
<td>query</td>
<td>integer</td>
<td>Query ID. The query column can be used to join other system tables and views.</td>
</tr>
<tr>
<td>stream</td>
<td>integer</td>
<td>The set of concurrent segments of a query.</td>
</tr>
<tr>
<td>segment</td>
<td>integer</td>
<td>Number that identifies the query segment.</td>
</tr>
</tbody>
</table>

#### Sample queries

To view the relationship between streams and concurrent segments for the most recent query, type the following query:

```sql
select * from stl_stream_segs where query = pg_last_query_id();
```

<table>
<thead>
<tr>
<th>query</th>
<th>stream</th>
<th>segment</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>
STL_TR_CONFLICT

Displays information to identify and resolve lock conflicts with database tables.

A lock conflict can occur when two or more users are loading, inserting, deleting, or updating data rows in the same table at the same time. Every time a lock conflict occurs, Amazon Redshift writes a data row to the STL_TR_CONFLICT system table.

This table is visible to superusers.

Table columns

<table>
<thead>
<tr>
<th>Column name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xact_id</td>
<td>bigint</td>
<td>Transaction ID for the rolled back transaction.</td>
</tr>
<tr>
<td>process_id</td>
<td>bigint</td>
<td>Process associated with the lock.</td>
</tr>
<tr>
<td>xact_start_ts</td>
<td>timestamp</td>
<td>Timestamp for the transaction start.</td>
</tr>
<tr>
<td>abort_time</td>
<td>timestamp</td>
<td>Time that the transaction was aborted.</td>
</tr>
<tr>
<td>table_id</td>
<td>bigint</td>
<td>Table ID for the table where the conflict occurred.</td>
</tr>
</tbody>
</table>

Sample query

For examples of lock conflicts, see Managing concurrent write operations (p. 169). To return information about conflicts that involved a particular table, run a query that specifies the table ID:

```sql
select * from stl_tr_conflict where table_id=100234
order by xact_start_ts;
```

<table>
<thead>
<tr>
<th>xact_id</th>
<th>process_id</th>
<th>xact_start_ts</th>
<th>abort_time</th>
<th>table_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>1876</td>
<td>8551</td>
<td>2010-03-30 09:19:15.852326</td>
<td>2010-03-30 09:20:17.582499</td>
<td>100234</td>
</tr>
<tr>
<td>1928</td>
<td>15034</td>
<td>2010-03-30 13:20:00.636045</td>
<td>2010-03-30 13:20:47.766817</td>
<td>100234</td>
</tr>
<tr>
<td>1991</td>
<td>23753</td>
<td>2010-04-01 13:05:01.220059</td>
<td>2010-04-01 13:06:06.94098</td>
<td>100234</td>
</tr>
</tbody>
</table>

You can get the table ID from the DETAIL section of the error message for serializability violations (error 1023).

STL_UNDONE

Displays information about transactions that have been undone.

This table is visible to all users.
### Table columns

<table>
<thead>
<tr>
<th>Column name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>userid</td>
<td>integer</td>
<td>ID of the user who generated the entry.</td>
</tr>
<tr>
<td>xact_id</td>
<td>bigint</td>
<td>ID for the undo transaction.</td>
</tr>
<tr>
<td>xact_id_undone</td>
<td>bigint</td>
<td>ID for the transaction that was undone.</td>
</tr>
<tr>
<td>undo_start_ts</td>
<td>timestamp</td>
<td>Start time for the undo transaction.</td>
</tr>
<tr>
<td>undo_end_ts</td>
<td>timestamp</td>
<td>End time for the undo transaction.</td>
</tr>
<tr>
<td>table_id</td>
<td>bigint</td>
<td>ID for the table that was affected by the undo transaction.</td>
</tr>
</tbody>
</table>

### Sample query

To view a concise log of all undone transactions, type the following command:

```sql
select xact_id, xact_id_undone, table_id from stl_undone;
```

This command returns the following sample output:

```
<table>
<thead>
<tr>
<th>xact_id</th>
<th>xact_id_undone</th>
<th>table_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>1344</td>
<td>1344</td>
<td>100192</td>
</tr>
<tr>
<td>1326</td>
<td>1326</td>
<td>100192</td>
</tr>
<tr>
<td>1551</td>
<td>1551</td>
<td>100192</td>
</tr>
</tbody>
</table>
(3 rows)
```

### STL_UNIQUE

Analyzes execution steps that occur when a DISTINCT function is used in the SELECT list or when duplicates are removed in a UNION or INTERSECT query.

This table is visible to all users.

### Table columns

<table>
<thead>
<tr>
<th>Column name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>userid</td>
<td>integer</td>
<td>ID of the user who generated the entry.</td>
</tr>
<tr>
<td>query</td>
<td>integer</td>
<td>Query ID. The query column can be used to join other system tables and views.</td>
</tr>
<tr>
<td>slice</td>
<td>integer</td>
<td>Number that identifies the slice where the query was running.</td>
</tr>
<tr>
<td>segment</td>
<td>integer</td>
<td>Number that identifies the query segment.</td>
</tr>
<tr>
<td>step</td>
<td>integer</td>
<td>Query step that executed.</td>
</tr>
</tbody>
</table>
**Sample queries**

Suppose you execute the following query:

```sql
select distinct eventname
from event order by 1;
```

Assuming the ID for the previous query is 6313, the following example shows the number of rows produced by the unique step for each slice in segments 0 and 1.

```sql
select query, slice, segment, step, datediff(msec, starttime, endtime) as msec, tasknum, rows
from stl_unique where query = 6313
order by query desc, slice, segment, step;
```
STL_UNLOAD_LOG

Records the details for an unload operation.

STL_UNLOAD_LOG records one row for each file created by an UNLOAD statement. For example, if an UNLOAD creates 12 files, STL_UNLOAD_LOG will contain 12 corresponding rows.

This table is visible to all users.

**Table columns**

<table>
<thead>
<tr>
<th>Column name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>userid</td>
<td>integer</td>
<td>ID of the user who generated the entry.</td>
</tr>
<tr>
<td>query</td>
<td>integer</td>
<td>ID for the transaction.</td>
</tr>
<tr>
<td>slice</td>
<td>integer</td>
<td>Number that identifies the slice where the query was running.</td>
</tr>
<tr>
<td>pid</td>
<td>integer</td>
<td>Process ID associated with the query statement.</td>
</tr>
<tr>
<td>path</td>
<td>character(1280)</td>
<td>The complete Amazon S3 object path for the file.</td>
</tr>
<tr>
<td>start_time</td>
<td>timestamp</td>
<td>Start time for the transaction.</td>
</tr>
<tr>
<td>end_time</td>
<td>timestamp</td>
<td>End time for the transaction.</td>
</tr>
<tr>
<td>line_count</td>
<td>bigint</td>
<td>Number of lines (rows) unloaded to the file.</td>
</tr>
<tr>
<td>transfer_size</td>
<td>bigint</td>
<td>Number of bytes transferred.</td>
</tr>
</tbody>
</table>

**Sample query**

To get a list of the files that were written to Amazon S3 by an UNLOAD command, you can call an Amazon S3 list operation after the UNLOAD completes; however, depending on how quickly you issue the call, the list might be incomplete because an Amazon S3 list operation is eventually consistent. To get a complete, authoritative list immediately, query STL_UNLOAD_LOG.

The following query returns the pathname for files that were created by an UNLOAD with query ID 2320:

```sql
select query, substring(path,0,40) as path
from stl_unload_log
where query=2320
order by path;
```
This command returns the following sample output:

<table>
<thead>
<tr>
<th>query</th>
<th>path</th>
</tr>
</thead>
<tbody>
<tr>
<td>2320</td>
<td>s3://my-bucket/venue0000_part_00</td>
</tr>
<tr>
<td>2320</td>
<td>s3://my-bucket/venue0001_part_00</td>
</tr>
<tr>
<td>2320</td>
<td>s3://my-bucket/venue0002_part_00</td>
</tr>
<tr>
<td>2320</td>
<td>s3://my-bucket/venue0003_part_00</td>
</tr>
</tbody>
</table>

(4 rows)

**STL_USERLOG**

Records details for the following changes to a database user:

- Create user
- Drop user
- Alter user (rename)
- Alter user (alter properties)

This table is visible to superusers.

**Table columns**

<table>
<thead>
<tr>
<th>Column name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>userid</td>
<td>integer</td>
<td>ID of the user who generated the entry.</td>
</tr>
<tr>
<td>username</td>
<td>character(50)</td>
<td>User name of the user affected by the change.</td>
</tr>
<tr>
<td>oldusername</td>
<td>character(50)</td>
<td>For a rename action, the original user name. For any other action, this field is empty.</td>
</tr>
<tr>
<td>action</td>
<td>character(10)</td>
<td>Action that occurred. Valid values:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Alter</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Create</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Drop</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Rename</td>
</tr>
<tr>
<td>usecreatedb</td>
<td>integer</td>
<td>If true (1), indicates that the user has create database privileges.</td>
</tr>
<tr>
<td>usesuper</td>
<td>integer</td>
<td>If true (1), indicates that the user is a superuser.</td>
</tr>
<tr>
<td>usecatupd</td>
<td>integer</td>
<td>If true (1), indicates that the user can update system catalogs.</td>
</tr>
<tr>
<td>valuntil</td>
<td>timestamp</td>
<td>Password expiration date.</td>
</tr>
<tr>
<td>pid</td>
<td>integer</td>
<td>Process ID.</td>
</tr>
<tr>
<td>xid</td>
<td>bigint</td>
<td>Transaction ID.</td>
</tr>
<tr>
<td>recordtime</td>
<td>timestamp</td>
<td>Time in UTC that the query started.</td>
</tr>
</tbody>
</table>
Sample queries

The following example performs four user actions, then queries the STL_USERLOG table.

```sql
create user userlog1 password 'Userlog1';
alter user userlog1 createdb createuser;
alter user userlog1 rename to userlog2;
drop user userlog2;

select userid, username, oldusername, action, usecreatedb, usesuper from
stl_userlog order by recordtime desc;
```

<table>
<thead>
<tr>
<th>userid</th>
<th>username</th>
<th>oldusername</th>
<th>action</th>
<th>usecreatedb</th>
<th>usesuper</th>
</tr>
</thead>
<tbody>
<tr>
<td>108</td>
<td>userlog2</td>
<td></td>
<td>drop</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>108</td>
<td>userlog2</td>
<td>userlog1</td>
<td>rename</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>108</td>
<td>userlog1</td>
<td></td>
<td>alter</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>108</td>
<td>userlog1</td>
<td></td>
<td>create</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

(4 rows)

STL_UTILITYTEXT

Captures the text of non-SELECT SQL commands run on the database.

Query the STL_UTILITYTEXT table to capture the following subset of SQL statements that were run on
the system:

- ABORT, BEGIN, COMMIT, END, ROLLBACK
- CANCEL
- COMMENT
- CREATE, ALTER, DROP DATABASE
- CREATE, ALTER, DROP USER
- EXPLAIN
- GRANT, REVOKE
- LOCK
- RESET
- SET
- SHOW
- TRUNCATE

See also STL_DDLTEXT (p. 591), STL_QUERYTEXT (p. 622), and SVL_STATEMENTTEXT (p. 683).

Use the starttime and endtime columns to find out which statements were logged during a given
time period. Long blocks of SQL text are broken into lines 200 characters long; the sequence column
identifies fragments of text that belong to a single statement.

This table is visible to all users.
Table columns

<table>
<thead>
<tr>
<th>Column name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>userid</td>
<td>integer</td>
<td>ID of the user who generated the entry.</td>
</tr>
<tr>
<td>xid</td>
<td>bigint</td>
<td>Transaction ID.</td>
</tr>
<tr>
<td>pid</td>
<td>integer</td>
<td>Process ID associated with the query statement.</td>
</tr>
<tr>
<td>label</td>
<td>character(30)</td>
<td>Either the name of the file used to run the query or a label defined with</td>
</tr>
<tr>
<td></td>
<td></td>
<td>a SET QUERY_GROUP command. If the query is not file-based or the QUERY_</td>
</tr>
<tr>
<td></td>
<td></td>
<td>GROUP parameter is not set, this field is blank.</td>
</tr>
<tr>
<td>starttime</td>
<td>timestamp</td>
<td>Time in UTC that the query started executing, with 6 digits of precision</td>
</tr>
<tr>
<td>endtime</td>
<td>timestamp</td>
<td>Time in UTC that the query finished executing, with 6 digits of precision</td>
</tr>
<tr>
<td>sequence</td>
<td>integer</td>
<td>When a single statement contains more than 200 characters, additional rows</td>
</tr>
<tr>
<td></td>
<td></td>
<td>are logged for that statement. Sequence 0 is the first row, 1 is the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>second, and so on.</td>
</tr>
<tr>
<td>text</td>
<td>character(200)</td>
<td>SQL text, in 200-character increments.</td>
</tr>
</tbody>
</table>

Sample queries

The following query returns the text for "utility" commands that were run on January 26th, 2012. In this case, some SET commands and a SHOW ALL command were run:

```sql
select starttime, sequence, rtrim(text)
from stl_utilitytext
where starttime like '2012-01-26%'
order by starttime, sequence;
```

<table>
<thead>
<tr>
<th>starttime</th>
<th>sequence</th>
<th>rtrim</th>
</tr>
</thead>
<tbody>
<tr>
<td>2012-01-26 13:05:52.529235</td>
<td>0</td>
<td>show all;</td>
</tr>
<tr>
<td>2012-01-26 13:20:31.660255</td>
<td>0</td>
<td>SET query_group to ''</td>
</tr>
<tr>
<td>2012-01-26 13:20:54.956131</td>
<td>0</td>
<td>SET query_group to 'soldunsold.sql'</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

STL_VACUUM

Displays row and block statistics for tables that have been vacuumed.

The table shows information specific to when each vacuum operation started and finished, and demonstrates the benefits of running the operation. See the VACUUM (p. 411) command description for information about the requirements for running this command.

This table is visible to superusers.
Table columns

<table>
<thead>
<tr>
<th>Column name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>userid</td>
<td>integer</td>
<td>ID of the user who generated the entry.</td>
</tr>
<tr>
<td>xid</td>
<td>bigint</td>
<td>Transaction ID for the VACUUM statement. You can join this table to the STL_QUERY table to see the individual SQL statements that are run for a given VACUUM transaction. If you vacuum the whole database, each table is vacuumed in a separate transaction.</td>
</tr>
<tr>
<td>tbl</td>
<td>integer</td>
<td>Table ID.</td>
</tr>
<tr>
<td>status</td>
<td>character(30)</td>
<td>Started, Started Delete Only, Started Sort Only, Finished, Skipped, Skipped (delete only), or Skipped (sort only). Subtract the Started time from the Finished time for a particular transaction ID and table ID to find out how long the vacuum operation took on a specific table. A Skipped status means that the vacuum operation was not needed for a particular table.</td>
</tr>
<tr>
<td>rows</td>
<td>bigint</td>
<td>Actual number of rows in the table plus any deleted rows that are still stored on disk (waiting to be vacuumed). This column shows the count before the vacuum started for rows with a Started status, and the count after the vacuum for rows with a Finished status.</td>
</tr>
<tr>
<td>sortedrows</td>
<td>integer</td>
<td>Number of rows in the table that are sorted. This column shows the count before the vacuum started for rows with Started in the Status column, and the count after the vacuum for rows with Finished in the Status column.</td>
</tr>
<tr>
<td>blocks</td>
<td>integer</td>
<td>Total number of data blocks used to store the table data before the vacuum operation (rows with a Started status) and after the vacuum operation (Finished column). Each data block uses 1 MB.</td>
</tr>
<tr>
<td>max_merge_partitions</td>
<td>integer</td>
<td>This column is used for performance analysis and represents the maximum number of partitions that vacuum can process for the table per merge phase iteration. (Vacuum sorts the unsorted region into one or more sorted partitions. Depending on the number of columns in the table and the current Amazon Redshift configuration, the merge phase can process a maximum number of partitions in a single merge iteration. The merge phase will still work if the number of sorted partitions exceeds the maximum number of merge partitions, but more merge iterations will be required.)</td>
</tr>
<tr>
<td>eventtime</td>
<td>timestamp</td>
<td>When the vacuum operation started or finished.</td>
</tr>
</tbody>
</table>

Sample queries

The following query reports statistics for table 100236. The following operations were run in succession on this table:

1. DELETE 2,932,146 rows (the table contained 12,319,812 rows).
2. VACUUM the table.
3. INSERT 146,678 rows.
4. VACUUM the table.

```
select xid, table_id, status, rows, sortedrows, blocks, eventtime
from stl_vacuum where table_id=100236 order by eventtime;
```

<table>
<thead>
<tr>
<th>xid</th>
<th>table_id</th>
<th>status</th>
<th>rows</th>
<th>sortedrows</th>
<th>blocks</th>
<th>eventtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>1922</td>
<td>100236</td>
<td>Started</td>
<td>12319812</td>
<td>12319812</td>
<td>2476</td>
<td>2010-05-26 14:08:59...</td>
</tr>
<tr>
<td>1922</td>
<td>100236</td>
<td>Finished</td>
<td>9387396</td>
<td>9387396</td>
<td>2120</td>
<td>2010-05-26 14:09:10...</td>
</tr>
<tr>
<td>1927</td>
<td>100236</td>
<td>Started</td>
<td>9534074</td>
<td>9534074</td>
<td>2120</td>
<td>2010-05-26 14:18:25...</td>
</tr>
<tr>
<td>1927</td>
<td>100236</td>
<td>Finished</td>
<td>9534074</td>
<td>9534074</td>
<td>2120</td>
<td>2010-05-26 14:18:26...</td>
</tr>
</tbody>
</table>

At the start of the first VACUUM transaction (1922), the table contained 12319812 rows stored in 2476 blocks. When this transaction completed, space had been reclaimed for the deleted rows; therefore, the ROWS column shows a value of 9387396, and the BLOCKS column has dropped from 2476 to 2120. 356 blocks of disk space (35.6 GB) were reclaimed.

At the start of the second vacuum operation, the ROWS column had increased to 9534074 because of the INSERT operation. However, the SORTEDROWS column shows a value of 9387396 because the new rows were stored in the unsorted region when the vacuum started. When the VACUUM finished, the ROWS and SORTEDROWS values matched because all of the rows in the table were now in sorted order.

The following example shows the statistics for a SORT ONLY vacuum on the SALES table (table 110116 in this example) after a large INSERT operation dramatically increased the size of the table:

```
vacuum sort only sales;
```

```
select xid, table_id, status, rows, sortedrows, blocks, eventtime
from stl_vacuum order by xid, table_id, eventtime;
```

<table>
<thead>
<tr>
<th>xid</th>
<th>table_id</th>
<th>status</th>
<th>rows</th>
<th>sortedrows</th>
<th>blocks</th>
<th>eventtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>2925</td>
<td>110116</td>
<td>Started Sort Only</td>
<td>1379648</td>
<td>172456</td>
<td>132</td>
<td>2011-02-24 16:25:21...</td>
</tr>
<tr>
<td>2925</td>
<td>110116</td>
<td>Finished</td>
<td>1379648</td>
<td>1379648</td>
<td>132</td>
<td>2011-02-24 16:26:28...</td>
</tr>
</tbody>
</table>

**STL_WARNING**

Displays a log of any unexpected occurrences for Amazon Redshift that were not severe enough to cause an error.

This table is visible to all users.

**Table columns**

<table>
<thead>
<tr>
<th>Column name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>userid</td>
<td>integer</td>
<td>ID of the user who generated the entry.</td>
</tr>
</tbody>
</table>
**Sample query**

This table is used mainly for troubleshooting by Amazon Redshift support. If you are working with support on an issue, you might be asked to query this table to provide support with information to help resolve the problem.

**STL_WINDOW**

Analyzes query steps that execute window functions.

This table is visible to all users.

**Table columns**

<table>
<thead>
<tr>
<th>Column name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>process</td>
<td>character(12)</td>
<td>Process that triggered the warning.</td>
</tr>
<tr>
<td>pid</td>
<td>integer</td>
<td>Process ID.</td>
</tr>
<tr>
<td>recordtime</td>
<td>timestamp</td>
<td>Time that the warning occurred.</td>
</tr>
<tr>
<td>file</td>
<td>character(20)</td>
<td>Name of the source file where the warning occurred.</td>
</tr>
<tr>
<td>linenum</td>
<td>integer</td>
<td>Line number in the source file where the warning occurred.</td>
</tr>
<tr>
<td>bug_desc</td>
<td>character(512)</td>
<td>Warning message.</td>
</tr>
<tr>
<td>userid</td>
<td>integer</td>
<td>ID of the user who generated the entry.</td>
</tr>
<tr>
<td>query</td>
<td>integer</td>
<td>Query ID. The query column can be used to join other system tables and views.</td>
</tr>
<tr>
<td>slice</td>
<td>integer</td>
<td>Number that identifies the slice where the query was running.</td>
</tr>
<tr>
<td>segment</td>
<td>integer</td>
<td>Number that identifies the query segment.</td>
</tr>
<tr>
<td>step</td>
<td>integer</td>
<td>Query step that executed.</td>
</tr>
<tr>
<td>starttime</td>
<td>timestamp</td>
<td>Time in UTC that the query started executing, with 6 digits of precision for fractional seconds. For example: 2009-06-12 11:29:19.131358.</td>
</tr>
<tr>
<td>endtime</td>
<td>timestamp</td>
<td>Time in UTC that the query finished executing, with 6 digits of precision for fractional seconds. For example: 2009-06-12 11:29:19.131358.</td>
</tr>
<tr>
<td>tasknum</td>
<td>integer</td>
<td>Number of the query task process that was assigned to execute the step.</td>
</tr>
<tr>
<td>rows</td>
<td>bigint</td>
<td>Total number of rows that were processed.</td>
</tr>
<tr>
<td>is_diskbased</td>
<td>character(1)</td>
<td>If true (t), the query was executed as a disk-based operation. If false (f), the query was executed in memory.</td>
</tr>
</tbody>
</table>
### STL_WLM_ERROR

Records all WLM-related errors as they occur.

This table is visible to all users.

#### Table columns

<table>
<thead>
<tr>
<th>Column name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>userid</td>
<td>integer</td>
<td>ID of the user who generated the entry.</td>
</tr>
<tr>
<td>recordtime</td>
<td>timestamp</td>
<td>Time that the error occurred.</td>
</tr>
<tr>
<td>pid</td>
<td>integer</td>
<td>ID for the process that generated the error.</td>
</tr>
<tr>
<td>error_string</td>
<td>character(256)</td>
<td>Error description.</td>
</tr>
</tbody>
</table>

### STL_WLM_QUERY

Contains a record of each attempted execution of a query in a service class handled by WLM.

This table is visible to all users.
Table columns

<table>
<thead>
<tr>
<th>Column name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>userid</td>
<td>integer</td>
<td>ID of the user who generated the entry.</td>
</tr>
<tr>
<td>xid</td>
<td>integer</td>
<td>Transaction ID of the query or subquery.</td>
</tr>
<tr>
<td>task</td>
<td>integer</td>
<td>ID used to track a query through the workload manager. Can be associated with multiple query IDs. If a query is restarted, the query is assigned a new query ID but not a new task ID.</td>
</tr>
<tr>
<td>query</td>
<td>integer</td>
<td>Query ID. If a query is restarted, the query is assigned a new query ID but not a new task ID.</td>
</tr>
<tr>
<td>service_class</td>
<td>integer</td>
<td>ID for the service class. Service classes are defined in the WLM configuration file.</td>
</tr>
<tr>
<td>slot_count</td>
<td>integer</td>
<td>Number of WLM query slots.</td>
</tr>
<tr>
<td>service_class_start_time</td>
<td>timestamp</td>
<td>Time that the query was assigned to the service class.</td>
</tr>
<tr>
<td>queue_start_time</td>
<td>timestamp</td>
<td>Time that the query entered the queue for the service class.</td>
</tr>
<tr>
<td>queue_end_time</td>
<td>timestamp</td>
<td>Time when the query left the queue for the service class.</td>
</tr>
<tr>
<td>total_queue_time</td>
<td>bigint</td>
<td>Total number of microseconds that the query spent in the queue.</td>
</tr>
<tr>
<td>exec_start_time</td>
<td>timestamp</td>
<td>Time that the query began executing in the service class.</td>
</tr>
<tr>
<td>exec_end_time</td>
<td>timestamp</td>
<td>Time that the query completed execution in the service class.</td>
</tr>
<tr>
<td>total_exec_time</td>
<td>bigint</td>
<td>Number of microseconds that the query spent executing.</td>
</tr>
<tr>
<td>service_class_end_time</td>
<td>timestamp</td>
<td>Time that the query left the service class.</td>
</tr>
<tr>
<td>final_state</td>
<td>character(16)</td>
<td>End status of the query. Valid values are Completed, Evicted, or Rejected.</td>
</tr>
</tbody>
</table>

Sample queries

The WLM configuration used for these examples has three service classes:

- A system table service class (Service Class 1)
- An evictable service class for user queries with a threshold of 2000000 microseconds (2 seconds) with three query tasks (Service Class 2)
- A non-evictable service class for user queries with one query task (Service Class 3)

View average query time in queues and executing

The following query returns the average time (in microseconds) that each query spent in query queues and executing for each service class:
select service_class as svc_class, count(*) as count, 
avg(datediff(microseconds, queue_start_time, queue_end_time)) as avg_queue_time, 
avg(datediff(microseconds, exec_start_time, exec_end_time )) as avg_exec_time 
from stl_wlm_query 
group by service_class 
order by service_class;

This query returns the following sample output:

<table>
<thead>
<tr>
<th>svc_class</th>
<th>count</th>
<th>avg_queue_time</th>
<th>avg_exec_time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>20103</td>
<td>0</td>
<td>80415</td>
</tr>
<tr>
<td>2</td>
<td>3421</td>
<td>34015</td>
<td>234015</td>
</tr>
<tr>
<td>4</td>
<td>42</td>
<td>0</td>
<td>944266</td>
</tr>
<tr>
<td>6</td>
<td>196</td>
<td>6439</td>
<td>1364399</td>
</tr>
</tbody>
</table>

(4 rows)

View maximum query time in queues and executing

The following query returns the maximum amount of time (in microseconds) that a query spent in any query queue and executing for each service class. Use the execution time to check that no query greatly exceeded the eviction thresholds for the service class:

select service_class as svc_class, count(*) as count, 
max(datediff(microseconds, queue_start_time, queue_end_time)) as max_queue_time, 
max(datediff(microseconds, exec_start_time, exec_end_time )) as max_exec_time 
from stl_wlm_query 
group by service_class 
order by service_class;

This query returns the following sample output:

<table>
<thead>
<tr>
<th>svc_class</th>
<th>count</th>
<th>max_queue_time</th>
<th>max_exec_time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>20125</td>
<td>0</td>
<td>6342943</td>
</tr>
<tr>
<td>2</td>
<td>3421</td>
<td>22143</td>
<td>10221439</td>
</tr>
<tr>
<td>4</td>
<td>42</td>
<td>0</td>
<td>3775896</td>
</tr>
<tr>
<td>6</td>
<td>197</td>
<td>37947</td>
<td>16379473</td>
</tr>
</tbody>
</table>

(4 rows)

Note that the only service class where any queries had to spend time waiting in queue was Service Class 3, which only has one available query task. None of the queries' execution time in Service Class 2 exceeded the eviction threshold before moving to Service Class 3.

View all evicted/restarted queries

The following query returns records for all evicted queries that were restarted in another service class:

select xid, task, query, service_class as svc_class, 
datediff(milliseconds, exec_start_time, exec_end_time) as exec_time, final_state 
from stl_wlm_query 
where task in (select task 
from stl_wlm_query)
where final_state like '%Evicted%')
order by xid, task, query;

This query returns the following sample output:

<table>
<thead>
<tr>
<th>xid</th>
<th>task</th>
<th>query</th>
<th>svc_class</th>
<th>exec_time</th>
<th>final_state</th>
</tr>
</thead>
<tbody>
<tr>
<td>1504</td>
<td>122</td>
<td>122</td>
<td>2</td>
<td>2001845</td>
<td>Evicted</td>
</tr>
<tr>
<td>1504</td>
<td>122</td>
<td>123</td>
<td>3</td>
<td>7738786</td>
<td>Completed</td>
</tr>
<tr>
<td>1509</td>
<td>123</td>
<td>124</td>
<td>2</td>
<td>2002352</td>
<td>Evicted</td>
</tr>
<tr>
<td>1509</td>
<td>123</td>
<td>125</td>
<td>3</td>
<td>788426</td>
<td>Completed</td>
</tr>
<tr>
<td>1520</td>
<td>126</td>
<td>128</td>
<td>2</td>
<td>2001024</td>
<td>Evicted</td>
</tr>
<tr>
<td>1520</td>
<td>126</td>
<td>129</td>
<td>3</td>
<td>809977</td>
<td>Completed</td>
</tr>
<tr>
<td>1525</td>
<td>127</td>
<td>130</td>
<td>2</td>
<td>2001726</td>
<td>Evicted</td>
</tr>
<tr>
<td>1525</td>
<td>127</td>
<td>131</td>
<td>3</td>
<td>5624033</td>
<td>Completed</td>
</tr>
</tbody>
</table>

(26 rows)

Notice that each TASK in these results was assigned a new Query ID upon entering a new Service Class.

View queries that exceeded service class thresholds

The following query returns information for up to ten queries that exceeded the eviction threshold for Service Class 2.

```
select xid, task, query, service_class as svc_class, trim(final_state) as final_state, 
datediff(millisecond, exec_start_time, exec_end_time) as total_exec_time 
from stl_wlm_query 
where service_class = 2 and 
datediff(millisecond, exec_start_time, exec_end_time) > 1000 
order by exec_start_time desc 
limit 10;
```

This query returns the following sample output:

<table>
<thead>
<tr>
<th>xid</th>
<th>task</th>
<th>query</th>
<th>svc_class</th>
<th>final_state</th>
<th>total_exec_time</th>
</tr>
</thead>
<tbody>
<tr>
<td>69611</td>
<td>23039</td>
<td>23039</td>
<td>2</td>
<td>Completed</td>
<td>2481</td>
</tr>
<tr>
<td>69605</td>
<td>23037</td>
<td>23037</td>
<td>2</td>
<td>Completed</td>
<td>2503</td>
</tr>
<tr>
<td>69602</td>
<td>23036</td>
<td>23036</td>
<td>2</td>
<td>Completed</td>
<td>1646</td>
</tr>
<tr>
<td>69599</td>
<td>23035</td>
<td>23035</td>
<td>2</td>
<td>Completed</td>
<td>1225</td>
</tr>
<tr>
<td>68794</td>
<td>22766</td>
<td>22766</td>
<td>2</td>
<td>Completed</td>
<td>1306</td>
</tr>
<tr>
<td>67956</td>
<td>22439</td>
<td>22439</td>
<td>2</td>
<td>Completed</td>
<td>1555</td>
</tr>
<tr>
<td>67785</td>
<td>22382</td>
<td>22382</td>
<td>2</td>
<td>Completed</td>
<td>1033</td>
</tr>
<tr>
<td>67782</td>
<td>22381</td>
<td>22381</td>
<td>2</td>
<td>Completed</td>
<td>1733</td>
</tr>
<tr>
<td>67692</td>
<td>22351</td>
<td>22351</td>
<td>2</td>
<td>Completed</td>
<td>1788</td>
</tr>
<tr>
<td>67239</td>
<td>22200</td>
<td>22200</td>
<td>2</td>
<td>Completed</td>
<td>1161</td>
</tr>
</tbody>
</table>

(10 rows)
STV tables for snapshot data

STV tables are actually virtual system tables that contain snapshots of the current system data.

Topics
- STV_ACTIVE_CURSORS (p. 649)
- STV_BLOCKLIST (p. 650)
- STV_CURSOR_CONFIGURATION (p. 653)
- STV_EXEC_STATE (p. 653)
- STV_INFLIGHT (p. 654)
- STV_LOAD_STATE (p. 655)
- STV_LOCKS (p. 657)
- STV_PARTITIONS (p. 657)
- STV_RECENTS (p. 659)
- STV_SLICES (p. 660)
- STV_SESSIONS (p. 661)
- STV_TBL_PERM (p. 662)
- STV_TBL_TRANS (p. 664)
- STV_WLM_CLASSIFICATION_CONFIG (p. 665)
- STV_WLM_QUERY_QUEUE_STATE (p. 666)
- STV_WLM_QUERY_STATE (p. 667)
- STV_WLM_QUERY_TASK_STATE (p. 668)
- STV_WLM_SERVICE_CLASS_CONFIG (p. 669)
- STV_WLM_SERVICE_CLASS_STATE (p. 670)

STV_ACTIVE_CURSORS

STV ACTIVE CURSORS displays details for currently open cursors. For more information, see DECLARE (p. 328).

STV ACTIVE CURSORS is user visible. A user can only view cursors opened by that user. A superuser can view all cursors.

Table columns

<table>
<thead>
<tr>
<th>Column name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>userid</td>
<td>integer</td>
<td>ID of user who generated entry.</td>
</tr>
<tr>
<td>name</td>
<td>varchar</td>
<td>Cursor name.</td>
</tr>
<tr>
<td>xid</td>
<td>bigint</td>
<td>Transaction context.</td>
</tr>
<tr>
<td>pid</td>
<td>integer</td>
<td>Leader process running the query.</td>
</tr>
<tr>
<td>starttime</td>
<td>timestamp without time zone</td>
<td>Time when the cursor was declared.</td>
</tr>
</tbody>
</table>
STV_BLOCKLIST contains the number of 1 MB disk blocks that are used by each slice, table, or column in a database.

Use aggregate queries with STV_BLOCKLIST, as the following examples show, to determine the number of 1 MB disk blocks allocated per database, table, slice, or column. You can also use STV_PARTITIONS (p. 657) to view summary information about disk utilization.

STV_BLOCKLIST is superuser visible.

**Table columns**

<table>
<thead>
<tr>
<th>Column name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>row_count</td>
<td>bigint</td>
<td>Number of rows in the cursor result set.</td>
</tr>
<tr>
<td>byte_count</td>
<td>bigint</td>
<td>Number of bytes in the cursor result set.</td>
</tr>
<tr>
<td>fetched_rows</td>
<td>bigint</td>
<td>Number of rows currently fetched from the cursor result set.</td>
</tr>
<tr>
<td>slice</td>
<td>integer</td>
<td>Zero-based index for the column. Every table you create has three hidden columns appended to it: INSERT_XID, DELETE_XID, and ROW_ID (OID). A table with 3 user-defined columns contains 6 actual columns, and the user-defined columns are internally numbered as 0, 1, and 2. The INSERT_XID, DELETE_XID, and ROW_ID columns are numbered 3, 4, and 5, respectively, in this example.</td>
</tr>
<tr>
<td>col</td>
<td>integer</td>
<td>Table ID for the database table.</td>
</tr>
<tr>
<td>tbl</td>
<td>integer</td>
<td>ID for the data block.</td>
</tr>
<tr>
<td>blocknum</td>
<td>integer</td>
<td>Number of values contained on the block.</td>
</tr>
<tr>
<td>num_values</td>
<td>integer</td>
<td>For internal use.</td>
</tr>
<tr>
<td>extended_limits</td>
<td>integer</td>
<td>Minimum data value of the block. Stores first eight characters as 64-bit integer for non-numeric data. Used for disk scanning.</td>
</tr>
<tr>
<td>minvalue</td>
<td>bigint</td>
<td>Maximum data value of the block. Stores first eight characters as 64-bit integer for non-numeric data. Used for disk scanning.</td>
</tr>
<tr>
<td>maxvalue</td>
<td>bigint</td>
<td>Internal Amazon Redshift identifier for super block position on the disk.</td>
</tr>
<tr>
<td>sb_pos</td>
<td>integer</td>
<td>Whether or not the block is pinned into memory as part of pre-load. 0 = false; 1 = true. Default is false.</td>
</tr>
<tr>
<td>on_disk</td>
<td>integer</td>
<td>Whether or not the block is automatically stored on disk. 0 = false; 1 = true. Default is false.</td>
</tr>
<tr>
<td>modified</td>
<td>integer</td>
<td>Whether or not the block has been modified. 0 = false; 1 = true. Default is false.</td>
</tr>
<tr>
<td>Column name</td>
<td>Data type</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>-----------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>hdr_modified</td>
<td>integer</td>
<td>Whether or not the block header has been modified. 0 = false; 1 = true. Default is false.</td>
</tr>
<tr>
<td>unsorted</td>
<td>integer</td>
<td>Whether or not a block is unsorted. 0 = false; 1 = true. Default is true.</td>
</tr>
<tr>
<td>tombstone</td>
<td>integer</td>
<td>Whether or not a block is tombstoned. 0 = false; 1 = true. Default is false.</td>
</tr>
<tr>
<td>preferred_diskno</td>
<td>integer</td>
<td>Disk number that the block should be on, unless the disk has failed. Once the disk has been fixed, the block will move back to this disk.</td>
</tr>
<tr>
<td>temporary</td>
<td>integer</td>
<td>Whether or not the block contains temporary data, such as from a temporary table or intermediate query results. 0 = false; 1 = true. Default is false.</td>
</tr>
<tr>
<td>newblock</td>
<td>integer</td>
<td>Indicates whether or not a block is new (true) or was never committed to disk (false). 0 = false; 1 = true. Default is false.</td>
</tr>
<tr>
<td>num_readers</td>
<td>integer</td>
<td>Number of references on each block.</td>
</tr>
<tr>
<td>flags</td>
<td>integer</td>
<td>Internal Amazon Redshift flags for the block header.</td>
</tr>
</tbody>
</table>

**Sample queries**

STV_BLOCKLIST contains one row per allocated disk block, so a query that selects all the rows potentially returns a very large number of rows. We recommend using only aggregate queries with STV_BLOCKLIST.

The SVV_DISKUSAGE (p. 673) view provides similar information in a more user-friendly format; however, the following example demonstrates one use of the STV_BLOCKLIST table.

To determine the number of 1 MB blocks used by each column in the VENUE table, type the following query:

```sql
select col, count(*)
from stv_blocklist, stv_tbl_perm
where stv_blocklist.tbl = stv_tbl_perm.id
and stv_blocklist.slice = stv_tbl_perm.slice
and stv_tbl_perm.name = 'venue'
group by col
order by col;
```

This query returns the number of 1 MB blocks allocated to each column in the VENUE table, shown by the following sample data:

<table>
<thead>
<tr>
<th>col</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
</tr>
</tbody>
</table>

(8 rows)
The following query shows whether or not table data is actually distributed over all slices:

```sql
select trim(name) as table, stv_blocklist.slice, stv_tbl_perm.rows
from stv_blocklist, stv_tbl_perm
where stv_blocklist.tbl = stv_tbl_perm.id
and stv_tbl_perm.slice = stv_blocklist.slice
and stv_blocklist.id > 10000 and name not like '%#m%'
and name not like 'systable'
group by name, stv_blocklist.slice, stv_tbl_perm.rows
order by 3 desc;
```

This query produces the following sample output, showing the even data distribution for the table with the most rows:

<table>
<thead>
<tr>
<th>table</th>
<th>slice</th>
<th>rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>listing</td>
<td>13</td>
<td>10527</td>
</tr>
<tr>
<td>listing</td>
<td>14</td>
<td>10526</td>
</tr>
<tr>
<td>listing</td>
<td>8</td>
<td>10526</td>
</tr>
<tr>
<td>listing</td>
<td>9</td>
<td>10526</td>
</tr>
<tr>
<td>listing</td>
<td>7</td>
<td>10525</td>
</tr>
<tr>
<td>listing</td>
<td>4</td>
<td>10525</td>
</tr>
<tr>
<td>listing</td>
<td>17</td>
<td>10525</td>
</tr>
<tr>
<td>listing</td>
<td>11</td>
<td>10525</td>
</tr>
<tr>
<td>listing</td>
<td>5</td>
<td>10525</td>
</tr>
<tr>
<td>listing</td>
<td>18</td>
<td>10525</td>
</tr>
<tr>
<td>listing</td>
<td>12</td>
<td>10525</td>
</tr>
<tr>
<td>listing</td>
<td>3</td>
<td>10525</td>
</tr>
<tr>
<td>listing</td>
<td>10</td>
<td>10525</td>
</tr>
<tr>
<td>listing</td>
<td>2</td>
<td>10524</td>
</tr>
<tr>
<td>listing</td>
<td>15</td>
<td>10524</td>
</tr>
<tr>
<td>listing</td>
<td>16</td>
<td>10524</td>
</tr>
<tr>
<td>listing</td>
<td>6</td>
<td>10524</td>
</tr>
<tr>
<td>listing</td>
<td>19</td>
<td>10524</td>
</tr>
<tr>
<td>listing</td>
<td>1</td>
<td>10523</td>
</tr>
<tr>
<td>listing</td>
<td>0</td>
<td>10521</td>
</tr>
</tbody>
</table>

... (180 rows)

The following query determines whether any tombstoned blocks were committed to disk:

```sql
select slice, col, tbl, blocknum, newblock
from stv_blocklist
where tombstone > 0;
```

<table>
<thead>
<tr>
<th>slice</th>
<th>col</th>
<th>tbl</th>
<th>blocknum</th>
<th>newblock</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>0</td>
<td>101285</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>101285</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>101285</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>101285</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>101285</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>101285</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>101285</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
STV_CURSOR_CONFIGURATION

STV_CURSOR_CONFIGURATION displays cursor configuration constraints. For more information, see DECLARE (p. 328).

STV_CURSOR_CONFIGURATION is superuser visible.

Table columns

<table>
<thead>
<tr>
<th>Column name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>maximum_cursor_count</td>
<td>integer</td>
<td>Maximum number of cursors allowed concurrently.</td>
</tr>
<tr>
<td>maximum_cursor_size</td>
<td>integer</td>
<td>Maximum size of an individual cursor result set, in megabytes.</td>
</tr>
</tbody>
</table>

STV_EXEC_STATE

Use the STV_EXEC_STATE table to find out information about queries and query steps that are actively running on Amazon Redshift.

This information is usually used only to troubleshoot engineering issues. The views SVV_QUERY_STATE and SVL_QUERY_SUMMARY extract their information from STV_EXEC_STATE.

STV_EXEC_STATE is user visible.

Table columns

<table>
<thead>
<tr>
<th>Column name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>userid</td>
<td>integer</td>
<td>ID of user who generated entry.</td>
</tr>
<tr>
<td>query</td>
<td>integer</td>
<td>Query ID. Can be used to join various other system tables and views.</td>
</tr>
<tr>
<td>slice</td>
<td>integer</td>
<td>Data slice where the step executed.</td>
</tr>
<tr>
<td>segment</td>
<td>integer</td>
<td>Segment of the query that executed. A query segment is a series of steps.</td>
</tr>
<tr>
<td>step</td>
<td>integer</td>
<td>Step of the query segment that executed. A step is the smallest unit of query execution.</td>
</tr>
<tr>
<td>starttime</td>
<td>timestamp without time zone</td>
<td>Time that the step executed.</td>
</tr>
<tr>
<td>currenttime</td>
<td>timestamp without time zone</td>
<td>Current time.</td>
</tr>
<tr>
<td>tasknum</td>
<td>integer</td>
<td>Query task process that is assigned to the execute the step.</td>
</tr>
</tbody>
</table>
Sample queries

Rather than querying STV_EXEC_STATE directly, Amazon Redshift recommends querying SVL_QUERY_SUMMARY or SVV_QUERY_STATE to obtain the information in STV_EXEC_STATE in a more user-friendly format. See the SVL_QUERY_SUMMARY (p. 681) or SVV_QUERY_STATE (p. 679) table documentation for more details.

STV_INFLIGHT

Use the STV_INFLIGHT table to determine what queries are currently running on the database.

STV_INFLIGHT is user visible.

Table columns

<table>
<thead>
<tr>
<th>Column name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>userid</td>
<td>integer</td>
<td>ID of user who generated entry.</td>
</tr>
<tr>
<td>slice</td>
<td>integer</td>
<td>Slice where the query is running.</td>
</tr>
<tr>
<td>query</td>
<td>integer</td>
<td>Query ID. Can be used to join various other system tables and views.</td>
</tr>
<tr>
<td>Column name</td>
<td>Data type</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>-------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>label</td>
<td>character(30)</td>
<td>Either the name of the file used to run the query or a label defined with a SET QUERY_GROUP command. If the query is not file-based or the QUERY_GROUP parameter is not set, this field is blank.</td>
</tr>
<tr>
<td>xid</td>
<td>bigint</td>
<td>Transaction ID.</td>
</tr>
<tr>
<td>pid</td>
<td>integer</td>
<td>Process ID. All of the queries in a session are run in the same process, so this value remains constant if you run a series of queries in the same session. You can use this column to join to the STL_ERROR (p. 596) table.</td>
</tr>
<tr>
<td>starttime</td>
<td>timestamp without timezone</td>
<td>Time that the query started.</td>
</tr>
<tr>
<td>text</td>
<td>character(100)</td>
<td>Query text, truncated to 100 characters if the statement exceeds that limit.</td>
</tr>
<tr>
<td>suspended</td>
<td>integer</td>
<td>Whether the query is suspended or not. 0 = false; 1 = true.</td>
</tr>
<tr>
<td>insert_pristine</td>
<td>integer</td>
<td>Whether write queries are/were able to run while the current query is/was running. 1 = no write queries allowed. 0 = write queries allowed. This column is intended for use in debugging.</td>
</tr>
</tbody>
</table>

**Sample queries**

To view all active queries currently running on the database, type the following query:

```sql
select * from stv_inflight;
```

The sample output below shows two queries currently running, including the STV_INFLIGHT query itself and a query that was run from a script called `avgwait.sql`:

```sql
select slice, query, trim(label) querylabel, pid, starttime, substring(text,1,20) querytext
from stv_inflight;
```

<table>
<thead>
<tr>
<th>slice</th>
<th>query</th>
<th>querylabel</th>
<th>pid</th>
<th>starttime</th>
<th>querytext</th>
</tr>
</thead>
<tbody>
<tr>
<td>1011</td>
<td>20</td>
<td></td>
<td>646</td>
<td>2012-01-26 13:23:15.645503</td>
<td>select slice, query,</td>
</tr>
</tbody>
</table>

**STV_LOAD_STATE**

Use the STV_LOAD_STATE table to find information about current state of ongoing COPY statements. The COPY command updates this table after every million records are loaded.

STV_LOAD_STATE is user visible.
### Table columns

<table>
<thead>
<tr>
<th>Column name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>userid</td>
<td>integer</td>
<td>ID of user who generated entry.</td>
</tr>
<tr>
<td>session</td>
<td>integer</td>
<td>Session PID of process doing the load.</td>
</tr>
<tr>
<td>query</td>
<td>integer</td>
<td>Query ID. Can be used to join various other system tables and views.</td>
</tr>
<tr>
<td>slice</td>
<td>integer</td>
<td>Node slice number.</td>
</tr>
<tr>
<td>pid</td>
<td>integer</td>
<td>Process ID. All of the queries in a session are run in the same process, so this value remains constant if you run a series of queries in the same session.</td>
</tr>
<tr>
<td>recordtime</td>
<td>timestamp</td>
<td>Time the record is logged.</td>
</tr>
<tr>
<td>bytes_to_load</td>
<td>bigint</td>
<td>Total number of bytes to be loaded by this slice. This is 0 if the data being loaded is compressed</td>
</tr>
<tr>
<td>bytes_loaded</td>
<td>bigint</td>
<td>Number of bytes loaded by this slice. If the data being loaded is compressed, this is the number of bytes loaded after the data is uncompressed.</td>
</tr>
<tr>
<td>bytes_to_load_compressed</td>
<td>bigint</td>
<td>Total number of bytes of compressed data to be loaded by this slice. This is 0 if the data being loaded is not compressed.</td>
</tr>
<tr>
<td>bytes_loaded_compressed</td>
<td>bigint</td>
<td>Number of bytes of compressed data loaded by this slice. This is 0 if the data being loaded is not compressed.</td>
</tr>
<tr>
<td>lines</td>
<td>integer</td>
<td>Number of lines loaded by this slice.</td>
</tr>
<tr>
<td>num_files</td>
<td>integer</td>
<td>Number of files to be loaded by this slice.</td>
</tr>
<tr>
<td>num_files_complete</td>
<td>integer</td>
<td>Number of files loaded by this slice.</td>
</tr>
<tr>
<td>current_file</td>
<td>character(256)</td>
<td>Name of the file being loaded by this slice.</td>
</tr>
<tr>
<td>pct_complete</td>
<td>integer</td>
<td>Percentage of data load completed by this slice.</td>
</tr>
</tbody>
</table>

### Sample query

To view the progress of each slice for a COPY command, type the following query. This example uses the `PG_LAST_COPY_ID()` function to retrieve information for the last COPY command.

```sql
select slice , bytes_loaded, bytes_to_load , pct_complete from stv_load_state where query = pg_last_copy_id();
```

<table>
<thead>
<tr>
<th>slice</th>
<th>bytes_loaded</th>
<th>bytes_to_load</th>
<th>pct_complete</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>12840898</td>
<td>39104640</td>
<td>32</td>
</tr>
</tbody>
</table>

(2 rows)
STV_LOCKS

Use the STV_LOCKS table to view any current updates on tables in the database.

Amazon Redshift locks tables to prevent two users from updating the same table at the same time. While the STV_LOCKS table shows all current table updates, query the STL_TR_CONFLICT (p. 635) table to see a log of lock conflicts.

STV_LOCKS is superuser visible.

Table columns

<table>
<thead>
<tr>
<th>Column name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>table_id</td>
<td>bigint</td>
<td>Table ID for the table acquiring the lock.</td>
</tr>
<tr>
<td>last_commit</td>
<td>timestamp without time zone</td>
<td>Timestamp for the last commit in the table.</td>
</tr>
<tr>
<td>last_update</td>
<td>timestamp without time zone</td>
<td>Timestamp for the last update for the table.</td>
</tr>
<tr>
<td>lock_owner</td>
<td>bigint</td>
<td>Transaction ID associated with the lock.</td>
</tr>
<tr>
<td>lock_owner_pid</td>
<td>bigint</td>
<td>Process ID associated with the lock.</td>
</tr>
<tr>
<td>lock_owner_start_ts</td>
<td>timestamp without time zone</td>
<td>Timestamp for the transaction start time.</td>
</tr>
<tr>
<td>lock_owner_end_ts</td>
<td>timestamp without time zone</td>
<td>Timestamp for the transaction end time.</td>
</tr>
<tr>
<td>lock_status</td>
<td>character (22)</td>
<td>Status of the process either waiting for or holding a lock.</td>
</tr>
</tbody>
</table>

Sample query

To view all locks taking place in current transactions, type the following command:

```
select table_id, last_update, lock_owner, lock_owner_pid from stv_locks;
```

This query returns the following sample output, which displays three locks currently in effect:

```
table_id |        last_update         | lock_owner | lock_owner_pid
----------+----------------------------+------------+----------------
100004  | 2008-12-23 10:08:48.882319 |       1043 |           5656
100003  | 2008-12-23 10:08:48.779543 |       1043 |           5656
100140  | 2008-12-23 10:08:48.021576 |       1043 |           5656
(3 rows)
```

STV_PARTITIONS

Use the STV_PARTITIONS table to find out the disk speed performance and disk utilization for Amazon Redshift.
STV_PARTITIONS contains one row per node per logical disk partition, or slice.

STV_PARTITIONS is superuser visible.

### Table rows

<table>
<thead>
<tr>
<th>Row name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
<td>integer</td>
<td>Disk node that owns the partition.</td>
</tr>
<tr>
<td>host</td>
<td>integer</td>
<td>Node that is physically attached to the partition.</td>
</tr>
<tr>
<td>diskno</td>
<td>integer</td>
<td>Disk containing the partition.</td>
</tr>
<tr>
<td>part_begin</td>
<td>bigint</td>
<td>Offset of the partition. Raw devices are logically partitioned to open space for mirror blocks.</td>
</tr>
<tr>
<td>part_end</td>
<td>bigint</td>
<td>End of the partition.</td>
</tr>
<tr>
<td>used</td>
<td>integer</td>
<td>Number of 1 MB disk blocks currently in use on the partition.</td>
</tr>
<tr>
<td>tossed</td>
<td>integer</td>
<td>Number of blocks that are ready to be deleted but are not yet removed because it is not safe to free their disk addresses. If the addresses were freed immediately, a pending transaction could write to the same location on disk. Therefore, these tossed blocks are released as of the next commit. Disk blocks might be marked as tossed, for example, when a table column is dropped, during INSERT operations, or during disk-based query operations.</td>
</tr>
<tr>
<td>capacity</td>
<td>integer</td>
<td>Total capacity of the partition in 1 MB disk blocks.</td>
</tr>
<tr>
<td>reads</td>
<td>bigint</td>
<td>Number of reads that have occurred since the last Amazon Redshift xstart.</td>
</tr>
<tr>
<td>writes</td>
<td>bigint</td>
<td>Number of writes that have occurred since the last Amazon Redshift xstart.</td>
</tr>
<tr>
<td>seek_forward</td>
<td>integer</td>
<td>Number of times that a request is not for the subsequent address given the previous request address.</td>
</tr>
<tr>
<td>seek_back</td>
<td>integer</td>
<td>Number of times that a request is not for the previous address given the subsequent address.</td>
</tr>
<tr>
<td>is_san</td>
<td>integer</td>
<td>Whether the partition belongs to a SAN. Valid values are 0 (false) or 1 (true).</td>
</tr>
<tr>
<td>failed</td>
<td>integer</td>
<td>Whether the device has been marked as failed. Valid values are 0 (false) or 1 (true).</td>
</tr>
<tr>
<td>mbps</td>
<td>integer</td>
<td>Disk speed in megabytes per second.</td>
</tr>
<tr>
<td>mount</td>
<td>character(256)</td>
<td>Directory path to the device.</td>
</tr>
</tbody>
</table>

### Sample query

The following query returns the disk space used and capacity, in 1 MB disk blocks, and calculates disk utilization as a percentage of raw disk space. The raw disk space includes space that is reserved by Amazon Redshift for internal use, so it is larger than the nominal disk capacity, which is the amount of disk space available to the user. The Percentage of Disk Space Used metric on the Performance tab of the Amazon Redshift Management Console reports the percentage of nominal disk capacity used by
your cluster. We recommend that you monitor the **Percentage of Disk Space Used** metric to maintain your usage within your cluster's nominal disk capacity.

**Important**

We strongly recommend that you do not exceed your cluster's nominal disk capacity. While it might be technically possible under certain circumstances, exceeding your nominal disk capacity decreases your cluster's fault tolerance and increases your risk of losing data.

This example was run on a two-node cluster with six logical disk partitions per node. Space is being used very evenly across the disks, with approximately 25% of each disk in use.

```sql
select owner, host, diskno, used, capacity, 
(used-tossed)/capacity::numeric *100 as pctused 
from stv_partitions order by owner;
```

<table>
<thead>
<tr>
<th>owner</th>
<th>host</th>
<th>diskno</th>
<th>used</th>
<th>capacity</th>
<th>pctused</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>236480</td>
<td>949954</td>
<td>24.9</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>236420</td>
<td>949954</td>
<td>24.9</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>2</td>
<td>236440</td>
<td>949954</td>
<td>24.9</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>235150</td>
<td>949954</td>
<td>24.8</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>237100</td>
<td>949954</td>
<td>25.0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>237090</td>
<td>949954</td>
<td>25.0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>236310</td>
<td>949954</td>
<td>24.9</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>236300</td>
<td>949954</td>
<td>24.9</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>236320</td>
<td>949954</td>
<td>24.9</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>2</td>
<td>237910</td>
<td>949954</td>
<td>25.0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>235640</td>
<td>949954</td>
<td>24.8</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>235380</td>
<td>949954</td>
<td>24.8</td>
</tr>
</tbody>
</table>

(12 rows)

**STV_RECENTS**

Use the STV_RECENTS table to find out information about the currently active and recently run queries against a database.

STV_RECENTS is user visible.

**Table columns**

<table>
<thead>
<tr>
<th>Column name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>userid</td>
<td>integer</td>
<td>ID of user who generated entry.</td>
</tr>
<tr>
<td>status</td>
<td>character(20)</td>
<td>Query status. Valid values are <strong>Running</strong>, <strong>Done</strong>.</td>
</tr>
<tr>
<td>starttime</td>
<td>timestamp without time zone</td>
<td>Time that the query started.</td>
</tr>
<tr>
<td>duration</td>
<td>integer</td>
<td>Number of microseconds since the session started.</td>
</tr>
<tr>
<td>user_name</td>
<td>character(50)</td>
<td>User name who ran the process.</td>
</tr>
<tr>
<td>db_name</td>
<td>character(50)</td>
<td>Name of the database.</td>
</tr>
</tbody>
</table>
Sample queries

To determine what queries are currently running against the database, type the following query:

```sql
select user_name, db_name, pid, query
from stv_recent
where status = 'Running';
```

The sample output below shows a single query running on the TICKIT database:

```plaintext
user_name | db_name | pid | query
----------|---------|-----|------------------------
dwuser    | tickit  | 19996 | select venuename, venueseats from venue where venueseats > 50000 order by venueseats desc;
```

The following example returns a list of queries (if any) that are running or waiting in queue to be executed:

```sql
select * from stv_recent
where status<>'Done';
```

```plaintext
status | starttime | duration | user_name | db_name | query | pid
-------|-----------|----------|-----------|---------|-------|------
Running| 2010-04-21 16:11... | 281566454 | dwuser | tickit | select ... | 23347
```

This query does not return results unless you are running a number of concurrent queries and some of those queries are in queue.

The following example extends the previous example. In this case, queries that are truly "in flight" (running, not waiting) are excluded from the result:

```sql
select * from stv_recent
where status<>'Done'
and pid not in (select pid from stv_inflight);
```

...
Table columns

<table>
<thead>
<tr>
<th>Column name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>node</td>
<td>integer</td>
<td>Cluster node where the slice is located.</td>
</tr>
<tr>
<td>slice</td>
<td>integer</td>
<td>Data slice.</td>
</tr>
</tbody>
</table>

Sample query

To view which cluster nodes are managing which slices, type the following query:

```sql
select * from stv_slices;
```

This query returns the following sample output:

<table>
<thead>
<tr>
<th>node</th>
<th>slice</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

(4 rows)

STV_SESSIONS

Use the STV_SESSIONS table to view information about the active user sessions for Amazon Redshift.

To view session history, use the STL_SESSIONS (p. 631) table instead of STV_SESSIONS.

STV_SESSIONS is user visible.

Table columns

<table>
<thead>
<tr>
<th>Column name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>starttime</td>
<td>timestamp without time zone</td>
<td>Time that the session started.</td>
</tr>
<tr>
<td>process</td>
<td>integer</td>
<td>Process ID for the session.</td>
</tr>
<tr>
<td>user_name</td>
<td>character(50)</td>
<td>User associated with the session.</td>
</tr>
<tr>
<td>db_name</td>
<td>character(50)</td>
<td>Name of the database associated with the session.</td>
</tr>
</tbody>
</table>

Sample queries

To perform a quick check to see if any other users are currently logged into Amazon Redshift, type the following query:
select count(*)
from stv_sessions;

If the result is greater than one, then at least one other user is currently logged into the database.

To view all active sessions for Amazon Redshift, type the following query:

```sql
select *
from stv_sessions;
```

The sample query output below shows three active sessions currently running on Amazon Redshift:

<table>
<thead>
<tr>
<th>starttime</th>
<th>process</th>
<th>user_name</th>
<th>db_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>2008-08-06 08:54:20.50</td>
<td>19829</td>
<td>dwuser</td>
<td>dev</td>
</tr>
<tr>
<td>2008-08-06 08:56:34.50</td>
<td>20279</td>
<td>dwuser</td>
<td>dev</td>
</tr>
<tr>
<td>2008-08-06 08:55:00.50</td>
<td>19996</td>
<td>dwuser</td>
<td>tickit</td>
</tr>
</tbody>
</table>

(3 rows)

### STV_TBL_PERM

The STV_TBL_PERM table contains information about the permanent tables in Amazon Redshift, including temporary tables created by a user for the current session. STV_TBL_PERM contains information for all tables in all databases.

This table differs from STV_TBL_TRANS (p. 664), which contains information about transient database tables that the system creates during query processing.

STV_TBL_PERM is superuser visible.

### Table columns

<table>
<thead>
<tr>
<th>Column name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>slice</td>
<td>integer</td>
<td>Data slice allocated to the table.</td>
</tr>
<tr>
<td>id</td>
<td>integer</td>
<td>Table ID.</td>
</tr>
<tr>
<td>name</td>
<td>character(72)</td>
<td>Table name.</td>
</tr>
<tr>
<td>rows</td>
<td>bigint</td>
<td>Number of data rows in the slice.</td>
</tr>
<tr>
<td>sorted_rows</td>
<td>bigint</td>
<td>Number of rows in the slice that are already sorted on disk. If this number does not match the ROWS number, vacuum the table to resort the rows.</td>
</tr>
<tr>
<td>temp</td>
<td>integer</td>
<td>Whether or not the table is a temporary table. 0 = false; 1 = true.</td>
</tr>
<tr>
<td>db_id</td>
<td>integer</td>
<td>ID of the database where the table was created.</td>
</tr>
</tbody>
</table>

### Sample queries

The following query returns a list of distinct table IDs and names:

```sql
```

API Version 2012-12-01

662
Other system tables use table IDs, so knowing which table ID corresponds to a certain table can be very useful. In this example, SELECT DISTINCT is used to remove the duplicates (tables are distributed across multiple slices).

To determine the number of blocks used by each column in the VENUE table, type the following query:

```sql
select col, count(*)
from stv_blocklist, stv_tbl_perm
where stv_blocklist.tbl = stv_tbl_perm.id
and stv_blocklist.slice = stv_tbl_perm.slice
and stv_tbl_perm.name = 'venue'
group by col
order by col;
```

```
col | count
-----+-------
0 | 8
1 | 8
2 | 8
3 | 8
4 | 8
5 | 8
6 | 8
7 | 8
(8 rows)
```

### Usage notes

The ROWS column includes counts of deleted rows that have not been vacuumed (or have been vacuumed but with the SORT ONLY option). Therefore, the SUM of the ROWS column in the STV_TBL_PERM table might not match the COUNT(*) result when you query a given table directly. For example, if 2 rows are deleted from VENUE, the COUNT(*) result is 200 but the SUM(ROWS) result is still 202:

```sql
delete from venue
where venueid in (1,2);

select count(*) from venue;
count
-------
200
(1 row)
```
select trim(name) tablename, sum(rows)  
from stv_tbl_perm where name='venue' group by name;

<table>
<thead>
<tr>
<th>tablename</th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>venue</td>
<td>202</td>
</tr>
</tbody>
</table>

(1 row)

To "correct" the results of the STV_TBL_PERM query, run a "full" vacuum the VENUE table:

vacuum venue;

select trim(name) tablename, sum(rows)  
from stv_tbl_perm  
where name='venue'  
group by name;

<table>
<thead>
<tr>
<th>tablename</th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>venue</td>
<td>200</td>
</tr>
</tbody>
</table>

(1 row)

### STV_TBL_TRANS

Use the STV_TBL_TRANS table to find out information about the transient database tables that are currently in memory.

Transient tables are typically temporary row sets that are used as intermediate results while a query runs. STV_TBL_TRANS differs from STV_TBL_PERM (p. 662) in that STV_TBL_PERM contains information about permanent database tables.

STV_TBL_TRANS is superuser visible.

#### Table columns

<table>
<thead>
<tr>
<th>Column name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>slice</td>
<td>integer</td>
<td>Data slice allocated to the table.</td>
</tr>
<tr>
<td>id</td>
<td>integer</td>
<td>Table ID.</td>
</tr>
<tr>
<td>rows</td>
<td>bigint</td>
<td>Number of data rows in the table.</td>
</tr>
<tr>
<td>size</td>
<td>bigint</td>
<td>Number of bytes allocated to the table.</td>
</tr>
<tr>
<td>query_id</td>
<td>bigint</td>
<td>Query ID.</td>
</tr>
<tr>
<td>ref_cnt</td>
<td>integer</td>
<td>Number of references.</td>
</tr>
<tr>
<td>from_suspended</td>
<td>integer</td>
<td>Whether or not this table was created during a query that is now suspended.</td>
</tr>
<tr>
<td>prep_swap</td>
<td>integer</td>
<td>Whether or not this transient table is prepared to swap to disk if needed. (The swap will only occur in situations where memory is low.)</td>
</tr>
</tbody>
</table>
Sample queries

To view transient table information for a query with a query ID of 90, type the following command:

```sql
select slice, id, rows, size, query_id, ref_cnt
from stv_tbl_trans
where query_id = 90;
```

This query returns the transient table information for query 90, as shown in the following sample output:

<table>
<thead>
<tr>
<th>slice</th>
<th>id</th>
<th>rows</th>
<th>size</th>
<th>query_id</th>
<th>ref_cnt</th>
<th>from_cnt</th>
<th>suspended</th>
<th>prep_swap</th>
</tr>
</thead>
<tbody>
<tr>
<td>1013</td>
<td>95</td>
<td>0</td>
<td>0</td>
<td>90</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>96</td>
<td>0</td>
<td>0</td>
<td>90</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>96</td>
<td>0</td>
<td>0</td>
<td>90</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>17</td>
<td>96</td>
<td>0</td>
<td>0</td>
<td>90</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>96</td>
<td>0</td>
<td>0</td>
<td>90</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>96</td>
<td>0</td>
<td>0</td>
<td>90</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1013</td>
<td>99</td>
<td>0</td>
<td>0</td>
<td>90</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>96</td>
<td>0</td>
<td>0</td>
<td>90</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>96</td>
<td>0</td>
<td>0</td>
<td>90</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>19</td>
<td>96</td>
<td>0</td>
<td>0</td>
<td>90</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>96</td>
<td>0</td>
<td>0</td>
<td>90</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1013</td>
<td>98</td>
<td>0</td>
<td>0</td>
<td>90</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>96</td>
<td>0</td>
<td>0</td>
<td>90</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>96</td>
<td>0</td>
<td>0</td>
<td>90</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1013</td>
<td>96</td>
<td>0</td>
<td>0</td>
<td>90</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>96</td>
<td>0</td>
<td>0</td>
<td>90</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>96</td>
<td>0</td>
<td>0</td>
<td>90</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>96</td>
<td>0</td>
<td>0</td>
<td>90</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>18</td>
<td>96</td>
<td>0</td>
<td>0</td>
<td>90</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

In this example, you can see that the query data involves tables 95, 96, and 98. Because zero bytes are allocated to this table, this query can run in memory.

**STV_WLM_CLASSIFICATION_CONFIG**

Contains the current classification rules for WLM.

STV_WLM_CLASSIFICATION_CONFIG is superuser visible.

**Table columns**

<table>
<thead>
<tr>
<th>Column name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>integer</td>
<td>Service class ID.</td>
</tr>
<tr>
<td>condition</td>
<td>character(128)</td>
<td>Query type. Valid query types are read, write, system, or any.</td>
</tr>
<tr>
<td>action_seq</td>
<td>integer</td>
<td>Zero-based index specifying the action position in the order of actions.</td>
</tr>
</tbody>
</table>
### Sample query

The following query returns the current classification rules.

```sql
select id, trim(condition), action_seq, trim(action), action_service_class
from stv_wlm_classification_config
order by 1;
```

<table>
<thead>
<tr>
<th>id</th>
<th>btrim</th>
<th>action_seq</th>
<th>btrim</th>
<th>action_service_class</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(querytype: system)</td>
<td>0</td>
<td>assign</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>(querytype: read)</td>
<td>0</td>
<td>assign</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>(querytype: read)</td>
<td>1</td>
<td>assign</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>(querytype: write)</td>
<td>0</td>
<td>assign</td>
<td>3</td>
</tr>
</tbody>
</table>

(4 rows)

### STV_WLM_QUERY_QUEUE_STATE

Records the current state of the query queues for the service classes.

**STV_WLM_QUERY_QUEUE_STATE** is user visible.

#### Table columns

<table>
<thead>
<tr>
<th>Column name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>service_class</td>
<td>integer</td>
<td>ID for the service class. Service classes are defined in the WLM configuration.</td>
</tr>
<tr>
<td>position</td>
<td>integer</td>
<td>Position of the query in the queue. The query with the smallest position value runs next.</td>
</tr>
<tr>
<td>task</td>
<td>integer</td>
<td>ID used to track a query through the workload manager. Can be associated with multiple query IDs. If a query is restarted, the query is assigned a new query ID but not a new task ID.</td>
</tr>
<tr>
<td>query</td>
<td>integer</td>
<td>Query ID. If a query is restarted, the query is assigned a new query ID but not a new task ID.</td>
</tr>
<tr>
<td>slot_count</td>
<td>integer</td>
<td>Number of WLM query slots.</td>
</tr>
<tr>
<td>start_time</td>
<td>timestamp without timezone</td>
<td>Time that the query entered the queue.</td>
</tr>
<tr>
<td>queue_time</td>
<td>bigint</td>
<td>Number of microseconds that the query has been in the queue.</td>
</tr>
</tbody>
</table>
Sample query

The WLM configuration used for this example has three service classes:

- A system table service class (Service Class 1)
- An evictable service class for user queries with a threshold of 2000000 microseconds (2 seconds) with three query tasks (Service Class 2)
- A non-evictable service class for user queries with one query task (Service Class 3)

The following query shows the queries in the queue for service class 3. These are the queries that have been evicted from service class 2 and placed in the queue for service class 3:

```
select * from stv_wlm_query_queue_state
where service_class=3
order by 1, 2, 3, 4, 5;
```

This query returns the following sample output:

```
<table>
<thead>
<tr>
<th>service_class</th>
<th>position</th>
<th>task</th>
<th>query</th>
<th>start_time</th>
<th>queue_time</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>0</td>
<td>455</td>
<td>476</td>
<td>2010-10-06 13:18:24</td>
<td>20937257</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>456</td>
<td>478</td>
<td>2010-10-06 13:18:26</td>
<td>18350191</td>
</tr>
</tbody>
</table>
(2 rows)
```

**STV_WLM_QUERY_STATE**

Records the current state of queries being tracked by WLM.

*STV_WLM_QUERY_STATE* is user visible.

**Table columns**

<table>
<thead>
<tr>
<th>Column name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xid</td>
<td>integer</td>
<td>Transaction ID of the query or subquery.</td>
</tr>
<tr>
<td>task</td>
<td>integer</td>
<td>ID used to track a query through the workload manager. Can be associated with multiple query IDs. If a query is restarted, the query is assigned a new query ID but not a new task ID.</td>
</tr>
<tr>
<td>query</td>
<td>integer</td>
<td>Query ID. If a query is restarted, the query is assigned a new query ID but not a new task ID.</td>
</tr>
<tr>
<td>service_class</td>
<td>integer</td>
<td>ID for the service class. Service classes are defined in the WLM configuration.</td>
</tr>
<tr>
<td>slot_count</td>
<td>integer</td>
<td>Number of WLM query slots.</td>
</tr>
<tr>
<td>wlm_start_time</td>
<td>timestamp without time zone</td>
<td>Time that the query entered the system table queue or short query queue.</td>
</tr>
<tr>
<td>state</td>
<td>character(16)</td>
<td>Current state of the query or subquery.</td>
</tr>
</tbody>
</table>
### Sample query

The following query returns records for all currently executing queries:

```sql
select xid, query, trim(state), queue_time, exec_time
from stv_wlm_query_state
where state like '%Executing%';
```

This query returns the following sample output:

```
xid  | query |   btrim   | queue_time | exec_time
-----+-------+-----------+------------+-----------
2477 |   498 | Executing |          0 |    155981
(1 row)
```

### STV_WLM_QUERY_TASK_STATE

Contains the current state of service class query tasks.

STV_WLM_QUERY_TASK_STATE is user visible.

#### Table columns

<table>
<thead>
<tr>
<th>Column name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>service_class</td>
<td>integer</td>
<td>ID for the service class. Service classes are defined in the WLM configuration.</td>
</tr>
<tr>
<td>task</td>
<td>integer</td>
<td>ID used to track a query through the workload manager. Can be associated with multiple query IDs. If a query is restarted, the query is assigned a new query ID but not a new task ID.</td>
</tr>
<tr>
<td>query</td>
<td>integer</td>
<td>Query ID. If a query is restarted, the query is assigned a new query ID but not a new task ID.</td>
</tr>
<tr>
<td>slot_count</td>
<td>integer</td>
<td>Number of WLM query slots.</td>
</tr>
<tr>
<td>start_time</td>
<td>timestamp</td>
<td>Time that the query began executing.</td>
</tr>
<tr>
<td>exec_time</td>
<td>bigint</td>
<td>Number of microseconds that the query has been executing.</td>
</tr>
</tbody>
</table>

### Sample query

The WLM configuration used for this example has three service classes:
• A system table service class (service class 1)
• An evictable service class for user queries with a threshold of 2000000 microseconds (2 seconds) with three query tasks (service class 2)
• A non-evictable service class for user queries with one query task (service class 3)

The following sample query shows all queries currently associated with service class 3:

```sql
select * from stv_wlm_query_task_state
where service_class=3;
```

This query returns the following sample output:

<table>
<thead>
<tr>
<th>service_class</th>
<th>task</th>
<th>query</th>
<th>start_time</th>
<th>exec_time</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>466</td>
<td>491</td>
<td>2010-10-06 13:29:23.063787</td>
<td>357618748</td>
</tr>
</tbody>
</table>

(1 row)

The results show that one currently running query has been evicted from service class 2 and assigned to service class 3.

**STV_WLM_SERVICE_CLASS_CONFIG**

Records the service class configurations for WLM.

STV_WLM_SERVICE_CLASS_CONFIG is superuser visible.

**Table columns**

<table>
<thead>
<tr>
<th>Column name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>service_class</td>
<td>integer</td>
<td>ID for the service class. Service classes are defined in the WLM configuration.</td>
</tr>
<tr>
<td>queueing_strategy</td>
<td>character(32)</td>
<td>Queueing strategy used by the service class. Valid value is FIFO Queue Policy (first in, first out).</td>
</tr>
<tr>
<td>num_query_tasks</td>
<td>integer</td>
<td>Number of concurrent tasks that can run in the service class.</td>
</tr>
<tr>
<td>evictable</td>
<td>character(8)</td>
<td>Indicates whether or not the queries in the service class are evictable. Valid values are true or false.</td>
</tr>
<tr>
<td>eviction_threshold</td>
<td>bigint</td>
<td>Number of microseconds that the query can execute before being evicted and reassigned to the next service class.</td>
</tr>
<tr>
<td>query_working_mem</td>
<td>integer</td>
<td>Overrides the configuration setting for the amount of working memory assigned to a query for the service class. Working memory stores intermediate query results from hashes, sorts, and aggregates</td>
</tr>
<tr>
<td>min_step_mem</td>
<td>integer</td>
<td>Overrides the configuration setting for the minimum amount of memory assigned to a query step for the service class. This parameter specifies the minimum amount of memory per node (in MB) that Amazon Redshift can assign to a step in a multi-step query.</td>
</tr>
<tr>
<td>Column name</td>
<td>Data type</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------</td>
<td>--------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>name</td>
<td>character(64)</td>
<td>Comment about the state of the service class.</td>
</tr>
<tr>
<td>max_execution_time</td>
<td>bigint</td>
<td>Number of microseconds that the query can execute before being terminated.</td>
</tr>
<tr>
<td>user_group_wild_card</td>
<td>Boolean</td>
<td>If TRUE, the WLM queue treats an asterisk (*) as a wildcard character in user group strings in the WLM configuration. The default is FALSE.</td>
</tr>
<tr>
<td>query_group_wild_card</td>
<td>Boolean</td>
<td>If TRUE, the WLM queue treats an asterisk (*) as a wildcard character in query group strings in the WLM configuration. The default is FALSE.</td>
</tr>
</tbody>
</table>

### Sample query

The following query displays the current service class configuration:

```sql
select service_class, num_query_tasks, evictable, eviction_threshold, name
from stv_wlm_service_class_config
order by 1, 2, 3, 4, 5;
```

This query returns the following sample output for a configuration with three service classes:

```
<table>
<thead>
<tr>
<th>service_class</th>
<th>num_query_tasks</th>
<th>evictable</th>
<th>eviction_threshold</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>false</td>
<td>0</td>
<td>System table query service class</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>true</td>
<td>2000000</td>
<td>short read service class</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>false</td>
<td>0</td>
<td>write long read service class</td>
</tr>
</tbody>
</table>
```

### STV_WLM_SERVICE_CLASS_STATE

Contains the current state of the service classes.

STV_WLM_SERVICE_CLASS_STATE is superuser visible.

### Table columns

<table>
<thead>
<tr>
<th>Column name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>service_class</td>
<td>integer</td>
<td>ID for the service class. Service classes are defined in the WLM configuration.</td>
</tr>
<tr>
<td>num_queued_queries</td>
<td>integer</td>
<td>Number of queries currently in the queue.</td>
</tr>
<tr>
<td>num_executing_queries</td>
<td>integer</td>
<td>Number of queries currently executing.</td>
</tr>
<tr>
<td>num_serviced_queries</td>
<td>integer</td>
<td>Number of queries that have ever been in the service class.</td>
</tr>
<tr>
<td>Column name</td>
<td>Data type</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------</td>
<td>-----------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>num_executed_queries</td>
<td>integer</td>
<td>Number of queries to have executed since Amazon Redshift was initialized.</td>
</tr>
<tr>
<td>num_restarted_queries</td>
<td>integer</td>
<td>Number of queries that have restarted since Amazon Redshift was initialized.</td>
</tr>
</tbody>
</table>

**Sample query**

The following query shows the number of queries that are either executing or have finished executing by service class.

```sql
select service_class, num_executing_queries, num_executed_queries
from stv_wlm_service_class_state
order by service_class;
```

This query returns the following sample output for a three service class configuration:

<table>
<thead>
<tr>
<th>service_class</th>
<th>num_executing_queries</th>
<th>num_executed_queries</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>222</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>135</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>39</td>
</tr>
</tbody>
</table>

(3 rows)

**System views**

System views contain a subset of data found in several of the STL and STV system tables. These views provide quicker and easier access to commonly queried data found in those tables.

**Note**
The SVL_QUERY_SUMMARY view only contains information about queries executed by Amazon Redshift, not other utility and DDL commands. For a complete listing and information on all statements executed by Amazon Redshift, including DDL and utility commands, you can query the SVL_STATEMENTTEXT view.

**Topics**

- SVL_COMPILE (p. 672)
- SVV_DISKUSAGE (p. 673)
- SVL_QERROR (p. 675)
- SVL_QLOG (p. 675)
- SVV_QUERY_INFLIGHT (p. 676)
- SVL_QUERY_REPORT (p. 677)
- SVV_QUERY_STATE (p. 679)
- SVL_QUERY_SUMMARY (p. 681)
- SVL_STATEMENTTEXT (p. 683)
- SVV_VACUUM_PROGREES (p. 684)
SVL_COMPILE

Records compile time and location for each query segment of queries.

SVL_COMPILE is visible to all users.

Table rows

<table>
<thead>
<tr>
<th>Row name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>userid</td>
<td>integer</td>
<td>ID of the user who generated the entry.</td>
</tr>
<tr>
<td>xid</td>
<td>bigint</td>
<td>Transaction ID associated with the statement.</td>
</tr>
<tr>
<td>pid</td>
<td>integer</td>
<td>Process ID associated with the statement.</td>
</tr>
<tr>
<td>query</td>
<td>integer</td>
<td>Query ID. Can be used to join various other system tables and views.</td>
</tr>
<tr>
<td>segment</td>
<td>integer</td>
<td>The query segment to be compiled.</td>
</tr>
<tr>
<td>locus</td>
<td>integer</td>
<td>Location where the segment executes. 1 if on a computer node and 2 if on the leader node.</td>
</tr>
<tr>
<td>starttime</td>
<td>timestamp</td>
<td>Time in UTC that the compile started.</td>
</tr>
<tr>
<td>endtime</td>
<td>timestamp</td>
<td>Time in UTC that the compile ended.</td>
</tr>
<tr>
<td>compile</td>
<td>integer</td>
<td>0 if the compile was reused, 1 if the segment was compiled.</td>
</tr>
</tbody>
</table>

Sample queries

In this example, queries 35878 and 35879 executed the same SQL statement. The compile column for query 35878 shows 1 for four query segments, which indicates that the segments were compiled. Query 35879 shows 0 in the compile column for every segment, indicating that the segments did not need to be compiled again.

```sql
select userid, xid, pid, query, segment, locus,
       datediff(ms, starttime, endtime) as duration, compile
from svl_compile
where query = 35878 or query = 35879
order by query, segment;
```

<table>
<thead>
<tr>
<th>userid</th>
<th>xid</th>
<th>pid</th>
<th>query</th>
<th>segment</th>
<th>locus</th>
<th>duration</th>
<th>compile</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>112780</td>
<td>23028</td>
<td>35878</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>100</td>
<td>112780</td>
<td>23028</td>
<td>35878</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>100</td>
<td>112780</td>
<td>23028</td>
<td>35878</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>100</td>
<td>112780</td>
<td>23028</td>
<td>35878</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>100</td>
<td>112780</td>
<td>23028</td>
<td>35878</td>
<td>4</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>100</td>
<td>112780</td>
<td>23028</td>
<td>35878</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>100</td>
<td>112780</td>
<td>23028</td>
<td>35878</td>
<td>6</td>
<td>1</td>
<td>1380</td>
<td>1</td>
</tr>
</tbody>
</table>
SVV_DISKUSAGE

Amazon Redshift creates the SVV_DISKUSAGE system view by joining the STV_TBL_PERM and STV_BLOCKLIST tables. The SVV_DISKUSAGE view contains information about data allocation for the tables in a database.

Use aggregate queries with SVV_DISKUSAGE, as the following examples show, to determine the number of disk blocks allocated per database, table, slice, or column. Each data block uses 1 MB. You can also use STV_PARTITIONSP (p. 657) to view summary information about disk utilization.

SVV_DISKUSAGE is superuser visible.

Table rows

<table>
<thead>
<tr>
<th>Row name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>db_id</td>
<td>integer</td>
<td>Database ID.</td>
</tr>
<tr>
<td>name</td>
<td>character</td>
<td>Table name.</td>
</tr>
<tr>
<td>slice</td>
<td>integer</td>
<td>Data slice allocated to the table.</td>
</tr>
<tr>
<td>col</td>
<td>integer</td>
<td>Zero-based index for the column. Every table you create has three hidden columns appended to it: INSERT_XID, DELETE_XID, and ROW_ID (OID). A table with 3 user-defined columns contains 6 actual columns, and the user-defined columns are internally numbered as 0, 1, and 2. The INSERT_XID, DELETE_XID, and ROW_ID columns are numbered 3, 4, and 5, respectively, in this example.</td>
</tr>
<tr>
<td>tbl</td>
<td>integer</td>
<td>Table ID.</td>
</tr>
<tr>
<td>blocknum</td>
<td>integer</td>
<td>ID for the data block.</td>
</tr>
<tr>
<td>num_values</td>
<td>integer</td>
<td>Number of values contained on the block.</td>
</tr>
<tr>
<td>minvalue</td>
<td>bigint</td>
<td>Minimum value contained on the block.</td>
</tr>
<tr>
<td>maxvalue</td>
<td>bigint</td>
<td>Maximum value contained on the block.</td>
</tr>
<tr>
<td>sb_pos</td>
<td>integer</td>
<td>Internal identifier for the position of the super block on disk.</td>
</tr>
<tr>
<td>pinned</td>
<td>integer</td>
<td>Whether or not the block is pinned into memory as part of pre-load. 0 = false; 1 = true. Default is false.</td>
</tr>
</tbody>
</table>
SVV_DISKUSAGE contains one row per allocated disk block, so a query that selects all the rows potentially returns a very large number of rows. We recommend using only aggregate queries with SVV_DISKUSAGE.

Return the highest number of blocks ever allocated to column 6 in the USERS table (the EMAIL column):

```sql
select db_id, trim(name) as tablename, max(blocknum) from svv_diskusage where name='users' and col=6 group by db_id, name;
```

<table>
<thead>
<tr>
<th>db_id</th>
<th>tablename</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>175857</td>
<td>users</td>
<td>2</td>
</tr>
</tbody>
</table>

(1 row)

The following query returns similar results for all of the columns in a large 10-column table called SALESNEW. (The last three rows, for columns 10 through 12, are for the hidden metadata columns.)

```sql
select db_id, trim(name) as tablename, col, tbl, max(blocknum) from svv_diskusage where name='salesnew' group by db_id, name, col, tbl order by db_id, name, col, tbl;
```

<table>
<thead>
<tr>
<th>db_id</th>
<th>tablename</th>
<th>col</th>
<th>tbl</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>175857</td>
<td>salesnew</td>
<td>0</td>
<td>187605</td>
<td>154</td>
</tr>
<tr>
<td>175857</td>
<td>salesnew</td>
<td>1</td>
<td>187605</td>
<td>154</td>
</tr>
<tr>
<td>175857</td>
<td>salesnew</td>
<td>2</td>
<td>187605</td>
<td>154</td>
</tr>
<tr>
<td>175857</td>
<td>salesnew</td>
<td>3</td>
<td>187605</td>
<td>154</td>
</tr>
</tbody>
</table>
SVL_QERROR

The SVL_QERROR view is deprecated.

SVL_QLOG

The SVL_QLOG view contains a log of all queries run against the database.

Amazon Redshift creates the SVL_QLOG view as a readable subset of information from the STL_QUERY (p. 620) table. Use this table to find the query ID for a recently run query or to see how long it took a query to complete.

SVL_QLOG is user visible.

Table columns

<table>
<thead>
<tr>
<th>Column name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>userid</td>
<td>integer</td>
<td>ID of user who generated entry.</td>
</tr>
<tr>
<td>query</td>
<td>integer</td>
<td>Query ID. Can be used to join various other system tables and views.</td>
</tr>
<tr>
<td>xid</td>
<td>bigint</td>
<td>Transaction ID.</td>
</tr>
<tr>
<td>pid</td>
<td>integer</td>
<td>Process ID associated with the query.</td>
</tr>
<tr>
<td>starttime</td>
<td>timestamp without time zone</td>
<td>Exact time when the statement started executing, with 6 digits of precision for fractional seconds. For example: 2009-06-12 11:29:19.131358</td>
</tr>
<tr>
<td>endtime</td>
<td>timestamp without time zone</td>
<td>Exact time when the statement finished executing, with 6 digits of precision for fractional seconds. For example: 2009-06-12 11:29:19.193640</td>
</tr>
<tr>
<td>elapsed</td>
<td>bigint</td>
<td>Length of time that it took the query to execute (in microseconds).</td>
</tr>
<tr>
<td>aborted</td>
<td>integer</td>
<td>If a query was aborted by the system or cancelled by the user, this column contains 1. If the query ran to completion, this column contains 0. Queries that are aborted for workload management purposes (and subsequently restarted) also have a value of 1 in this column.</td>
</tr>
</tbody>
</table>
### Sample queries

The following example returns the query ID, execution time, and truncated query text for the five most recent database queries executed by the user with `userid = 100`.

```sql
select query, pid, elapsed, substring from svl_qlog
where userid = 100
order by starttime desc
limit 5;
```

<table>
<thead>
<tr>
<th>query</th>
<th>pid</th>
<th>elapsed</th>
<th>substring</th>
</tr>
</thead>
<tbody>
<tr>
<td>187752</td>
<td>18921</td>
<td>18465685</td>
<td>select query, elapsed, substring from svl_...</td>
</tr>
<tr>
<td>204168</td>
<td>5117</td>
<td>59603</td>
<td>insert into testtable values (100);</td>
</tr>
<tr>
<td>187561</td>
<td>17046</td>
<td>1003052</td>
<td>select * from pg_table_def where tablename...</td>
</tr>
<tr>
<td>187549</td>
<td>17046</td>
<td>1108584</td>
<td>select * from STV_WLM_SERVICE_CLASS_CONFIG</td>
</tr>
<tr>
<td>187468</td>
<td>17046</td>
<td>5670661</td>
<td>select * from pg_table_def where schemaname...</td>
</tr>
</tbody>
</table>

(5 rows)

The following example returns the SQL script name (LABEL column) and elapsed time for a query that was cancelled (`aborted=1`):

```sql
select query, elapsed, label
from svl_qlog where aborted=1;
```

<table>
<thead>
<tr>
<th>query</th>
<th>elapsed</th>
<th>label</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>6935292</td>
<td>alltickittablesjoin.sql</td>
</tr>
</tbody>
</table>

(1 row)

### SVV_QUERY_INFLIGHT

Use the `SVV_QUERY_INFLIGHT` view to determine what queries are currently running on the database. This view joins `STV_INFLIGHT` (p. 654) to `STL_QUERYTEXT` (p. 622).

`SVV_QUERY_INFLIGHT` is user visible.

### Table columns

<table>
<thead>
<tr>
<th>Column name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>userid</td>
<td>integer</td>
<td>ID of user who generated entry.</td>
</tr>
</tbody>
</table>
### SVL_QUERY_REPORT

Amazon Redshift creates the SVL_QUERY_REPORT view from a UNION of a number of Amazon Redshift STL system tables to provide information about executed query steps.

This view breaks down the information about executed queries by slice and by step, which can help with troubleshooting node and slice issues in the Amazon Redshift cluster.

SVL_QUERY_REPORT is user visible.

### Sample queries

The sample output below shows two queries currently running, the SVV_QUERY_INFLIGHT query itself and query 428, which is split into three rows in the table. (The starttime and statement columns are truncated in this sample output.)

```sql
select slice, query, pid, starttime, suspended, trim(text) as statement, sequence
from svv_query_inflight
order by query, sequence;
```

<table>
<thead>
<tr>
<th>slice</th>
<th>query</th>
<th>pid</th>
<th>starttime</th>
<th>suspended</th>
<th>statement</th>
<th>sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>1012</td>
<td>428</td>
<td>1658</td>
<td>2012-04-10 13:53:...</td>
<td>0</td>
<td>select ...</td>
<td>0</td>
</tr>
<tr>
<td>1012</td>
<td>428</td>
<td>1658</td>
<td>2012-04-10 13:53:...</td>
<td>0</td>
<td>enueid ...</td>
<td>1</td>
</tr>
<tr>
<td>1012</td>
<td>428</td>
<td>1658</td>
<td>2012-04-10 13:53:...</td>
<td>0</td>
<td>atname,...</td>
<td>2</td>
</tr>
<tr>
<td>1012</td>
<td>429</td>
<td>1608</td>
<td>2012-04-10 13:53:...</td>
<td>0</td>
<td>select ...</td>
<td>0</td>
</tr>
</tbody>
</table>

(4 rows)

### Table rows

<table>
<thead>
<tr>
<th>Row name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>userid</td>
<td>integer</td>
<td>ID of user who generated entry.</td>
</tr>
</tbody>
</table>
### Table Description

<table>
<thead>
<tr>
<th>Row name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>query</td>
<td>integer</td>
<td>Query ID. Can be used to join various other system tables and views.</td>
</tr>
<tr>
<td>slice</td>
<td>integer</td>
<td>Data slice where the step executed.</td>
</tr>
<tr>
<td>segment</td>
<td>integer</td>
<td>Segment number. A query consists of multiple segments, and each segment consists of one or more steps. Query segments can run in parallel. Each segment runs in a single process.</td>
</tr>
<tr>
<td>step</td>
<td>integer</td>
<td>Query step that executed.</td>
</tr>
<tr>
<td>start_time</td>
<td>timestamp</td>
<td>Exact time when the step started executing, with 6 digits of precision for fractional seconds. For example: 2012-12-12 11:29:19.131358</td>
</tr>
<tr>
<td>end_time</td>
<td>timestamp</td>
<td>Exact time when the step finished executing, with 6 digits of precision for fractional seconds. For example: 2012-12-12 11:29:19.131467</td>
</tr>
<tr>
<td>elapsed_time</td>
<td>bigint</td>
<td>Time (in microseconds) that it took the step to execute.</td>
</tr>
<tr>
<td>rows</td>
<td>bigint</td>
<td>Number of rows produced by the step (per slice). This number represents the number of rows for the slice that result from the execution of the step, not the number of rows received or processed by the step. In other words, this is the number of rows that survive the step and are passed on to the next step.</td>
</tr>
<tr>
<td>bytes</td>
<td>bigint</td>
<td>Number of bytes produced by the step (per slice).</td>
</tr>
<tr>
<td>label</td>
<td>char(256)</td>
<td>Step label, which consists of a query step name and, when applicable, table ID and table name (for example, scan tbl=100448 name=user). Three-digit table IDs usually refer to scans of transient tables. When you see tbl=0, it usually refers to a scan of a constant value.</td>
</tr>
<tr>
<td>is_diskbased</td>
<td>character(1)</td>
<td>Whether this step of the query was executed as a disk-based operation: true (t) or false (f). Only certain steps, such as hash, sort, and aggregate steps, can go to disk. Many types of steps are always executed in memory.</td>
</tr>
<tr>
<td>workmem</td>
<td>bigint</td>
<td>Amount of working memory (in bytes) assigned to the query step. This value is the query_working_mem threshold allocated for use during execution, not the amount of memory that was actually used.</td>
</tr>
<tr>
<td>is_rrscan</td>
<td>character(1)</td>
<td>If true (t), indicates that range-restricted scan was used on the step. Default is false (f).</td>
</tr>
<tr>
<td>is_delayed_scan</td>
<td>character(1)</td>
<td>If true (t), indicates that delayed scan was used on the step. Default is false (f).</td>
</tr>
<tr>
<td>rows_pre_filter</td>
<td>bigint</td>
<td>For scans of permanent tables, the total number of rows emitted before filtering rows marked for deletion (ghost rows) and before applying user-defined query filters.</td>
</tr>
</tbody>
</table>

### Sample queries

The following query demonstrates the data skew of the returned rows for the query with query ID 279. Use this query to determine if database data is evenly distributed over the slices in the data warehouse cluster:
```sql
select query, segment, step, max(rows), min(rows),
case when sum(rows) > 0
then ((cast(max(rows) - min(rows) as float) * count(rows)) / sum(rows))
else 0 end
from svl_query_report
where query = 279
group by query, segment, step
order by segment, step;
```

This query should return data similar to the following sample output:

```plaintext
query  | segment | step |   max    |   min    |         case
-------|---------|------|----------|----------|----------------------
279    |       0 |    0 | 19721687 | 19721687 |                    0
279    |       0 |    1 | 19721687 | 19721687 |                    0
279    |       1 |    0 | 986085   | 986084   | 1.01411202804304e-06
279    |       1 |    1 | 986085   | 986084   | 1.01411202804304e-06
279    |       1 |    4 | 986085   | 986084   | 1.01411202804304e-06
279    |       2 |    0 | 1775517  | 788460   |     1.00098637606408
279    |       2 |    2 | 1775517  | 788460   |     1.00098637606408
279    |       3 |    0 | 1775517  | 788460   |     1.00098637606408
279    |       3 |    2 | 1775517  | 788460   |     1.00098637606408
279    |       3 |    3 | 1775517  | 788460   |     1.00098637606408
279    |       4 |    0 | 1775517  | 788460   |     1.00098637606408
279    |       4 |    1 | 1775517  | 788460   |     1.00098637606408
279    |       4 |    2 |        1 |        1 |                    0
279    |       5 |    0 |        1 |        1 |                    0
279    |       5 |    1 |        1 |        1 |                    0
279    |       6 |    0 |        20|        20|                    0
279    |       6 |    1 |        1 |        1 |                    0
279    |       7 |    0 |        1 |        1 |                    0
279    |       7 |    1 |        0 |        0 |                    0
(19 rows)
```

**SVV_QUERY_STATE**

Use SVV_QUERY_STATE to view information about the execution of currently running queries.

The SVV_QUERY_STATE view contains a data subset of the STV_EXEC_STATE table.

SVV_QUERY_STATE is user visible.

**Table columns**

<table>
<thead>
<tr>
<th>Column name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>userid</td>
<td>integer</td>
<td>ID of user who generated entry.</td>
</tr>
<tr>
<td>query</td>
<td>integer</td>
<td>Query ID. Can be used to join various other system tables and views.</td>
</tr>
<tr>
<td>seg</td>
<td>integer</td>
<td>Number of the query segment that is executing. A query consists of multiple segments, and each segment consists of one or more steps. Query segments can run in parallel. Each segment runs in a single process.</td>
</tr>
</tbody>
</table>
### Description

<table>
<thead>
<tr>
<th>Column name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>step</td>
<td>integer</td>
<td>Number of the query step that is executing. A step is the smallest unit of query execution. Each step represents a discrete unit of work, such as scanning a table, returning results, or sorting data.</td>
</tr>
<tr>
<td>maxtime</td>
<td>interval</td>
<td>Maximum amount of time (in microseconds) for this step to execute.</td>
</tr>
<tr>
<td>avgtime</td>
<td>interval</td>
<td>Average time (in microseconds) for this step to execute.</td>
</tr>
<tr>
<td>rows</td>
<td>bigint</td>
<td>Number of rows produced by the step that is executing.</td>
</tr>
<tr>
<td>bytes</td>
<td>bigint</td>
<td>Number of bytes produced by the step that is executing.</td>
</tr>
<tr>
<td>cpu</td>
<td>bigint</td>
<td>The percentage of CPU used by the query processes in the current stream. (Time is measured in microseconds.)</td>
</tr>
<tr>
<td>memory</td>
<td>bigint</td>
<td>The current amount of shared memory used by the current stream (in bytes).</td>
</tr>
<tr>
<td>rate_row</td>
<td>double precision</td>
<td>Rows-per-second rate since the query started, computed by summing the rows and dividing by the number of seconds from when the query started to the current time.</td>
</tr>
<tr>
<td>rate_byte</td>
<td>double precision</td>
<td>Bytes-per-second rate since the query started, computed by summing the bytes and dividing by the number of seconds from when the query started to the current time.</td>
</tr>
<tr>
<td>label</td>
<td>character(25)</td>
<td>Query label: a name for the step, such as scan or sort.</td>
</tr>
<tr>
<td>is_diskbased</td>
<td>character(1)</td>
<td>Whether this step of the query is executing as a disk-based operation: true (t) or false (f). Only certain steps, such as hash, sort, and aggregate steps, can go to disk. Many types of steps are always executed in memory.</td>
</tr>
<tr>
<td>workmem</td>
<td>bigint</td>
<td>Amount of working memory (in bytes) assigned to the query step.</td>
</tr>
<tr>
<td>num_parts</td>
<td>integer</td>
<td>Number of partitions a hash table is divided into during a hash step. The hash table is partitioned when it is estimated that the entire hash table might not fit into memory. A positive number in this column does not imply that the hash step executed as a disk-based operation. Check the value in the IS_DISKBASED column to see if the hash step was disk-based.</td>
</tr>
<tr>
<td>is_rrscan</td>
<td>character(1)</td>
<td>If true (t), indicates that range-restricted scan was used on the step. Default is false (f).</td>
</tr>
<tr>
<td>is_delayed_scan</td>
<td>character(1)</td>
<td>If true (t), indicates that delayed scan was used on the step. Default is false (f).</td>
</tr>
</tbody>
</table>

### Sample queries

#### Determining the processing time of a query by step

The following query shows how long each step of the query with query ID 279 took to execute and how many data rows Amazon Redshift processed:
This query retrieves the processing information about query 279, as shown in the following sample output:

<table>
<thead>
<tr>
<th>query</th>
<th>seg</th>
<th>step</th>
<th>maxtime</th>
<th>avgtime</th>
<th>rows</th>
<th>label</th>
</tr>
</thead>
<tbody>
<tr>
<td>279</td>
<td>3</td>
<td>0</td>
<td>1658054</td>
<td>1645711</td>
<td>1405360</td>
<td>scan</td>
</tr>
<tr>
<td>279</td>
<td>3</td>
<td>1</td>
<td>1658072</td>
<td>1645809</td>
<td>0</td>
<td>project</td>
</tr>
<tr>
<td>279</td>
<td>3</td>
<td>2</td>
<td>1658074</td>
<td>1645812</td>
<td>1405434</td>
<td>insert</td>
</tr>
<tr>
<td>279</td>
<td>3</td>
<td>3</td>
<td>1658080</td>
<td>1645816</td>
<td>1405437</td>
<td>distribute</td>
</tr>
<tr>
<td>279</td>
<td>4</td>
<td>0</td>
<td>1677443</td>
<td>1666189</td>
<td>1268431</td>
<td>scan</td>
</tr>
<tr>
<td>279</td>
<td>4</td>
<td>1</td>
<td>1677446</td>
<td>1666192</td>
<td>1268434</td>
<td>insert</td>
</tr>
<tr>
<td>279</td>
<td>4</td>
<td>2</td>
<td>1677451</td>
<td>1666195</td>
<td>0</td>
<td>aggr</td>
</tr>
</tbody>
</table>

(7 rows)

Determining if any active queries are currently running on disk

The following query shows if any active queries are currently running on disk:

```sql
select query, label, is_diskbased from svv_query_state
where is_diskbased = 't';
```

This sample output shows any active queries currently running on disk:

<table>
<thead>
<tr>
<th>query</th>
<th>label</th>
<th>is_diskbased</th>
</tr>
</thead>
<tbody>
<tr>
<td>1025</td>
<td>hash tbl=142</td>
<td></td>
</tr>
</tbody>
</table>

(1 row)

SVL_QUERY_SUMMARY

Use the SVL_QUERY_SUMMARY view to find general information about the execution of a query.

The SVL_QUERY_SUMMARY view contains a subset of data from the SVL_QUERY_REPORT view. Note that the information in SVL_QUERY_SUMMARY is aggregated from all nodes.

**Note**

The SVL_QUERY_SUMMARY view only contains information about queries executed by Amazon Redshift, not other utility and DDL commands. For a complete listing and information on all statements executed by Amazon Redshift, including DDL and utility commands, you can query the SVL_STATEMENTTEXT view.

SVL_QUERY_SUMMARY is user visible.

**Table columns**

<table>
<thead>
<tr>
<th>Column name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>userid</td>
<td>integer</td>
<td>ID of user who generated entry.</td>
</tr>
<tr>
<td>Column name</td>
<td>Data type</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>-----------</td>
<td>-------------</td>
</tr>
<tr>
<td>query</td>
<td>integer</td>
<td>Query ID. Can be used to join various other system tables and views.</td>
</tr>
<tr>
<td>stm</td>
<td>integer</td>
<td>Stream: A set of concurrent segments in a query. A query has one or more streams.</td>
</tr>
<tr>
<td>seg</td>
<td>integer</td>
<td>Segment number. A query consists of multiple segments, and each segment consists of one or more steps. Query segments can run in parallel. Each segment runs in a single process.</td>
</tr>
<tr>
<td>step</td>
<td>integer</td>
<td>Query step that executed.</td>
</tr>
<tr>
<td>maxtime</td>
<td>bigint</td>
<td>Maximum amount of time for the step to execute (in microseconds).</td>
</tr>
<tr>
<td>avgtime</td>
<td>bigint</td>
<td>Average time for the step to execute (in microseconds).</td>
</tr>
<tr>
<td>rows</td>
<td>bigint</td>
<td>Number of data rows involved in the query step.</td>
</tr>
<tr>
<td>bytes</td>
<td>bigint</td>
<td>Number of data bytes involved in the query step.</td>
</tr>
<tr>
<td>rate_row</td>
<td>double precision</td>
<td>Query execution rate per row.</td>
</tr>
<tr>
<td>rate_byte</td>
<td>double precision</td>
<td>Query execution rate per byte.</td>
</tr>
<tr>
<td>label</td>
<td>text</td>
<td>Step label, which consists of a query step name and, when applicable, table ID and table name (for example, scan tbl=100448 name =user). Three-digit table IDs usually refer to scans of transient tables. When you see tbl=0, it usually refers to a scan of a constant value.</td>
</tr>
<tr>
<td>is_diskbased</td>
<td>character(1)</td>
<td>Whether this step of the query was executed as a disk-based operation on any node in the cluster: true (t) or false (f). Only certain steps, such as hash, sort, and aggregate steps, can go to disk. Many types of steps are always executed in memory.</td>
</tr>
<tr>
<td>workmem</td>
<td>bigint</td>
<td>Amount of working memory (in bytes) assigned to the query step.</td>
</tr>
<tr>
<td>is_rrscan</td>
<td>character(1)</td>
<td>If true (t), indicates that range-restricted scan was used on the step. Default is false (f).</td>
</tr>
<tr>
<td>is_delayed_scan</td>
<td>character(1)</td>
<td>If true (t), indicates that delayed scan was used on the step. Default is false (f).</td>
</tr>
<tr>
<td>rows_pre_filter</td>
<td>bigint</td>
<td>For scans of permanent tables, the total number of rows emitted before filtering rows marked for deletion (ghost rows).</td>
</tr>
</tbody>
</table>

**Sample queries**

**Viewing processing information for a query step**

The following query shows basic processing information for each step of query 87:

```sql
select query, stm, seg, step, rows, bytes
from svl_query_summary
where query = 87
order by query, seg, step;
```
This query retrieves the processing information about query 87, as shown in the following sample output:

<table>
<thead>
<tr>
<th>query</th>
<th>stm</th>
<th>seg</th>
<th>step</th>
<th>rows</th>
<th>bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>87</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>90</td>
<td>1890</td>
</tr>
<tr>
<td>87</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>90</td>
<td>360</td>
</tr>
<tr>
<td>87</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>90</td>
<td>360</td>
</tr>
<tr>
<td>87</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>90</td>
<td>1440</td>
</tr>
<tr>
<td>87</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>210494</td>
<td>4209880</td>
</tr>
<tr>
<td>87</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>89500</td>
<td>0</td>
</tr>
<tr>
<td>87</td>
<td>1</td>
<td>2</td>
<td>6</td>
<td>4</td>
<td>96</td>
</tr>
<tr>
<td>87</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>4</td>
<td>96</td>
</tr>
<tr>
<td>87</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>4</td>
<td>96</td>
</tr>
<tr>
<td>87</td>
<td>2</td>
<td>4</td>
<td>0</td>
<td>4</td>
<td>96</td>
</tr>
<tr>
<td>87</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>24</td>
</tr>
<tr>
<td>87</td>
<td>3</td>
<td>5</td>
<td>0</td>
<td>1</td>
<td>24</td>
</tr>
<tr>
<td>87</td>
<td>3</td>
<td>5</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

(13 rows)

Determining if any query steps spilled to disk

The following query shows whether or not any of the steps for the query with query ID 1025 (see the SVL_QLOG (p. 675) view to learn how to obtain the query ID for a query) spilled to disk or if the query ran entirely in-memory:

```sql
select query, step, rows, workmem, label, is_diskbased
from svl_query_summary
where query = 1025
order by workmem desc;
```

This query returns the following sample output:

<table>
<thead>
<tr>
<th>query</th>
<th>step</th>
<th>rows</th>
<th>workmem</th>
<th>label</th>
<th>is_diskbased</th>
</tr>
</thead>
<tbody>
<tr>
<td>1025</td>
<td>0</td>
<td>1600000</td>
<td>141557760</td>
<td>scan tbl=9</td>
<td>f</td>
</tr>
<tr>
<td>1025</td>
<td>2</td>
<td>1600000</td>
<td>135266304</td>
<td>hash tbl=142</td>
<td>t</td>
</tr>
<tr>
<td>1025</td>
<td>0</td>
<td>1600000</td>
<td>128974848</td>
<td>scan tbl=116536</td>
<td>f</td>
</tr>
<tr>
<td>1025</td>
<td>2</td>
<td>1600000</td>
<td>122683392</td>
<td>dist</td>
<td>f</td>
</tr>
</tbody>
</table>

(4 rows)

By scanning the values for IS.DISKBASED, you can see if any query steps went to disk. For query 1025, the hash step ran on disk. Steps that could possibly run on disk would be hash, aggr, and sort steps. Another option to view disk-based query steps is to add a `WHERE is_diskbased = "t"` clause to the SQL statement in the above example.

**SVL_STATEMENTTEXT**

Use the SVL_STATEMENTTEXT view to get a complete record of all of the SQL commands that have been run on the system.

The SVL_STATEMENTTEXT view contains the union of all of the rows in the STL_DDLTEXT (p. 591), STL_QUERYTEXT (p. 622), and STL_UTILITYTEXT (p. 640) tables. This view also includes a join to the STL_QUERY table.

SVL_STATEMENTTEXT is user visible.
## Table columns

<table>
<thead>
<tr>
<th>Column name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>userid</td>
<td>integer</td>
<td>ID of user who generated entry.</td>
</tr>
<tr>
<td>xid</td>
<td>bigint</td>
<td>Transaction ID associated with the statement.</td>
</tr>
<tr>
<td>pid</td>
<td>integer</td>
<td>Process ID for the statement.</td>
</tr>
<tr>
<td>label</td>
<td>character(30)</td>
<td>Either the name of the file used to run the query or a label defined with a SET QUERY_GROUP command. If the query is not file-based or the QUERY_GROUP parameter is not set, this field is blank.</td>
</tr>
<tr>
<td>starttime</td>
<td>timestamp without time zone</td>
<td>Exact time when the statement started executing, with 6 digits of precision for fractional seconds. For example: 2009-06-12 11:29:19.131358</td>
</tr>
<tr>
<td>endtime</td>
<td>timestamp without time zone</td>
<td>Exact time when the statement finished executing, with 6 digits of precision for fractional seconds. For example: 2009-06-12 11:29:19.193640</td>
</tr>
<tr>
<td>sequence</td>
<td>integer</td>
<td>When a single statement contains more than 200 characters, additional rows are logged for that statement. Sequence 0 is the first row, 1 is the second, and so on.</td>
</tr>
<tr>
<td>type</td>
<td>varchar(10)</td>
<td>Type of SQL statement: QUERY, DDL, or UTILITY.</td>
</tr>
<tr>
<td>text</td>
<td>character(200)</td>
<td>SQL text, in 200-character increments.</td>
</tr>
</tbody>
</table>

### Sample query

The following query returns DDL statements that were run on June 16th, 2009:

```sql
select starttime, type, rtrim(text) from svl_statementtext
where starttime like '2009-06-16%' and type='DDL' order by starttime asc;
```

<table>
<thead>
<tr>
<th>starttime</th>
<th>type</th>
<th>rtrim</th>
</tr>
</thead>
<tbody>
<tr>
<td>2009-06-16 10:36:50.625097</td>
<td>DDL</td>
<td>create table ddltest(c1 int);</td>
</tr>
<tr>
<td>2009-06-16 15:02:16.006341</td>
<td>DDL</td>
<td>drop view alltickitjoin;</td>
</tr>
<tr>
<td>2009-06-16 15:02:23.65285</td>
<td>DDL</td>
<td>drop table sales;</td>
</tr>
<tr>
<td>2009-06-16 15:02:24.548928</td>
<td>DDL</td>
<td>drop table listing;</td>
</tr>
<tr>
<td>2009-06-16 15:02:25.536655</td>
<td>DDL</td>
<td>drop table event;</td>
</tr>
</tbody>
</table>

### SVV_VACUUM_PROGRESS

This view returns an estimate of how much time it will take to complete a vacuum operation that is currently in progress.

SVV_VACUUM_PROGRESS is superuser visible.
Table columns

<table>
<thead>
<tr>
<th>Row name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>table_name</td>
<td>text</td>
<td>Name of the table currently being vacuumed, or the table that was last vacuumed if no operation is in progress.</td>
</tr>
<tr>
<td>status</td>
<td>text</td>
<td>Description of the current activity being done as part of the vacuum operation (initialize, sort, or merge, for example).</td>
</tr>
<tr>
<td>time_remaining_estimate</td>
<td>text</td>
<td>Estimated time left for the current vacuum operation to complete, in minutes and seconds: 5m 10s, for example. An estimated time is not returned until the vacuum completes its first sort operation. If no vacuum is in progress, the last vacuum that was executed is displayed with Completed in the STATUS column and an empty TIME_REMAINING_ESTIMATE column. The estimate typically becomes more accurate as the vacuum progresses.</td>
</tr>
</tbody>
</table>

Sample queries

The following queries, run a few minutes apart, show that a large table named SALESNEW is being vacuumed.

```
select * from svv_vacuum_progress;
```

<table>
<thead>
<tr>
<th>table_name</th>
<th>status</th>
<th>time_remaining_estimate</th>
</tr>
</thead>
<tbody>
<tr>
<td>salesnew</td>
<td>Vacuum: initialize salesnew</td>
<td></td>
</tr>
<tr>
<td>(1 row)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```
select * from svv_vacuum_progress;
```

<table>
<thead>
<tr>
<th>table_name</th>
<th>status</th>
<th>time_remaining_estimate</th>
</tr>
</thead>
<tbody>
<tr>
<td>salesnew</td>
<td>Vacuum salesnew sort</td>
<td>33m 21s</td>
</tr>
<tr>
<td>(1 row)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The following query shows that no vacuum operation is currently in progress. The last table to be vacuumed was the SALES table.

```
select * from svv_vacuum_progress;
```

<table>
<thead>
<tr>
<th>table_name</th>
<th>status</th>
<th>time_remaining_estimate</th>
</tr>
</thead>
<tbody>
<tr>
<td>sales</td>
<td>Complete</td>
<td></td>
</tr>
<tr>
<td>(1 row)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SVV_VACUUM_SUMMARY

The SVV_VACUUM_SUMMARY view joins the STL_VACUUM, STL_QUERY, and STV_TBL_PERM tables to summarize information about vacuum operations logged by the system. The view returns one row per table per vacuum transaction. The view records the elapsed time of the operation, the number of sort partitions created, the number of merge increments required, and deltas in row and block counts before and after the operation was performed.
SVV_VACUUM_SUMMARY is superuser visible.

## Table columns

<table>
<thead>
<tr>
<th>Row name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>table_name</td>
<td>text</td>
<td>Name of the vacuumed table.</td>
</tr>
<tr>
<td>xid</td>
<td>bigint</td>
<td>Transaction ID of the VACUUM operation.</td>
</tr>
<tr>
<td>sort_partitions</td>
<td>bigint</td>
<td>Number of sorted partitions created during the sort phase of the vacuum operation.</td>
</tr>
<tr>
<td>merge_increments</td>
<td>bigint</td>
<td>Number of merge increments required to complete the merge phase of the vacuum operation.</td>
</tr>
<tr>
<td>elapsed_time</td>
<td>bigint</td>
<td>Elapsed run time of the vacuum operation (in microseconds).</td>
</tr>
<tr>
<td>row_delta</td>
<td>bigint</td>
<td>Difference in the total number of table rows before and after the vacuum.</td>
</tr>
<tr>
<td>sortedrow_delta</td>
<td>bigint</td>
<td>Difference in the number of sorted table rows before and after the vacuum.</td>
</tr>
<tr>
<td>block_delta</td>
<td>integer</td>
<td>Difference in block count for the table before and after the vacuum.</td>
</tr>
<tr>
<td>max_merge_partitions</td>
<td>integer</td>
<td>This column is used for performance analysis and represents the maximum number of partitions that vacuum can process for the table per merge phase iteration. (Vacuum sorts the unsorted region into one or more sorted partitions. Depending on the number of columns in the table and the current Amazon Redshift configuration, the merge phase can process a maximum number of partitions in a single merge iteration. The merge phase will still work if the number of sorted partitions exceeds the maximum number of merge partitions, but more merge iterations will be required.)</td>
</tr>
</tbody>
</table>

## Sample query

The following query returns statistics for vacuum operations on three different tables. The SALES table was vacuumed twice.

```sql
select table_name, xid, sort_partitions as parts, merge_increments as merges, elapsed_time, row_delta, sortedrow_delta as sorted_delta, block_delta
from svv_vacuum_summary
order by xid;
```

<table>
<thead>
<tr>
<th>table_name</th>
<th>xid</th>
<th>parts</th>
<th>merges</th>
<th>elapsed_time</th>
<th>row_delta</th>
<th>sorted_delta</th>
<th>block_delta</th>
</tr>
</thead>
<tbody>
<tr>
<td>users</td>
<td>2985</td>
<td>1</td>
<td>1</td>
<td>61919653</td>
<td>0</td>
<td>49990</td>
<td>20</td>
</tr>
<tr>
<td>category</td>
<td>3982</td>
<td>1</td>
<td>1</td>
<td>24136484</td>
<td>0</td>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>sales</td>
<td>3992</td>
<td>2</td>
<td>1</td>
<td>71736163</td>
<td>0</td>
<td>1207192</td>
<td>32</td>
</tr>
<tr>
<td>sales</td>
<td>4000</td>
<td>1</td>
<td>1</td>
<td>15363010</td>
<td>-851648</td>
<td>-851648</td>
<td>-140</td>
</tr>
</tbody>
</table>

(4 rows)
SVL_VACUUM_PERCENTAGE

The SVL_VACUUM_PERCENTAGE view reports the percentage of data blocks allocated to a table after performing a vacuum. This percentage number shows how much disk space was reclaimed. See the VACUUM (p. 411) command for more information about the vacuum utility.

SVL_VACUUM_PERCENTAGE is superuser visible.

Table rows

<table>
<thead>
<tr>
<th>Row name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xid</td>
<td>bigint</td>
<td>Transaction ID for the vacuum statement.</td>
</tr>
<tr>
<td>table_id</td>
<td>integer</td>
<td>Table ID for the vacuumed table.</td>
</tr>
<tr>
<td>percentage</td>
<td>bigint</td>
<td>Percentage of data blocks after a vacuum (relative to the number of blocks in the table before the vacuum was run).</td>
</tr>
</tbody>
</table>

Sample query

The following query displays the percentage for a specific operation on table 100238:

```
select * from svl_vacuum_percentage
where table_id=100238 and xid=2200;
```

<table>
<thead>
<tr>
<th>xid</th>
<th>table_id</th>
<th>percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>1337</td>
<td>100238</td>
<td>60</td>
</tr>
</tbody>
</table>

(1 row)

After this vacuum operation, the table contained 60% of the original blocks.

System catalog tables

Topics

- PG_TABLE_DEF (p. 687)
- Querying the catalog tables (p. 689)

The system catalogs store schema metadata, such as information about tables and columns. System catalog tables have a PG prefix.

The standard PostgreSQL catalog tables are accessible to Amazon Redshift users. For more information about PostgreSQL system catalogs, see PostgreSQL System Tables

PG_TABLE_DEF

Stores information about table columns.
PG_TABLE_DEF only returns information about tables that are visible to the user. If PG_TABLE_DEF does not return the expected results, verify that the search_path (p. 696) parameter is set correctly to include the relevant schemas.

### Table columns

<table>
<thead>
<tr>
<th>Column name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>schemaname</td>
<td>name</td>
<td>Schema name.</td>
</tr>
<tr>
<td>tablename</td>
<td>name</td>
<td>Table name.</td>
</tr>
<tr>
<td>column</td>
<td>name</td>
<td>Column name.</td>
</tr>
<tr>
<td>type</td>
<td>text</td>
<td>Datatype of column.</td>
</tr>
<tr>
<td>encoding</td>
<td>character(32)</td>
<td>Encoding of column.</td>
</tr>
<tr>
<td>distkey</td>
<td>boolean</td>
<td>True if this column is the distribution key for the table.</td>
</tr>
<tr>
<td>sortkey</td>
<td>integer</td>
<td>If greater than 0, the column is part of the sort key. 1 = primary sort key, 2 = secondary sort key, and so on.</td>
</tr>
<tr>
<td>notnull</td>
<td>boolean</td>
<td>True if the column has a NOT NULL constraint.</td>
</tr>
</tbody>
</table>

### Example

This example returns the information for table T2.

```
select * from pg_table_def where tablename = 't2';
```

<table>
<thead>
<tr>
<th>schemaname</th>
<th>tablename</th>
<th>column</th>
<th>type</th>
<th>encoding</th>
<th>distkey</th>
<th>sortkey</th>
<th>notnull</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>t2</td>
<td>c1</td>
<td>bigint</td>
<td>none</td>
<td>t</td>
<td>0</td>
<td>f</td>
</tr>
<tr>
<td>public</td>
<td>t2</td>
<td>c2</td>
<td>integer</td>
<td>mostly16</td>
<td>f</td>
<td>0</td>
<td>f</td>
</tr>
<tr>
<td>public</td>
<td>t2</td>
<td>c3</td>
<td>integer</td>
<td>none</td>
<td>f</td>
<td>1</td>
<td>t</td>
</tr>
<tr>
<td>public</td>
<td>t2</td>
<td>c4</td>
<td>integer</td>
<td>none</td>
<td>f</td>
<td>2</td>
<td>f</td>
</tr>
</tbody>
</table>

(4 rows)

PG_TABLE_DEF will only return information for tables in schemas that are included in the search path. See search_path (p. 696).

For example, suppose you create a new schema and a new table, then query PG_TABLE_DEF.

```
create schema demo;
create table demo.demotable (one int);
select * from pg_table_def where tablename = 'demotable';
```

<table>
<thead>
<tr>
<th>schemaname</th>
<th>tablename</th>
<th>column</th>
<th>type</th>
<th>encoding</th>
<th>distkey</th>
<th>sortkey</th>
<th>notnull</th>
</tr>
</thead>
</table>

The query returns no rows for the new table. Examine the setting for search_path.

```
show search_path;
```
search_path
--------------
$user, public
(1 row)

Add the demo schema to the search path and execute the query again.

```
set search_path to '$user', 'public', 'demo';
select * from pg_table_def where tablename = 'demotable';
```

```
schemaname | tablename | column | type    | encoding | distkey | sortkey | notnull
----------+-----------+--------+---------+----------+--------+--------+--------
demo      | demotable | one    | integer | none     | f      |     0  | f
(1 row)
```

**Querying the catalog tables**

**Topics**

- [Examples of catalog queries (p. 690)](#)

In general, you can join catalog tables and views (relations whose names begin with `PG_`) to Amazon Redshift tables and views.

The catalog tables use a number of data types that Amazon Redshift does not support. The following data types are supported when queries join catalog tables to Amazon Redshift tables:

- `bool`
- "char"
- `float4`
- `int2`
- `int4`
- `int8`
- `name`
- `oid`
- `text`
- `varchar`

If you write a join query that explicitly or implicitly references a column that has an unsupported data type, the query returns an error. The SQL functions that are used in some of the catalog tables are also unsupported, except for those used by the `PG_SETTINGS` and `PG_LOCKS` tables.

For example, the `PG_STATS` table cannot be queried in a join with an Amazon Redshift table because of unsupported functions.

The following catalog tables and views provide useful information that can be joined to information in Amazon Redshift tables. Some of these tables allow only partial access because of data type and function restrictions. When you query the partially accessible tables, select or reference their columns carefully.

The following tables are completely accessible and contain no unsupported types or functions:

- `pg_attribute`
The following tables are partially accessible and contain some unsupported types, functions, and truncated text columns. Values in text columns are truncated to `varchar(256)` values.

- `pg_class`
- `pg_constraint`
- `pg_database`
- `pg_group`
- `pg_language`
- `pg_namespace`
- `pg_operator`
- `pg_proc`
- `pg_settings`
- `pg_statistic`
- `pg_tables`
- `pg_type`
- `pg_user`
- `pg_views`

The catalog tables that are not listed here are either inaccessible or unlikely to be useful to Amazon Redshift administrators. However, you can query any catalog table or view openly if your query does not involve a join to an Amazon Redshift table.

You can use the OID columns in the Postgres catalog tables as joining columns. For example, the join condition `pg_database.oid = stv_tbl_perm.db_id` matches the internal database object ID for each `PG_DATABASE` row with the visible `DB_ID` column in the `STV_TBL_PERM` table. The OID columns are internal primary keys that are not visible when you select from the table. The catalog views do not have OID columns.

**Examples of catalog queries**

The following queries show a few of the ways in which you can query the catalog tables to get useful information about an Amazon Redshift database.

**List the number of columns per Amazon Redshift table**

The following query joins some catalog tables to find out how many columns each Amazon Redshift table contains. Amazon Redshift table names are stored in both `PG_TABLES` and `STV_TBL_PERM`; where possible, use `PG_TABLES` to return Amazon Redshift table names.

This query does not involve any Amazon Redshift tables.

```sql
select nspname, relname, max(attnum) as num_cols
from pg_attribute a, pg_namespace n, pg_class c
where n.oid = c.relnamespace and a.attrelid = c.oid
and c.relname not like '%pkey'
```
and n.nspname not like 'pg\%'
and n.nspname not like 'information\%'
group by 1, 2
order by 1, 2;

<table>
<thead>
<tr>
<th>nspsname</th>
<th>relname</th>
<th>num_cols</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>category</td>
<td>4</td>
</tr>
<tr>
<td>public</td>
<td>date</td>
<td>8</td>
</tr>
<tr>
<td>public</td>
<td>event</td>
<td>6</td>
</tr>
<tr>
<td>public</td>
<td>listing</td>
<td>8</td>
</tr>
<tr>
<td>public</td>
<td>sales</td>
<td>10</td>
</tr>
<tr>
<td>public</td>
<td>users</td>
<td>18</td>
</tr>
<tr>
<td>public</td>
<td>venue</td>
<td>5</td>
</tr>
</tbody>
</table>
(7 rows)

List the schemas and tables in a database

The following query joins STV_TBL_PERM to some PG tables to return a list of tables in the TICKIT
database and their schema names (NSPNAME column). The query also returns the total number of rows
in each table. (This query is helpful when multiple schemas in your system have the same table names.)

```
select datname, nspname, relname, sum(rows) as rows
from pg_class, pg_namespace, pg_database, stv_tbl_perm
where pg_namespace.oid = relnamespace
and pg_class.oid = stv_tbl_perm.id
and pg_database.oid = stv_tbl_perm.db_id
and datname = 'tickit'
group by datname, nspname, relname
order by datname, nspname, relname;
```

<table>
<thead>
<tr>
<th>datname</th>
<th>nspsname</th>
<th>relname</th>
<th>rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>tickit</td>
<td>public</td>
<td>category</td>
<td>11</td>
</tr>
<tr>
<td>tickit</td>
<td>public</td>
<td>date</td>
<td>365</td>
</tr>
<tr>
<td>tickit</td>
<td>public</td>
<td>event</td>
<td>8798</td>
</tr>
<tr>
<td>tickit</td>
<td>public</td>
<td>listing</td>
<td>192497</td>
</tr>
<tr>
<td>tickit</td>
<td>public</td>
<td>sales</td>
<td>172456</td>
</tr>
<tr>
<td>tickit</td>
<td>public</td>
<td>users</td>
<td>49990</td>
</tr>
<tr>
<td>tickit</td>
<td>public</td>
<td>venue</td>
<td>202</td>
</tr>
</tbody>
</table>
(7 rows)

List table IDs, data types, column names, and table names

The following query lists some information about each user table and its columns: the table ID, the table
name, its column names, and the data type of each column:

```
select distinct attrelid, rtrim(name), attname, typname
from pg_attribute a, pg_type t, stv_tbl_perm p
where t.oid=a.atttypid and a.attrelid=p.id
and a.attrelid between 100100 and 110000
and typname not in('oid','xid','tid','cid')
order by a.attrelid asc, typname, attname;
```

<table>
<thead>
<tr>
<th>attrelid</th>
<th>rtrim</th>
<th>attname</th>
<th>typname</th>
</tr>
</thead>
</table>
Count the number of data blocks for each column in a table

The following query joins the STV_BLOCKLIST table to PG_CLASS to return storage information for the columns in the SALES table.

```sql
select col, count(*)
from stv_blocklist s, pg_class p
where s.tbl=p.oid and relname='sales'
group by col
order by col;
```

<table>
<thead>
<tr>
<th>col</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>10</td>
<td>4</td>
</tr>
<tr>
<td>12</td>
<td>4</td>
</tr>
<tr>
<td>13</td>
<td>8</td>
</tr>
</tbody>
</table>

(13 rows)
Modifying the server configuration

You can make changes to the server configuration in the following ways:

- By using a SET (p. 389) command to override a setting for the duration of the current session only.

  For example:

  ```sql
  set extra_float_digits to 2;
  ```

- By modifying the parameter group settings for the cluster. The parameter group settings include additional parameters that you can configure. For more information, see Amazon Redshift Parameter Groups in the Amazon Redshift Management Guide.

Use the SHOW command to view the current parameter settings. Use SHOW ALL to view all the settings that you can configure by using the SET (p. 389) command.

```sql
show all;
```

<table>
<thead>
<tr>
<th>name</th>
<th>setting</th>
</tr>
</thead>
</table>

API Version 2012-12-01
693
**datestyle**

**Values (default in bold)**

Format specification (ISO, Postgres, SQL, or German), and year/month/day ordering (DMY, MDY, YMD).

ISO, MDY

**Description**

Sets the display format for date and time values as well as the rules for interpreting ambiguous date input values. The string contains two parameters that can be changed separately or together.

**Note**

The initdb command results in a setting that corresponds to the chosen lc_time locale.

**Example**

```
show datestyle;
DateStyle
--------
ISO, MDY
(1 row)

set datestyle to 'SQL,DMY';
```

**extra_float_digits**

**Values (default in bold)**

0, -15 to 2

**Description**

Sets the number of digits displayed for floating-point values, including float4 and float8. The value is added to the standard number of digits (FLT_DIG or DBL_DIG as appropriate). The value can be set as high as 2, to include partially significant digits; this is especially useful for outputting float data that needs to be restored exactly. Or it can be set negative to suppress unwanted digits.
**max_cursor_result_set_size**

**Values (default in bold)**

0 (defaults to maximum) - 14400000 MB

**Description**

Defines the maximum size of data, in megabytes, that can be returned per cursor result set of a larger query. This parameter value also affects the number of concurrent cursors for the cluster, enabling you to configure a value that increases or decreases the number of cursors for your cluster.

For more information, see `DECLARE (p. 328)` in this guide and `Configure Maximum Size of a Cursor Result Set` in the Amazon Redshift Cluster Management Guide.

**query_group**

**Values (default in bold)**

No default; the value can be any character string.

**Description**

This parameter applies a user-defined label to a group of queries that are run during the same session. This label is captured in the query logs and can be used to constrain results from the STL_QUERY and STV_INFLIGHT tables and the SVL_QLOG view. For example, you can apply a separate label to every query that you run to uniquely identify queries without having to look up their IDs.

This parameter does not exist in the server configuration file and must be set at runtime with a `SET` command. Although you can use a long character string as a label, the label is truncated to 30 characters in the LABEL column of the STL_QUERY table and the SVL_QLOG view (and to 15 characters in STV_INFLIGHT).

In the following example, `query_group` is set to Monday, then several queries are executed with that label:

```
set query_group to 'Monday';
SET
select * from category limit 1;
...
...
select query, pid, substring, elapsed, label
from svl_qlog where label = 'Monday'
order by query;
```

<table>
<thead>
<tr>
<th>query</th>
<th>pid</th>
<th>substring</th>
<th>elapsed</th>
<th>label</th>
</tr>
</thead>
<tbody>
<tr>
<td>789</td>
<td>6084</td>
<td><code>select * from category limit 1;</code></td>
<td>65468</td>
<td>Monday</td>
</tr>
<tr>
<td>790</td>
<td>6084</td>
<td><code>select query, trim(label) from ...</code></td>
<td>1260327</td>
<td>Monday</td>
</tr>
<tr>
<td>791</td>
<td>6084</td>
<td><code>select * from svl_qlog where ..</code></td>
<td>2293547</td>
<td>Monday</td>
</tr>
<tr>
<td>792</td>
<td>6084</td>
<td><code>select count(*) from bigsales;</code></td>
<td>108235617</td>
<td>Monday</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
search_path

Values (default in bold)

'S$user', public, schema_names

A comma-separated list of existing schema names. If 'S$user' is present, then the schema having the same name as SESSION_USER is substituted, otherwise it is ignored. If public is present and no schema with the name public exists, it is ignored.

Description

This parameter specifies the order in which schemas are searched when an object (such as a table or a function) is referenced by a simple name with no schema component.

- When objects are created without a specific target schema, they are placed in the first schema listed in the search path. If the search path is empty, the system returns an error.
- When objects with identical names exist in different schemas, the one found first in the search path is used.
- An object that is not in any of the schemas in the search path can only be referenced by specifying its containing schema with a qualified (dotted) name.
- The system catalog schema, pg_catalog, is always searched. If it is mentioned in the path, it is searched in the specified order. If not, it is searched before any of the path items.
- The current session's temporary-table schema, pg_temp.nnn, is always searched if it exists. It can be explicitly listed in the path by using the alias pg_temp. If it is not listed in the path, it is searched first (even before pg_catalog). However, the temporary schema is only searched for relation names (tables, views). It is not searched for function names.

Example

The following example creates the schema ENTERPRISE and sets the search_path to the new schema.

```sql
create schema enterprise;
set search_path to enterprise;
show search_path;

search_path
-------------
enterprise
(1 row)
```

The following example adds the schema ENTERPRISE to the default search_path.

```sql
set search_path to '$user', public, enterprise;
show search_path;

search_path
----------------
"$user", public, enterprise
(1 row)
```

The following example adds the table FRONTIER to the schema ENTERPRISE:
create table enterprise.frontier (c1 int);

When the table PUBLIC.FRONTIER is created in the same database, and the user does not specify the
schema name in a query, PUBLIC.FRONTIER takes precedence over ENTERPRISE.FRONTIER:

create table public.frontier(c1 int);
insert into enterprise.frontier values(1);
select * from frontier;

frontier
----
(0 rows)

select * from enterprise.frontier;
c1
----
1
(1 row)

statement_timeout

Values (default in bold)

0 (turns off limitation), x milliseconds

Description

Aborts any statement that takes over the specified number of milliseconds.

If WLM timeout (max_execution_time) is also specified as part of a WLM configuration, the lower of
statement_timeout and max_execution_time is used. For more information, see WLM timeout (p. 199).

Example

Because the following query takes longer than 1 millisecond, it times out and is cancelled.

set statement_timeout to 1;
select * from listing where listid>5000;
ERROR: Query (150) cancelled on user’s request

wlm_query_slot_count

Values (default in bold)

1, 1 to 50 (cannot exceed number of available slots (concurrency level) for the service class)
Description

Sets the number of query slots a query will use.

Workload management (WLM) reserves slots in a service class according to the concurrency level set for the queue (for example, if concurrency level is set to 5, then the service class has 5 slots). WLM allocates the available memory for a service class equally to each slot. For more information, see Implementing workload management (p. 196).

Note

If the value of wlm_query_slot_count is larger than the number of available slots (concurrency level) for the service class, the query will fail. If you encounter an error, decrease wlm_query_slot_count to an allowable value.

For operations where performance is heavily affected by the amount of memory allocated, such as Vacuum, increasing the value of wlm_query_slot_count can improve performance. In particular, for slow Vacuum commands, inspect the corresponding record in the SVV_VACUUM_SUMMARY view. If you see high values (close to or higher than 100) for sort_partitions and merge_increments in the SVV_VACUUM_SUMMARY view, consider increasing the value for wlm_query_slot_count the next time you run Vacuum against that table.

Increasing the value of wlm_query_slot_count limits the number of concurrent queries that can be run. For example, suppose the service class has a concurrency level of 5 and wlm_query_slot_count is set to 3. While a query is running within the session with wlm_query_slot_count set to 3, a maximum of 2 more concurrent queries can be executed within the same service class. Subsequent queries wait in the queue until currently executing queries complete and slots are freed.

Examples

Use the SET command to set the value of wlm_query_slot_count for the duration of the current session.

```
set wlm_query_slot_count to 3;
```
Sample Database

Most of the examples in the Amazon Redshift documentation use a sample database called TICKIT. This small database consists of seven tables: two fact tables and five dimensions.

This sample database application helps analysts track sales activity for the fictional TICKIT web site, where users buy and sell tickets online for sporting events, shows, and concerts. In particular, analysts
can identify ticket movement over time, success rates for sellers, and the best-selling events, venues, and seasons. Analysts can use this information to provide incentives to buyers and sellers who frequent the site, to attract new users, and to drive advertising and promotions.

For example, the following query finds the top five sellers in San Diego, based on the number of tickets sold in 2008:

```
select sellerid, username, (firstname || ' ' || lastname) as name,
city, sum(qtysold)
from sales, date, users
where sales.sellerid = users.userid
and sales.dateid = date.dateid
and year = 2008
and city = 'San Diego'
group by sellerid, username, name, city
order by 5 desc
limit 5;
```

<table>
<thead>
<tr>
<th>sellerid</th>
<th>username</th>
<th>name</th>
<th>city</th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>49977</td>
<td>JJK84WTE</td>
<td>Julie Hanson</td>
<td>San Diego</td>
<td>22</td>
</tr>
<tr>
<td>19750</td>
<td>AAS23BDR</td>
<td>Charity Zimmerman</td>
<td>San Diego</td>
<td>21</td>
</tr>
<tr>
<td>29069</td>
<td>SVL81MEQ</td>
<td>Axel Grant</td>
<td>San Diego</td>
<td>17</td>
</tr>
<tr>
<td>43632</td>
<td>VAG08HKW</td>
<td>Griffin Dodson</td>
<td>San Diego</td>
<td>16</td>
</tr>
<tr>
<td>36712</td>
<td>RXT40MKU</td>
<td>Hiram Turner</td>
<td>San Diego</td>
<td>14</td>
</tr>
</tbody>
</table>

The database used for the examples in this guide contains a small data set; the two fact tables each contain less than 200,000 rows, and the dimensions range from 11 rows in the CATEGORY table up to about 50,000 rows in the USERS table.

In particular, the database examples in this guide demonstrate the key features of Amazon Redshift table design:

- Data distribution
- Data sort
- Columnar compression

**CATEGORY table**

<table>
<thead>
<tr>
<th>Column name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CATID</td>
<td>SMALLINT</td>
<td>Primary key, a unique ID value for each row. Each row represents a specific type of event for which tickets are bought and sold.</td>
</tr>
<tr>
<td>CATGROUP</td>
<td>VARCHAR(10)</td>
<td>Descriptive name for a group of events, such as Shows and Sports.</td>
</tr>
<tr>
<td>CATNAME</td>
<td>VARCHAR(10)</td>
<td>Short descriptive name for a type of event within a group, such as Opera and Musicals.</td>
</tr>
<tr>
<td>CATDESC</td>
<td>VARCHAR(30)</td>
<td>Longer descriptive name for the type of event, such as Musical theatre.</td>
</tr>
</tbody>
</table>
# DATE table

<table>
<thead>
<tr>
<th>Column name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATEID</td>
<td>SMALLINT</td>
<td>Primary key, a unique ID value for each row. Each row represents a day in the calendar year.</td>
</tr>
<tr>
<td>CALDATE</td>
<td>DATE</td>
<td>Calendar date, such as 2008-06-24.</td>
</tr>
<tr>
<td>DAY</td>
<td>CHAR(3)</td>
<td>Day of week (short form), such as SA.</td>
</tr>
<tr>
<td>WEEK</td>
<td>SMALLINT</td>
<td>Week number, such as 26.</td>
</tr>
<tr>
<td>MONTH</td>
<td>CHAR(5)</td>
<td>Month name (short form), such as JUN.</td>
</tr>
<tr>
<td>QTR</td>
<td>CHAR(5)</td>
<td>Quarter number (1 through 4).</td>
</tr>
<tr>
<td>YEAR</td>
<td>SMALLINT</td>
<td>Four-digit year (2008).</td>
</tr>
<tr>
<td>HOLIDAY</td>
<td>BOOLEAN</td>
<td>Flag that denotes whether the day is a public holiday (U.S.).</td>
</tr>
</tbody>
</table>

# EVENT table

<table>
<thead>
<tr>
<th>Column name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVENTID</td>
<td>INTEGER</td>
<td>Primary key, a unique ID value for each row. Each row represents a separate event that takes place at a specific venue at a specific time.</td>
</tr>
<tr>
<td>VENUEID</td>
<td>SMALLINT</td>
<td>Foreign-key reference to the VENUE table.</td>
</tr>
<tr>
<td>CATID</td>
<td>SMALLINT</td>
<td>Foreign-key reference to the CATEGORY table.</td>
</tr>
<tr>
<td>DATEID</td>
<td>SMALLINT</td>
<td>Foreign-key reference to the DATE table.</td>
</tr>
<tr>
<td>EVENTNAME</td>
<td>VARCHAR(200)</td>
<td>Name of the event, such as Hamlet or La Traviata.</td>
</tr>
<tr>
<td>STARTTIME</td>
<td>TIMESTAMP</td>
<td>Full date and start time for the event, such as 2008-10-10 19:30:00.</td>
</tr>
</tbody>
</table>

# VENUE table

<table>
<thead>
<tr>
<th>Column name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VENUEID</td>
<td>SMALLINT</td>
<td>Primary key, a unique ID value for each row. Each row represents a specific venue where events take place.</td>
</tr>
<tr>
<td>VENUENAME</td>
<td>VARCHAR(100)</td>
<td>Exact name of the venue, such as Cleveland Browns Stadium.</td>
</tr>
<tr>
<td>VENUECITY</td>
<td>VARCHAR(30)</td>
<td>City name, such as Cleveland.</td>
</tr>
<tr>
<td>VENUESTATE</td>
<td>CHAR(2)</td>
<td>Two-letter state or province abbreviation (United States and Canada), such as OH.</td>
</tr>
</tbody>
</table>
# USERS table

<table>
<thead>
<tr>
<th>Column name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>USERID</td>
<td>INTEGER</td>
<td>Primary key, a unique ID value for each row. Each row represents a registered user (a buyer or seller or both) who has listed or bought tickets for at least one event.</td>
</tr>
<tr>
<td>USERNAME</td>
<td>CHAR(8)</td>
<td>An 8-character alphanumeric username, such as PGL08LJI.</td>
</tr>
<tr>
<td>FIRSTNAME</td>
<td>VARCHAR(30)</td>
<td>The user's first name, such as Victor.</td>
</tr>
<tr>
<td>LASTNAME</td>
<td>VARCHAR(30)</td>
<td>The user's last name, such as Hernandez.</td>
</tr>
<tr>
<td>CITY</td>
<td>VARCHAR(30)</td>
<td>The user's home city, such as Naperville.</td>
</tr>
<tr>
<td>STATE</td>
<td>CHAR(2)</td>
<td>The user's home state, such as GA.</td>
</tr>
<tr>
<td>EMAIL</td>
<td>VARCHAR(100)</td>
<td>The user's email address; this column contains random Latin values, such as <a href="mailto:turpis@accumsanlaoreet.org">turpis@accumsanlaoreet.org</a>.</td>
</tr>
<tr>
<td>PHONE</td>
<td>CHAR(14)</td>
<td>The user's 14-character phone number, such as (818) 765-4255.</td>
</tr>
<tr>
<td>LIKESPORTS, ...</td>
<td>BOOLEAN</td>
<td>A series of 10 different columns that identify the user's likes and dislikes with true and false values.</td>
</tr>
</tbody>
</table>

# LISTING table

<table>
<thead>
<tr>
<th>Column name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LISTID</td>
<td>INTEGER</td>
<td>Primary key, a unique ID value for each row. Each row represents a listing of a batch of tickets for a specific event.</td>
</tr>
<tr>
<td>SELLERID</td>
<td>INTEGER</td>
<td>Foreign-key reference to the USERS table, identifying the user who is selling the tickets.</td>
</tr>
<tr>
<td>EVENTID</td>
<td>INTEGER</td>
<td>Foreign-key reference to the EVENT table.</td>
</tr>
<tr>
<td>DATEID</td>
<td>SMALLINT</td>
<td>Foreign-key reference to the DATE table.</td>
</tr>
<tr>
<td>NUMTICKETS</td>
<td>SMALLINT</td>
<td>The number of tickets available for sale, such as 2 or 20.</td>
</tr>
<tr>
<td>PRICEPERTICKET</td>
<td>DECIMAL(8,2)</td>
<td>The fixed price of an individual ticket, such as 27.00 or 206.00.</td>
</tr>
<tr>
<td>TOTALPRICE</td>
<td>DECIMAL(8,2)</td>
<td>The total price for this listing (NUMTICKETS*PRICEPERTICKET).</td>
</tr>
</tbody>
</table>
**SALES table**

<table>
<thead>
<tr>
<th>Column name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SALESID</td>
<td>INTEGER</td>
<td>Primary key, a unique ID value for each row. Each row represents a sale of one or more tickets for a specific event, as offered in a specific listing.</td>
</tr>
<tr>
<td>LISTID</td>
<td>INTEGER</td>
<td>Foreign-key reference to the LISTING table.</td>
</tr>
<tr>
<td>SELLERID</td>
<td>INTEGER</td>
<td>Foreign-key reference to the USERS table (the user who sold the tickets).</td>
</tr>
<tr>
<td>BUYERID</td>
<td>INTEGER</td>
<td>Foreign-key reference to the USERS table (the user who bought the tickets).</td>
</tr>
<tr>
<td>EVENTID</td>
<td>INTEGER</td>
<td>Foreign-key reference to the EVENT table.</td>
</tr>
<tr>
<td>DATEID</td>
<td>SMALLINT</td>
<td>Foreign-key reference to the DATE table.</td>
</tr>
<tr>
<td>QTYSOLD</td>
<td>SMALLINT</td>
<td>The number of tickets that were sold, from 1 to 8. (A maximum of 8 tickets can be sold in a single transaction.)</td>
</tr>
<tr>
<td>PRICEPAID</td>
<td>DECIMAL(8,2)</td>
<td>The total price paid for the tickets, such as 75.00 or 488.00. The individual price of a ticket is PRICEPAID/QTYSOLD.</td>
</tr>
<tr>
<td>COMMISSION</td>
<td>DECIMAL(8,2)</td>
<td>The 15% commission that the business collects from the sale, such as 11.25 or 73.20. The seller receives 85% of the PRICEPAID value.</td>
</tr>
<tr>
<td>SALETIME</td>
<td>TIMESTAMP</td>
<td>The full date and time when the sale was completed, such as 2008-05-24 06:21:47.</td>
</tr>
</tbody>
</table>
Appendix: Time Zone Names and Abbreviations

Topics

• Time Zone Names (p. 704)
• Time Zone Abbreviations (p. 714)

The following lists contain all of the valid time zone names and time zone abbreviations that can be specified with the CONVERT_TIMEZONE function (p. 474).

Time Zone Names

The following list contains all of the valid time zone names that can be specified with the CONVERT_TIMEZONE function (p. 474).

Even though some of the time zone names in this list are capitalized initialisms or acronyms (for example; GB, PRC, ROK), the CONVERT_TIMEZONE function treats them as time zone names, not time zone abbreviations.

<table>
<thead>
<tr>
<th>Time Zone Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Africa/Abidjan</td>
</tr>
<tr>
<td>Africa/Accra</td>
</tr>
<tr>
<td>Africa/Addis_Ababa</td>
</tr>
<tr>
<td>Africa/Algiers</td>
</tr>
<tr>
<td>Africa/Africa/Ceuta</td>
</tr>
<tr>
<td>Africa/Conakry</td>
</tr>
<tr>
<td>Africa/Dakar</td>
</tr>
<tr>
<td>Africa/Asmara</td>
</tr>
<tr>
<td>Africa/Asmera</td>
</tr>
<tr>
<td>Africa/Bamako</td>
</tr>
<tr>
<td>Africa/Bangui</td>
</tr>
<tr>
<td>Africa/Banjul</td>
</tr>
<tr>
<td>Africa/Bissau</td>
</tr>
<tr>
<td>Africa/Blantyre</td>
</tr>
<tr>
<td>Africa/Brazzaville</td>
</tr>
<tr>
<td>Africa/Bujumbura</td>
</tr>
<tr>
<td>Time Zone Names</td>
</tr>
<tr>
<td>----------------</td>
</tr>
<tr>
<td>Africa/Cairo</td>
</tr>
<tr>
<td>Africa/Casablanca</td>
</tr>
<tr>
<td>/Dar_es_Salaam</td>
</tr>
<tr>
<td>Africa/Djibouti</td>
</tr>
<tr>
<td>Africa/Douala</td>
</tr>
<tr>
<td>Africa/El_Aaiun</td>
</tr>
<tr>
<td>Africa/Freetown</td>
</tr>
<tr>
<td>Africa/Gaborone</td>
</tr>
<tr>
<td>Africa/Harare</td>
</tr>
<tr>
<td>Africa/Johannesburg</td>
</tr>
<tr>
<td>Africa/Juba</td>
</tr>
<tr>
<td>Africa/Kampala</td>
</tr>
<tr>
<td>Africa/Khartoum</td>
</tr>
<tr>
<td>Africa/Kigali</td>
</tr>
<tr>
<td>Africa/Kinshasa</td>
</tr>
<tr>
<td>Africa/Lagos</td>
</tr>
<tr>
<td>Africa/Libreville</td>
</tr>
<tr>
<td>Africa/Lome</td>
</tr>
<tr>
<td>Africa/Luanda</td>
</tr>
<tr>
<td>Africa/Lubumbashi</td>
</tr>
<tr>
<td>Africa/Lusaka</td>
</tr>
<tr>
<td>Africa/Malabo</td>
</tr>
<tr>
<td>Africa/Maputo</td>
</tr>
<tr>
<td>Africa/Maseru</td>
</tr>
<tr>
<td>Africa/Mbabane</td>
</tr>
<tr>
<td>Africa/Mogadishu</td>
</tr>
<tr>
<td>Africa/Monrovia</td>
</tr>
<tr>
<td>Africa/Nairobi</td>
</tr>
<tr>
<td>Africa/NDjamena</td>
</tr>
<tr>
<td>Africa/Niamey</td>
</tr>
<tr>
<td>Africa/Nouakchott</td>
</tr>
<tr>
<td>Africa/Ouagadougou</td>
</tr>
<tr>
<td>Africa/Porto-Novo</td>
</tr>
<tr>
<td>Africa/Sao_Tome</td>
</tr>
<tr>
<td>Africa/Timbuktu</td>
</tr>
<tr>
<td>Africa/Tripoli</td>
</tr>
<tr>
<td>Africa/Tunis</td>
</tr>
<tr>
<td>Africa/Windhoek</td>
</tr>
<tr>
<td>America/Adak</td>
</tr>
<tr>
<td>America/Anchorage</td>
</tr>
<tr>
<td>America/Anguilla</td>
</tr>
<tr>
<td>America/Antigua</td>
</tr>
<tr>
<td>America/Araguaina</td>
</tr>
<tr>
<td>America/Argentina/Buenos_Aires</td>
</tr>
<tr>
<td>America/Argentina/Catamarca</td>
</tr>
<tr>
<td>America/Argentina/ComodRivadavia</td>
</tr>
<tr>
<td>America/Argentina/Cordoba</td>
</tr>
<tr>
<td>America/Argentina/Jujuy</td>
</tr>
<tr>
<td>America/Argentina/La_Rioja</td>
</tr>
<tr>
<td>America/Argentina/Mendoza</td>
</tr>
<tr>
<td>America/Argentina/Rio_Gallegos</td>
</tr>
<tr>
<td>America/Argentina/Salta</td>
</tr>
<tr>
<td>America/Argentina/San_Juan</td>
</tr>
<tr>
<td>America/Argentina/San_Luis</td>
</tr>
<tr>
<td>America/Argentina/Tucuman</td>
</tr>
<tr>
<td>America/Argentina/Ushuaia</td>
</tr>
<tr>
<td>America/Aruba</td>
</tr>
<tr>
<td>America/Asuncion</td>
</tr>
</tbody>
</table>
America/Atikokan
America/Atka
America/Bahia
America/Bahia_Banderas
America/Barbados
America/Belem
America/Belize
America/Blanc-Sablon
America/Boa_Vista
America/Bogota
America/Boise
America/Buenos_Aires
America/Cambridge_Bay
America/Campo_Grande
America/Cancun
America/Caracas
America/Catamarca
America/Cayenne
America/Cayman
America/Chicago
America/Chihuahua
America/Coral_Harbour
America/Cordoba
America/Costa_Rica
America/Creston
America/Cuiaba
America/Curacao
America/Danmarkshavn
America/Dawson
America/Dawson_Creek
America/Denver
America/Detroit
America/Dominica
America/Edmonton
America/Eirunepe
America/El_Salvador
America/Ensenada
America/Fort_Wayne
America/Fortaleza
America/Glace_Bay
America/Godthab
America/Goose_Bay
America/Grand_Turk
America/Grenada
America/Guadeloupe
America/Guatemala
America/Guayaquil
America/Guyana
America/Halifax
America/Havana
America/Hermosillo
America/Indiana/Indianapolis
America/Indiana/Knox
America/Indiana/Marengo
America/Indiana/Petersburg
America/Indiana/Tell_City
America/Indiana/Vevay
America/Indiana/Vincennes
<table>
<thead>
<tr>
<th>Time Zone Names</th>
</tr>
</thead>
<tbody>
<tr>
<td>America/Indiana/Winamac</td>
</tr>
<tr>
<td>America/Indianapolis</td>
</tr>
<tr>
<td>America/Inuvik</td>
</tr>
<tr>
<td>America/Iqaluit</td>
</tr>
<tr>
<td>America/Jamaica</td>
</tr>
<tr>
<td>America/Jujuy</td>
</tr>
<tr>
<td>America/Juneau</td>
</tr>
<tr>
<td>America/Kentucky/Louisville</td>
</tr>
<tr>
<td>America/Kentucky/Monticello</td>
</tr>
<tr>
<td>America/Knox_IN</td>
</tr>
<tr>
<td>America/Kralendijk</td>
</tr>
<tr>
<td>America/La_Paz</td>
</tr>
<tr>
<td>America/Lima</td>
</tr>
<tr>
<td>America/Los_Angeles</td>
</tr>
<tr>
<td>America/Louisville</td>
</tr>
<tr>
<td>America/Lower_Princes</td>
</tr>
<tr>
<td>America/Maceio</td>
</tr>
<tr>
<td>America/Managua</td>
</tr>
<tr>
<td>America/Manaus</td>
</tr>
<tr>
<td>America/Marigot</td>
</tr>
<tr>
<td>America/Martinique</td>
</tr>
<tr>
<td>America/Matamoros</td>
</tr>
<tr>
<td>America/Mazatlan</td>
</tr>
<tr>
<td>America/Mendoza</td>
</tr>
<tr>
<td>America/Menominee</td>
</tr>
<tr>
<td>America/Merida</td>
</tr>
<tr>
<td>America/Metlakatla</td>
</tr>
<tr>
<td>America/Mexico_City</td>
</tr>
<tr>
<td>America/Miquelon</td>
</tr>
<tr>
<td>America/Moncton</td>
</tr>
<tr>
<td>America/Monterrey</td>
</tr>
<tr>
<td>America/Montevideo</td>
</tr>
<tr>
<td>America/Montreal</td>
</tr>
<tr>
<td>America/Montserrat</td>
</tr>
<tr>
<td>America/Nassau</td>
</tr>
<tr>
<td>America/New_York</td>
</tr>
<tr>
<td>America/Nipigon</td>
</tr>
<tr>
<td>America/Nome</td>
</tr>
<tr>
<td>America/Noronha</td>
</tr>
<tr>
<td>America/North_Dakota/Beulah</td>
</tr>
<tr>
<td>America/North_Dakota/Center</td>
</tr>
<tr>
<td>America/North_Dakota/New_Salem</td>
</tr>
<tr>
<td>America/Ojinaga</td>
</tr>
<tr>
<td>America/Panama</td>
</tr>
<tr>
<td>America/Pangnirtung</td>
</tr>
<tr>
<td>America/Paramaribo</td>
</tr>
<tr>
<td>America/Phoenix</td>
</tr>
<tr>
<td>America/Port-au-Prince</td>
</tr>
<tr>
<td>America/Port_of_Spain</td>
</tr>
<tr>
<td>America/Porto_Acre</td>
</tr>
<tr>
<td>America/Porto_Velho</td>
</tr>
<tr>
<td>America/Puerto_Rico</td>
</tr>
<tr>
<td>America/Rainy_River</td>
</tr>
<tr>
<td>America/Rankin_Inlet</td>
</tr>
<tr>
<td>America/Recife</td>
</tr>
<tr>
<td>America/Regina</td>
</tr>
<tr>
<td>America/Resolute</td>
</tr>
<tr>
<td>America/Rio_Branco</td>
</tr>
<tr>
<td>Time Zone Name</td>
</tr>
<tr>
<td>----------------</td>
</tr>
<tr>
<td>America/Rosario</td>
</tr>
<tr>
<td>America/Santa_Isabel</td>
</tr>
<tr>
<td>America/Santarem</td>
</tr>
<tr>
<td>America/Santiago</td>
</tr>
<tr>
<td>America/Santo_Domingo</td>
</tr>
<tr>
<td>America/Sao_Paulo</td>
</tr>
<tr>
<td>America/Scoresbysund</td>
</tr>
<tr>
<td>America/Shiprock</td>
</tr>
<tr>
<td>America/Sitka</td>
</tr>
<tr>
<td>America/St_Barthelemy</td>
</tr>
<tr>
<td>America/St_Johns</td>
</tr>
<tr>
<td>America/St_Kitts</td>
</tr>
<tr>
<td>America/St_Lucia</td>
</tr>
<tr>
<td>America/St_Thomas</td>
</tr>
<tr>
<td>America/St_Vincent</td>
</tr>
<tr>
<td>America/Swift_Current</td>
</tr>
<tr>
<td>America/Tegucigalpa</td>
</tr>
<tr>
<td>America/Thule</td>
</tr>
<tr>
<td>America/Thunder_Bay</td>
</tr>
<tr>
<td>America/Tijuana</td>
</tr>
<tr>
<td>America/Toronto</td>
</tr>
<tr>
<td>America/Tortola</td>
</tr>
<tr>
<td>America/Vancouver</td>
</tr>
<tr>
<td>America/Virgin</td>
</tr>
<tr>
<td>America/Whitehorse</td>
</tr>
<tr>
<td>America/Winnipeg</td>
</tr>
<tr>
<td>America/Yakutat</td>
</tr>
<tr>
<td>America/Yellowknife</td>
</tr>
<tr>
<td>Antarctica/Casey</td>
</tr>
<tr>
<td>Antarctica/Davis</td>
</tr>
<tr>
<td>Antarctica/DumontDUrville</td>
</tr>
<tr>
<td>Antarctica/Macquarie</td>
</tr>
<tr>
<td>Antarctica/Mawson</td>
</tr>
<tr>
<td>Antarctica/McMurdo</td>
</tr>
<tr>
<td>Antarctica/Palmer</td>
</tr>
<tr>
<td>Antarctica/Rothera</td>
</tr>
<tr>
<td>Antarctica/South_Pole</td>
</tr>
<tr>
<td>Antarctica/Syowa</td>
</tr>
<tr>
<td>Antarctica/Vostok</td>
</tr>
<tr>
<td>Arctic/Longyearbyen</td>
</tr>
<tr>
<td>Asia/Aden</td>
</tr>
<tr>
<td>Asia/Almaty</td>
</tr>
<tr>
<td>Asia/Amman</td>
</tr>
<tr>
<td>Asia/Anadyr</td>
</tr>
<tr>
<td>Asia/Aqtau</td>
</tr>
<tr>
<td>Asia/Aqtobe</td>
</tr>
<tr>
<td>Asia/Ashgabat</td>
</tr>
<tr>
<td>Asia/Ashkhabad</td>
</tr>
<tr>
<td>Asia/Baghdad</td>
</tr>
<tr>
<td>Asia/Bahrain</td>
</tr>
<tr>
<td>Asia/Baku</td>
</tr>
<tr>
<td>Asia/Bangkok</td>
</tr>
<tr>
<td>Asia/Beirut</td>
</tr>
<tr>
<td>Asia/Bishkek</td>
</tr>
<tr>
<td>Asia/Brunei</td>
</tr>
<tr>
<td>Asia/Calcutta</td>
</tr>
<tr>
<td>Asia/Choibalsan</td>
</tr>
<tr>
<td>Asia/Chongqing</td>
</tr>
<tr>
<td>Time Zone Name</td>
</tr>
<tr>
<td>---------------</td>
</tr>
<tr>
<td>Asia/Chungking</td>
</tr>
<tr>
<td>Asia/Colombo</td>
</tr>
<tr>
<td>Asia/Dacca</td>
</tr>
<tr>
<td>Asia/Damascus</td>
</tr>
<tr>
<td>Asia/Dhaka</td>
</tr>
<tr>
<td>Asia/Dili</td>
</tr>
<tr>
<td>Asia/Dubai</td>
</tr>
<tr>
<td>Asia/Dushanbe</td>
</tr>
<tr>
<td>Asia/Gaza</td>
</tr>
<tr>
<td>Asia/Harbin</td>
</tr>
<tr>
<td>Asia/Hebron</td>
</tr>
<tr>
<td>Asia/Ho_Chi_Minh</td>
</tr>
<tr>
<td>Asia/Hong_Kong</td>
</tr>
<tr>
<td>Asia/Hovd</td>
</tr>
<tr>
<td>Asia/Irkutsk</td>
</tr>
<tr>
<td>Asia/Istanbul</td>
</tr>
<tr>
<td>Asia/Jakarta</td>
</tr>
<tr>
<td>Asia/Jayapura</td>
</tr>
<tr>
<td>Asia/Jerusalem</td>
</tr>
<tr>
<td>Asia/Kabul</td>
</tr>
<tr>
<td>Asia/Kamchatka</td>
</tr>
<tr>
<td>Asia/Karachi</td>
</tr>
<tr>
<td>Asia/Kashgar</td>
</tr>
<tr>
<td>Asia/Kathmandu</td>
</tr>
<tr>
<td>Asia/Katmandu</td>
</tr>
<tr>
<td>Asia/Khandyga</td>
</tr>
<tr>
<td>Asia/Kolkata</td>
</tr>
<tr>
<td>Asia/Krasnoyarsk</td>
</tr>
<tr>
<td>Asia/Kuala_Lumpur</td>
</tr>
<tr>
<td>Asia/Kuching</td>
</tr>
<tr>
<td>Asia/Kuwait</td>
</tr>
<tr>
<td>Asia/Macao</td>
</tr>
<tr>
<td>Asia/Macau</td>
</tr>
<tr>
<td>Asia/Magadan</td>
</tr>
<tr>
<td>Asia/Makassar</td>
</tr>
<tr>
<td>Asia/Manila</td>
</tr>
<tr>
<td>Asia/Muscat</td>
</tr>
<tr>
<td>Asia/Nicosia</td>
</tr>
<tr>
<td>Asia/Novokuznetsk</td>
</tr>
<tr>
<td>Asia/Novosibirsk</td>
</tr>
<tr>
<td>Asia/Phnom_Penh</td>
</tr>
<tr>
<td>Asia/Pontianak</td>
</tr>
<tr>
<td>Asia/Pyongyang</td>
</tr>
<tr>
<td>Asia/Qatar</td>
</tr>
<tr>
<td>Asia/Qyzylorda</td>
</tr>
<tr>
<td>Asia/Rangoon</td>
</tr>
<tr>
<td>Asia/Riyadh</td>
</tr>
<tr>
<td>Asia/Riyadh87</td>
</tr>
<tr>
<td>Asia/Riyadh88</td>
</tr>
<tr>
<td>Asia/Riyadh89</td>
</tr>
<tr>
<td>Asia/Saigon</td>
</tr>
<tr>
<td>Asia/Sakhalin</td>
</tr>
<tr>
<td>Asia/Samarkand</td>
</tr>
<tr>
<td>Asia/Seoul</td>
</tr>
<tr>
<td>Asia/Shanghai</td>
</tr>
<tr>
<td>Asia/Singapore</td>
</tr>
<tr>
<td>Time Zone Name</td>
</tr>
<tr>
<td>----------------</td>
</tr>
<tr>
<td>Asia/Taipei</td>
</tr>
<tr>
<td>Asia/Tashkent</td>
</tr>
<tr>
<td>Asia/Tbilisi</td>
</tr>
<tr>
<td>Asia/Tehran</td>
</tr>
<tr>
<td>Asia/Tel_Aviv</td>
</tr>
<tr>
<td>Asia/Thimbu</td>
</tr>
<tr>
<td>Asia/Thimphu</td>
</tr>
<tr>
<td>Asia/Tokyo</td>
</tr>
<tr>
<td>Asia/U'jung_Pandang</td>
</tr>
<tr>
<td>Asia/Ulaanbaatar</td>
</tr>
<tr>
<td>Asia/Ulan_Bator</td>
</tr>
<tr>
<td>Asia/Urumqi</td>
</tr>
<tr>
<td>Asia/Ust-Nera</td>
</tr>
<tr>
<td>Asia/Vientiane</td>
</tr>
<tr>
<td>Asia/Vladivostok</td>
</tr>
<tr>
<td>Asia/Yakutsk</td>
</tr>
<tr>
<td>Asia/Yekaterinburg</td>
</tr>
<tr>
<td>Asia/Yerevan</td>
</tr>
<tr>
<td>Atlantic/Azores</td>
</tr>
<tr>
<td>Atlantic/Bermuda</td>
</tr>
<tr>
<td>Atlantic/Canary</td>
</tr>
<tr>
<td>Atlantic/Cape_Verde</td>
</tr>
<tr>
<td>Atlantic/Faeroe</td>
</tr>
<tr>
<td>Atlantic/Faroe</td>
</tr>
<tr>
<td>Atlantic/Jan_Mayen</td>
</tr>
<tr>
<td>Atlantic/Madeira</td>
</tr>
<tr>
<td>Atlantic/Reykjavik</td>
</tr>
<tr>
<td>Atlantic/South_Georgia</td>
</tr>
<tr>
<td>Atlantic/St_Helena</td>
</tr>
<tr>
<td>Atlantic/Stanley</td>
</tr>
<tr>
<td>Australia/ACT</td>
</tr>
<tr>
<td>Australia/Adelaide</td>
</tr>
<tr>
<td>Australia/Brisbane</td>
</tr>
<tr>
<td>Australia/Broken_Hill</td>
</tr>
<tr>
<td>Australia/Canberra</td>
</tr>
<tr>
<td>Australia/Currie</td>
</tr>
<tr>
<td>Australia/Darwin</td>
</tr>
<tr>
<td>Australia/Eucla</td>
</tr>
<tr>
<td>Australia/Hobart</td>
</tr>
<tr>
<td>Australia/LHI</td>
</tr>
<tr>
<td>Australia/Lindeman</td>
</tr>
<tr>
<td>Australia/Lord_Howe</td>
</tr>
<tr>
<td>Australia/Melbourne</td>
</tr>
<tr>
<td>Australia/NSW</td>
</tr>
<tr>
<td>Australia/North</td>
</tr>
<tr>
<td>Australia/Perth</td>
</tr>
<tr>
<td>Australia/Queensland</td>
</tr>
<tr>
<td>Australia/South</td>
</tr>
<tr>
<td>Australia/Sydney</td>
</tr>
<tr>
<td>Australia/Tasmania</td>
</tr>
<tr>
<td>Australia/Victoria</td>
</tr>
<tr>
<td>Australia/West</td>
</tr>
<tr>
<td>Australia/Yancowinna</td>
</tr>
<tr>
<td>Brazil/Acre</td>
</tr>
<tr>
<td>Brazil/DeNoronha</td>
</tr>
<tr>
<td>Brazil/East</td>
</tr>
<tr>
<td>Brazil/West</td>
</tr>
</tbody>
</table>

API Version 2012-12-01
710
<table>
<thead>
<tr>
<th>Time Zone</th>
</tr>
</thead>
<tbody>
<tr>
<td>Canada/Atlantic</td>
</tr>
<tr>
<td>Canada/Central</td>
</tr>
<tr>
<td>Canada/East-Saskatchewan</td>
</tr>
<tr>
<td>Canada/Eastern</td>
</tr>
<tr>
<td>Canada/Mountain</td>
</tr>
<tr>
<td>Canada/Newfoundland</td>
</tr>
<tr>
<td>Canada/Pacific</td>
</tr>
<tr>
<td>Canada/Saskatchewan</td>
</tr>
<tr>
<td>Canada/Yukon</td>
</tr>
<tr>
<td>Chile/Continental</td>
</tr>
<tr>
<td>Chile/EasterIsland</td>
</tr>
<tr>
<td>Cuba</td>
</tr>
<tr>
<td>EST5EDT</td>
</tr>
<tr>
<td>Egypt</td>
</tr>
<tr>
<td>Eire</td>
</tr>
<tr>
<td>Europe/Amsterdam</td>
</tr>
<tr>
<td>Europe/Andorra</td>
</tr>
<tr>
<td>Europe/Athens</td>
</tr>
<tr>
<td>Europe/Belfast</td>
</tr>
<tr>
<td>Europe/Belgrade</td>
</tr>
<tr>
<td>Europe/Berlin</td>
</tr>
<tr>
<td>Europe/Bratislava</td>
</tr>
<tr>
<td>Europe/Brussels</td>
</tr>
<tr>
<td>Europe/Bucharest</td>
</tr>
<tr>
<td>Europe/Budapest</td>
</tr>
<tr>
<td>Europe/Busingen</td>
</tr>
<tr>
<td>Europe/Chisinau</td>
</tr>
<tr>
<td>Europe/Copenhagen</td>
</tr>
<tr>
<td>Europe/Dublin</td>
</tr>
<tr>
<td>Europe/Gibraltar</td>
</tr>
<tr>
<td>Europe/Guernsey</td>
</tr>
<tr>
<td>Europe/Helsinki</td>
</tr>
<tr>
<td>Europe/Isle_of_Man</td>
</tr>
<tr>
<td>Europe/Istanbul</td>
</tr>
<tr>
<td>Europe/Jersey</td>
</tr>
<tr>
<td>Europe/Kaliningrad</td>
</tr>
<tr>
<td>Europe/Kiev</td>
</tr>
<tr>
<td>Europe/Lisbon</td>
</tr>
<tr>
<td>Europe/Ljubljana</td>
</tr>
<tr>
<td>Europe/London</td>
</tr>
<tr>
<td>Europe/Luxembourg</td>
</tr>
<tr>
<td>Europe/Madrid</td>
</tr>
<tr>
<td>Europe/Malta</td>
</tr>
<tr>
<td>Europe/Mariehamn</td>
</tr>
<tr>
<td>Europe/Minsk</td>
</tr>
<tr>
<td>Europe/Monaco</td>
</tr>
<tr>
<td>Europe/Moscow</td>
</tr>
<tr>
<td>Europe/Nicosia</td>
</tr>
<tr>
<td>Europe/Oslo</td>
</tr>
<tr>
<td>Europe/Paris</td>
</tr>
<tr>
<td>Europe/Podgorica</td>
</tr>
<tr>
<td>Europe/Prague</td>
</tr>
<tr>
<td>Europe/Riga</td>
</tr>
<tr>
<td>Europe/Rome</td>
</tr>
<tr>
<td>Europe/Samara</td>
</tr>
<tr>
<td>Europe/San_Marino</td>
</tr>
<tr>
<td>Europe/Sarajevo</td>
</tr>
<tr>
<td>Europe/Simferopol</td>
</tr>
<tr>
<td>Time Zone</td>
</tr>
<tr>
<td>-------------------</td>
</tr>
<tr>
<td>Europe/Skopje</td>
</tr>
<tr>
<td>Europe/Sofia</td>
</tr>
<tr>
<td>Europe/Stockholm</td>
</tr>
<tr>
<td>Europe/Tallinn</td>
</tr>
<tr>
<td>Europe/Tirane</td>
</tr>
<tr>
<td>Europe/Tiraspol</td>
</tr>
<tr>
<td>Europe/Uzhgorod</td>
</tr>
<tr>
<td>Europe/Vaduz</td>
</tr>
<tr>
<td>Europe/Vatican</td>
</tr>
<tr>
<td>Europe/Vienna</td>
</tr>
<tr>
<td>Europe/Vilnius</td>
</tr>
<tr>
<td>Europe/Volgograd</td>
</tr>
<tr>
<td>Europe/Warsaw</td>
</tr>
<tr>
<td>Europe/Zagreb</td>
</tr>
<tr>
<td>Europe/Zaporozhye</td>
</tr>
<tr>
<td>Europe/Zurich</td>
</tr>
<tr>
<td>GB</td>
</tr>
<tr>
<td>GB-Eire</td>
</tr>
<tr>
<td>GMT+0</td>
</tr>
<tr>
<td>GMT-0</td>
</tr>
<tr>
<td>GMT0</td>
</tr>
<tr>
<td>Greenwich</td>
</tr>
<tr>
<td>Hongkong</td>
</tr>
<tr>
<td>Iceland</td>
</tr>
<tr>
<td>Indian/Antananarivo</td>
</tr>
<tr>
<td>Indian/Chagos</td>
</tr>
<tr>
<td>Indian/Christmas</td>
</tr>
<tr>
<td>Indian/Cocos</td>
</tr>
<tr>
<td>Indian/Comoro</td>
</tr>
<tr>
<td>Indian/Kerguelen</td>
</tr>
<tr>
<td>Indian/Mahe</td>
</tr>
<tr>
<td>Indian/Maldives</td>
</tr>
<tr>
<td>Indian/Mauritius</td>
</tr>
<tr>
<td>Indian/Mayotte</td>
</tr>
<tr>
<td>Indian/Reunion</td>
</tr>
<tr>
<td>Iran</td>
</tr>
<tr>
<td>Israel</td>
</tr>
<tr>
<td>Jamaica</td>
</tr>
<tr>
<td>Japan</td>
</tr>
<tr>
<td>Kwajalein</td>
</tr>
<tr>
<td>Libya</td>
</tr>
<tr>
<td>MST7MDT</td>
</tr>
<tr>
<td>Mexico/BajaNorte</td>
</tr>
<tr>
<td>Mexico/BajaSur</td>
</tr>
<tr>
<td>Mexico/General</td>
</tr>
<tr>
<td>Mideast/Riyadh87</td>
</tr>
<tr>
<td>Mideast/Riyadh88</td>
</tr>
<tr>
<td>Mideast/Riyadh89</td>
</tr>
<tr>
<td>NZ</td>
</tr>
<tr>
<td>NZ-CHAT</td>
</tr>
<tr>
<td>Navajo</td>
</tr>
<tr>
<td>PRC</td>
</tr>
<tr>
<td>PST8PDT</td>
</tr>
<tr>
<td>Pacific/Apiap</td>
</tr>
<tr>
<td>Pacific/Auckland</td>
</tr>
<tr>
<td>Pacific/Chatham</td>
</tr>
<tr>
<td>Pacific/Chuuk</td>
</tr>
<tr>
<td>Pacific/Easter</td>
</tr>
<tr>
<td>Time Zone Names</td>
</tr>
<tr>
<td>----------------</td>
</tr>
<tr>
<td>Pacific/Efate</td>
</tr>
<tr>
<td>Pacific/Enderbury</td>
</tr>
<tr>
<td>Pacific/Fakaofo</td>
</tr>
<tr>
<td>Pacific/Fiji</td>
</tr>
<tr>
<td>Pacific/Funafuti</td>
</tr>
<tr>
<td>Pacific/Galapagos</td>
</tr>
<tr>
<td>Pacific/Gambier</td>
</tr>
<tr>
<td>Pacific/Guadalcanal</td>
</tr>
<tr>
<td>Pacific/Guam</td>
</tr>
<tr>
<td>Pacific/Honolulu</td>
</tr>
<tr>
<td>Pacific/Johnston</td>
</tr>
<tr>
<td>Pacific/Kirimiti</td>
</tr>
<tr>
<td>Pacific/Kosrae</td>
</tr>
<tr>
<td>Pacific/Kwajalein</td>
</tr>
<tr>
<td>Pacific/Majuro</td>
</tr>
<tr>
<td>Pacific/Marquesas</td>
</tr>
<tr>
<td>Pacific/Midway</td>
</tr>
<tr>
<td>Pacific/Nauru</td>
</tr>
<tr>
<td>Pacific/Niue</td>
</tr>
<tr>
<td>Pacific/Norfolk</td>
</tr>
<tr>
<td>Pacific/Noumea</td>
</tr>
<tr>
<td>Pacific/Pago_Pago</td>
</tr>
<tr>
<td>Pacific/Palau</td>
</tr>
<tr>
<td>Pacific/Pitcairn</td>
</tr>
<tr>
<td>Pacific/Pohnpei</td>
</tr>
<tr>
<td>Pacific/Ponape</td>
</tr>
<tr>
<td>Pacific/Port_Moresby</td>
</tr>
<tr>
<td>Pacific/Rarotonga</td>
</tr>
<tr>
<td>Pacific/Saipan</td>
</tr>
<tr>
<td>Pacific/Samoa</td>
</tr>
<tr>
<td>Pacific/Tahiti</td>
</tr>
<tr>
<td>Pacific/Tarawa</td>
</tr>
<tr>
<td>Pacific/Tongatapu</td>
</tr>
<tr>
<td>Pacific/Truk</td>
</tr>
<tr>
<td>Pacific/Wake</td>
</tr>
<tr>
<td>Pacific/Wallis</td>
</tr>
<tr>
<td>Pacific/Yap</td>
</tr>
<tr>
<td>Poland</td>
</tr>
<tr>
<td>Portugal</td>
</tr>
<tr>
<td>ROC</td>
</tr>
<tr>
<td>ROK</td>
</tr>
<tr>
<td>Singapore</td>
</tr>
<tr>
<td>Turkey</td>
</tr>
<tr>
<td>US/Alaska</td>
</tr>
<tr>
<td>US/Aleutian</td>
</tr>
<tr>
<td>US/Arizona</td>
</tr>
<tr>
<td>US/Central</td>
</tr>
<tr>
<td>US/East-Indiana</td>
</tr>
<tr>
<td>US/Eastern</td>
</tr>
<tr>
<td>US/Hawaii</td>
</tr>
<tr>
<td>US/Indiana-Starke</td>
</tr>
<tr>
<td>US/Michigan</td>
</tr>
<tr>
<td>US/Mountain</td>
</tr>
<tr>
<td>US/Pacific</td>
</tr>
<tr>
<td>US/Pacific-New</td>
</tr>
<tr>
<td>US/Samoa</td>
</tr>
<tr>
<td>Universal</td>
</tr>
<tr>
<td>W-SU</td>
</tr>
</tbody>
</table>
# Time Zone Abbreviations

The following list contains all of the valid time zone abbreviations that can be specified with the `CONVERT_TIMEZONE` function (p. 474).

<table>
<thead>
<tr>
<th>Abbreviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACSST</td>
</tr>
<tr>
<td>ACST</td>
</tr>
<tr>
<td>ACT</td>
</tr>
<tr>
<td>ADT</td>
</tr>
<tr>
<td>AESST</td>
</tr>
<tr>
<td>AEST</td>
</tr>
<tr>
<td>AFT</td>
</tr>
<tr>
<td>AKDT</td>
</tr>
<tr>
<td>AKST</td>
</tr>
<tr>
<td>ALMST</td>
</tr>
<tr>
<td>ALMT</td>
</tr>
<tr>
<td>AMST</td>
</tr>
<tr>
<td>AMT</td>
</tr>
<tr>
<td>ANAST</td>
</tr>
<tr>
<td>ANAT</td>
</tr>
<tr>
<td>ARST</td>
</tr>
<tr>
<td>ART</td>
</tr>
<tr>
<td>AST</td>
</tr>
<tr>
<td>AWSST</td>
</tr>
<tr>
<td>AWST</td>
</tr>
<tr>
<td>AZOST</td>
</tr>
<tr>
<td>AZOT</td>
</tr>
<tr>
<td>AZST</td>
</tr>
<tr>
<td>AZT</td>
</tr>
<tr>
<td>BDST</td>
</tr>
<tr>
<td>BDT</td>
</tr>
<tr>
<td>BNT</td>
</tr>
<tr>
<td>BORT</td>
</tr>
<tr>
<td>BOT</td>
</tr>
<tr>
<td>BRA</td>
</tr>
<tr>
<td>BRST</td>
</tr>
<tr>
<td>BRT</td>
</tr>
<tr>
<td>BST</td>
</tr>
<tr>
<td>BTT</td>
</tr>
<tr>
<td>CADT</td>
</tr>
<tr>
<td>CAST</td>
</tr>
<tr>
<td>CCT</td>
</tr>
<tr>
<td>CDT</td>
</tr>
<tr>
<td>CEST</td>
</tr>
<tr>
<td>CET</td>
</tr>
<tr>
<td>CETDST</td>
</tr>
<tr>
<td>CHADT</td>
</tr>
<tr>
<td>CHAST</td>
</tr>
<tr>
<td>CHUT</td>
</tr>
<tr>
<td>CKT</td>
</tr>
<tr>
<td>CKT</td>
</tr>
<tr>
<td>CLST</td>
</tr>
<tr>
<td>CLT</td>
</tr>
<tr>
<td>COT</td>
</tr>
<tr>
<td>CST</td>
</tr>
<tr>
<td>CXT</td>
</tr>
<tr>
<td>DAVT</td>
</tr>
<tr>
<td>DDUT</td>
</tr>
<tr>
<td>Time Zone Abbreviation</td>
</tr>
<tr>
<td>------------------------</td>
</tr>
<tr>
<td>EASST</td>
</tr>
<tr>
<td>EAST</td>
</tr>
<tr>
<td>EAT</td>
</tr>
<tr>
<td>EDT</td>
</tr>
<tr>
<td>EEST</td>
</tr>
<tr>
<td>EET</td>
</tr>
<tr>
<td>EETDST</td>
</tr>
<tr>
<td>EGST</td>
</tr>
<tr>
<td>EGT</td>
</tr>
<tr>
<td>EST</td>
</tr>
<tr>
<td>EUT</td>
</tr>
<tr>
<td>FJST</td>
</tr>
<tr>
<td>FJT</td>
</tr>
<tr>
<td>FKST</td>
</tr>
<tr>
<td>FKT</td>
</tr>
<tr>
<td>FNST</td>
</tr>
<tr>
<td>FNT</td>
</tr>
<tr>
<td>GALT</td>
</tr>
<tr>
<td>GAMT</td>
</tr>
<tr>
<td>GEST</td>
</tr>
<tr>
<td>GET</td>
</tr>
<tr>
<td>GFT</td>
</tr>
<tr>
<td>GILT</td>
</tr>
<tr>
<td>GMT</td>
</tr>
<tr>
<td>GYT</td>
</tr>
<tr>
<td>HKT</td>
</tr>
<tr>
<td>HST</td>
</tr>
<tr>
<td>ICT</td>
</tr>
<tr>
<td>IDT</td>
</tr>
<tr>
<td>IOT</td>
</tr>
<tr>
<td>IRKST</td>
</tr>
<tr>
<td>IRKT</td>
</tr>
<tr>
<td>IRT</td>
</tr>
<tr>
<td>IST</td>
</tr>
<tr>
<td>JAYT</td>
</tr>
<tr>
<td>JST</td>
</tr>
<tr>
<td>KDT</td>
</tr>
<tr>
<td>KGST</td>
</tr>
<tr>
<td>KGT</td>
</tr>
<tr>
<td>KST</td>
</tr>
<tr>
<td>KST</td>
</tr>
<tr>
<td>LHDT</td>
</tr>
<tr>
<td>LHST</td>
</tr>
<tr>
<td>LIGT</td>
</tr>
<tr>
<td>LINT</td>
</tr>
<tr>
<td>LKT</td>
</tr>
<tr>
<td>MAGST</td>
</tr>
<tr>
<td>MAGT</td>
</tr>
<tr>
<td>MART</td>
</tr>
<tr>
<td>MAWT</td>
</tr>
<tr>
<td>MDT</td>
</tr>
<tr>
<td>MEST</td>
</tr>
<tr>
<td>MET</td>
</tr>
<tr>
<td>METDST</td>
</tr>
<tr>
<td>MEZ</td>
</tr>
<tr>
<td>MHT</td>
</tr>
<tr>
<td>Time Zone Abbreviations</td>
</tr>
<tr>
<td>-------------------------</td>
</tr>
<tr>
<td>MMT</td>
</tr>
<tr>
<td>MPT</td>
</tr>
<tr>
<td>MSD</td>
</tr>
<tr>
<td>MSK</td>
</tr>
<tr>
<td>MST</td>
</tr>
<tr>
<td>MUST</td>
</tr>
<tr>
<td>MUT</td>
</tr>
<tr>
<td>MVT</td>
</tr>
<tr>
<td>MYT</td>
</tr>
<tr>
<td>NDT</td>
</tr>
<tr>
<td>NFT</td>
</tr>
<tr>
<td>NOVST</td>
</tr>
<tr>
<td>NOVT</td>
</tr>
<tr>
<td>NPT</td>
</tr>
<tr>
<td>NST</td>
</tr>
<tr>
<td>NUT</td>
</tr>
<tr>
<td>NZDT</td>
</tr>
<tr>
<td>NZST</td>
</tr>
<tr>
<td>NZT</td>
</tr>
<tr>
<td>OMSST</td>
</tr>
<tr>
<td>OMST</td>
</tr>
<tr>
<td>PDT</td>
</tr>
<tr>
<td>PET</td>
</tr>
<tr>
<td>PETST</td>
</tr>
<tr>
<td>PETT</td>
</tr>
<tr>
<td>PGT</td>
</tr>
<tr>
<td>PHOT</td>
</tr>
<tr>
<td>PHT</td>
</tr>
<tr>
<td>PKST</td>
</tr>
<tr>
<td>PKT</td>
</tr>
<tr>
<td>PMDT</td>
</tr>
<tr>
<td>PMST</td>
</tr>
<tr>
<td>PONT</td>
</tr>
<tr>
<td>PST</td>
</tr>
<tr>
<td>PWT</td>
</tr>
<tr>
<td>PYST</td>
</tr>
<tr>
<td>PYT</td>
</tr>
<tr>
<td>RET</td>
</tr>
<tr>
<td>SADT</td>
</tr>
<tr>
<td>SAST</td>
</tr>
<tr>
<td>SCT</td>
</tr>
<tr>
<td>SGT</td>
</tr>
<tr>
<td>TAHT</td>
</tr>
<tr>
<td>THT</td>
</tr>
<tr>
<td>TKT</td>
</tr>
<tr>
<td>TMT</td>
</tr>
<tr>
<td>TOT</td>
</tr>
<tr>
<td>TRUT</td>
</tr>
<tr>
<td>TVT</td>
</tr>
<tr>
<td>UCT</td>
</tr>
<tr>
<td>ULAST</td>
</tr>
<tr>
<td>ULAT</td>
</tr>
<tr>
<td>UT</td>
</tr>
<tr>
<td>UTC</td>
</tr>
<tr>
<td>UYST</td>
</tr>
<tr>
<td>UYT</td>
</tr>
<tr>
<td>UZST</td>
</tr>
<tr>
<td>Abbreviation</td>
</tr>
<tr>
<td>--------------</td>
</tr>
<tr>
<td>UZT</td>
</tr>
<tr>
<td>VET</td>
</tr>
<tr>
<td>VLAST</td>
</tr>
<tr>
<td>VLAT</td>
</tr>
<tr>
<td>VOLT</td>
</tr>
<tr>
<td>VUT</td>
</tr>
<tr>
<td>WADT</td>
</tr>
<tr>
<td>WAKT</td>
</tr>
<tr>
<td>WAST</td>
</tr>
<tr>
<td>WAT</td>
</tr>
<tr>
<td>WDT</td>
</tr>
<tr>
<td>WET</td>
</tr>
<tr>
<td>WEDST</td>
</tr>
<tr>
<td>WFT</td>
</tr>
<tr>
<td>WGST</td>
</tr>
<tr>
<td>WGT</td>
</tr>
<tr>
<td>YAKST</td>
</tr>
<tr>
<td>YAKT</td>
</tr>
<tr>
<td>YAPT</td>
</tr>
<tr>
<td>YEKST</td>
</tr>
<tr>
<td>YEKT</td>
</tr>
<tr>
<td>Z</td>
</tr>
<tr>
<td>ZULU</td>
</tr>
</tbody>
</table>
# Document History

The following table describes the important changes since the last release of the *Amazon Redshift Developer Guide*.

## API version: 2012-12-01

## Latest documentation update: July 7, 2014

For a list of the changes to the *Amazon Redshift Management Guide*, see [Amazon Redshift Management Guide Document History](https://docs.aws.amazon.com/redshift/latest/mngguide/).

<table>
<thead>
<tr>
<th>Change</th>
<th>Description</th>
<th>Date Changed</th>
</tr>
</thead>
<tbody>
<tr>
<td>New Troubleshooting Queries</td>
<td>Troubleshooting Queries (p. 181) provides a quick reference for identifying and addressing some of the most common and most serious issues you are likely to encounter with Amazon Redshift queries.</td>
<td>July 7, 2014</td>
</tr>
<tr>
<td>New Loading Data tutorial</td>
<td>Tutorial: Loading Data from Amazon S3 (p. 62) walks you through the process of loading data into your Amazon Redshift database tables from data files in an Amazon S3 bucket, from beginning to end.</td>
<td>July 1, 2014</td>
</tr>
<tr>
<td>PERCENTILE_CONT window function</td>
<td>PERCENTILE_CONT window function (p. 441) is an inverse distribution function that assumes a continuous distribution model. It takes a percentile value and a sort specification, and returns an interpolated value that would fall into the given percentile value with respect to the sort specification.</td>
<td>June 30, 2014</td>
</tr>
<tr>
<td>PERCENTILE_DISC window function</td>
<td>PERCENTILE_DISC window function (p. 442) is an inverse distribution function that assumes a discrete distribution model. It takes a percentile value and a sort specification and returns an element from the set.</td>
<td>June 30, 2014</td>
</tr>
<tr>
<td>GREATEST and LEAST functions</td>
<td>The GREATEST and LEAST (p. 468) functions return the largest or smallest value from a list of expressions.</td>
<td>June 30, 2014</td>
</tr>
<tr>
<td>Change</td>
<td>Description</td>
<td>Date Changed</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
<td>--------------</td>
</tr>
<tr>
<td>Cross-region COPY</td>
<td>The COPY (p. 276) command supports loading data from an Amazon S3 bucket or Amazon DynamoDB table that is located in a different region than the Amazon Redshift cluster. For more information, see REGION (p. 285) in the COPY command reference.</td>
<td>June 30, 2014</td>
</tr>
<tr>
<td>Best Practices expanded</td>
<td>Amazon Redshift Best Practices (p. 26) has been expanded, reorganized, and moved to the top of the navigation hierarchy to make it more discoverable.</td>
<td>May 28, 2014</td>
</tr>
<tr>
<td>COPY from Amazon EMR bootstrap action</td>
<td>The COPY (p. 276) command supports a streamlined process to load data directly from Amazon EMR clusters that are created using the Amazon Redshift bootstrap action. For more information, see Loading data from Amazon EMR using the Amazon Redshift bootstrap action (p. 128).</td>
<td>May 14, 2014</td>
</tr>
<tr>
<td>UNLOAD to a single file</td>
<td>The UNLOAD (p. 395) command can optionally unload table data serially to a single file on Amazon S3 by adding the PARALLEL OFF option. If the size of the data is greater than the maximum file size of 6.2 GB, UNLOAD creates additional files.</td>
<td>May 6, 2014</td>
</tr>
<tr>
<td>REGEXP functions</td>
<td>The REGEXP_COUNT (p.537), REGEXP_INSTR (p.538), and REGEXP_REPLACE (p. 539) functions manipulate strings based on regular expression pattern matching.</td>
<td>May 6, 2014</td>
</tr>
<tr>
<td>New Tutorial</td>
<td>The new Tutorial: Tuning Table Design (p. 35) walks you through the steps to optimize the design of your tables, including testing load and query performance before and after tuning.</td>
<td>May 2, 2014</td>
</tr>
<tr>
<td>COPY from Amazon EMR</td>
<td>The COPY (p. 276) command supports loading data directly from Amazon EMR clusters. For more information, see Loading data from Amazon EMR (p. 127).</td>
<td>April 18, 2014</td>
</tr>
<tr>
<td>WLM concurrency limit increase</td>
<td>You can now configure workload management (WLM) to run up to 50 queries concurrently in user-defined query queues. This increase gives users more flexibility for managing system performance by modifying WLM configurations. For more information, see Defining query queues (p. 197)</td>
<td>April 18, 2014</td>
</tr>
<tr>
<td>New configuration parameter to manage cursor size</td>
<td>The max_cursor_result_set_size configuration parameter defines the maximum size of data, in megabytes, that can be returned per cursor result set of a larger query. This parameter value also affects the number of concurrent cursors for the cluster, enabling you to configure a value that increases or decreases the number of cursors for your cluster. For more information, see DECLARE (p. 328) in this guide and Configure Maximum Size of a Cursor Result Set in the Amazon Redshift Cluster Management Guide.</td>
<td>Mar 28, 2014</td>
</tr>
<tr>
<td>Change</td>
<td>Description</td>
<td>Date Changed</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
<td>--------------</td>
</tr>
<tr>
<td>COPY from JSON format</td>
<td>The COPY (p. 276) command supports loading data in JSON format from data files on Amazon S3 and from remote hosts using SSH. For more information, see COPY from JSON format (p. 291) usage notes.</td>
<td>Mar 25, 2014</td>
</tr>
<tr>
<td>New system table STL_PLAN_INFO</td>
<td>The STL_PLAN_INFO (p. 616) table supplements the EXPLAIN command as another way to look at query plans.</td>
<td>Mar 25, 2014</td>
</tr>
<tr>
<td>New function REGEXP_SUBSTR</td>
<td>The REGEXP_SUBSTR function (p. 540) returns the characters extracted from a string by searching for a regular expression pattern.</td>
<td>Mar 25, 2014</td>
</tr>
<tr>
<td>New columns for STL_COMMIT_STATS</td>
<td>The STL_COMMIT_STATS (p. 590) table has two new columns: numxids and oldestxid.</td>
<td>Mar 6, 2014</td>
</tr>
<tr>
<td>COPY from SSH support for gzip and lzop</td>
<td>The COPY (p. 276) command supports gzip and lzop compression when loading data through an SSH connection.</td>
<td>Feb 13, 2014</td>
</tr>
<tr>
<td>New functions</td>
<td>The ROW_NUMBER window function (p. 444) returns the number of the current row. The STRTOL function (p. 547) converts a string expression of a number of the specified base to the equivalent integer value. PG_CANCEL_BACKEND (p. 566) and PG_TERMINATE_BACKEND (p. 566) enable users to cancel queries and session connections. The LAST_DAY (p. 487) function has been added for Oracle compatibility.</td>
<td>Feb 13, 2014</td>
</tr>
<tr>
<td>New system table STL_COMMIT_STATS</td>
<td>The STL_COMMIT_STATS (p. 590) system table provides metrics related to commit performance, including the timing of the various stages of commit and the number of blocks committed.</td>
<td>Feb 13, 2014</td>
</tr>
<tr>
<td>FETCH with single-node clusters</td>
<td>When using a cursor on a single-node cluster, the maximum number of rows that can be fetched using the FETCH (p. 344) command is 1000. FETCH FORWARD ALL is not supported for single-node clusters.</td>
<td>Feb 13, 2014</td>
</tr>
<tr>
<td>DS_DIST_ALL_INNER redistribution strategy</td>
<td>DS_DIST_ALL_INNER in the Explain plan output indicates that the entire inner table was redistributed to a single slice because the outer table uses DISTSTYLE ALL. For more information, see Join examples (p. 188) and Evaluating the query plan (p. 104).</td>
<td>Jan 13, 2014</td>
</tr>
<tr>
<td>New system tables for queries</td>
<td>Amazon Redshift has added new system tables that customers can use to evaluate query execution for tuning and troubleshooting. For more information, see SVL_COMPILE (p. 672), STL_SCAN (p. 629), STL_RETURN (p. 624), STL_SAVE (p. 625) STL_ALERT_EVENT_LOG (p. 587).</td>
<td>Jan 13, 2014</td>
</tr>
<tr>
<td>Single-node cursors</td>
<td>Cursors are now supported for single-node clusters. A single-node cluster can have two cursors open at a time, with a maximum result set of 32 GB. On a single-node cluster, we recommend setting the ODBC Cache Size parameter to 1,000. For more information, see DECLARE (p. 328).</td>
<td>Dec 13, 2013</td>
</tr>
<tr>
<td>Change</td>
<td>Description</td>
<td>Date Changed</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
<td>--------------</td>
</tr>
<tr>
<td>ALL distribution style</td>
<td>ALL distribution can dramatically shorten execution times for certain types of queries. When a table uses ALL distribution style, a copy of the table is distributed to every node. Because the table is effectively collocated with every other table, no redistribution is needed during query execution. ALL distribution is not appropriate for all tables because it increases storage requirements and load time. For more information, see Choosing a data distribution style (p. 101).</td>
<td>Nov 11, 2013</td>
</tr>
<tr>
<td>COPY from remote hosts</td>
<td>In addition to loading tables from data files on Amazon S3 and from Amazon DynamoDB tables, the COPY command can load text data from Amazon EMR clusters, Amazon EC2 instances, and other remote hosts by using SSH connections. Amazon Redshift uses multiple simultaneous SSH connections to read and load data in parallel. For more information, see Loading data from remote hosts (p. 134).</td>
<td>Nov 11, 2013</td>
</tr>
<tr>
<td>WLM Memory Percent Used</td>
<td>You can balance workload by designating a specific percentage of memory for each queue in your workload management (WLM) configuration. For more information, see Defining query queues (p. 197).</td>
<td>Nov 11, 2013</td>
</tr>
<tr>
<td>APPROXIMATE COUNT(DISTINCT)</td>
<td>Queries that use APPROXIMATE COUNT(DISTINCT) execute much faster, with a relative error of about 2%. The APPROXIMATE COUNT(DISTINCT) function uses a HyperLogLog algorithm. For more information, see the COUNT function (p. 417).</td>
<td>Nov 11, 2013</td>
</tr>
<tr>
<td>New SQL functions to retrieve recent query details</td>
<td>Four new SQL functions retrieve details about recent queries and COPY commands. The new functions make it easier to query the system log tables, and in many cases provide necessary details without needing to access the system tables. For more information, see PG_BACKEND_PID() (p. 573), PG_LAST_COPY_ID() (p. 574), PG_LAST_COPY_COUNT() (p. 574), PG_LAST_QUERY_ID() (p. 576).</td>
<td>Nov 1, 2013</td>
</tr>
<tr>
<td>MANIFEST option for UNLOAD</td>
<td>The MANIFEST option for the UNLOAD command complements the MANIFEST option for the COPY command. Using the MANIFEST option with UNLOAD automatically creates a manifest file that explicitly lists the data files that were created on Amazon S3 by the unload operation. You can then use the same manifest file with a COPY command to load the data. For more information, see Unloading data to Amazon S3 (p. 173) and UNLOAD examples (p. 399).</td>
<td>Nov 1, 2013</td>
</tr>
<tr>
<td>MANIFEST option for COPY</td>
<td>You can use the MANIFEST option with the COPY command to explicitly list the data files that will be loaded from Amazon S3.</td>
<td>Oct 18, 2013</td>
</tr>
<tr>
<td>Change</td>
<td>Description</td>
<td>Date Changed</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
<td>--------------</td>
</tr>
<tr>
<td>System tables for troubleshooting queries</td>
<td>Added documentation for system tables that are used to troubleshoot queries. The STL tables for logging (p. 584) section now contains documentation for the following system tables: STL_AGGR, STL_BCAST, STL_DIST, STL_DELETE, STL_HASH, STL_HASHJOIN, STL_INSERT, STL_LIMIT, STL_MERGE, STL_MERGEJOIN, STL_NESTLOOP, STL_PARSE, STL_PROJECT, STL_SCAN, STL_SORT, STL_UNIQUE, STL_WINDOW.</td>
<td>Oct 3, 2013</td>
</tr>
<tr>
<td>CONVERT_TIMEZONE function</td>
<td>The CONVERT_TIMEZONE function (p. 474) converts a timestamp from one time zone to another, with the option to automatically adjust for daylight savings time.</td>
<td>Oct 3, 2013</td>
</tr>
<tr>
<td>SPLIT_PART function</td>
<td>The SPLIT_PART function (p. 545) splits a string on the specified delimiter and returns the part at the specified position.</td>
<td>Oct 3, 2013</td>
</tr>
<tr>
<td>STL_USERLOG system table</td>
<td>STL_USERLOG (p. 639) records details for changes that occur when a database user is created, altered, or deleted.</td>
<td>Oct 3, 2013</td>
</tr>
<tr>
<td>LZO column encoding and LZOP file compression.</td>
<td>LZO (p. 94) column compression encoding combines a very high compression ratio with good performance. COPY from Amazon S3 supports loading from files compressed using LZOP (p. 285) compression.</td>
<td>Sept 19, 2013</td>
</tr>
<tr>
<td>JSON, Regular Expressions, and Cursors</td>
<td>Added support for parsing JSON strings, pattern matching using regular expressions, and using cursors to retrieve large data sets over an ODBC connection. For more information, see JSON Functions (p.553), Pattern-matching conditions (p. 244), and DECLARE (p. 328).</td>
<td>Sept 10, 2013</td>
</tr>
<tr>
<td>ACCEPTINVCHAR option for COPY</td>
<td>You can successfully load data that contains invalid UTF-8 characters by specifying the ACCEPTINVCHAR option with the COPY (p. 276) command.</td>
<td>Aug 29, 2013</td>
</tr>
<tr>
<td>CSV option for COPY</td>
<td>The COPY (p. 276) command now supports loading from CSV formatted input files.</td>
<td>Aug 9, 2013</td>
</tr>
<tr>
<td>CRC32</td>
<td>The CRC32 function (p. 526) performs cyclic redundancy checks.</td>
<td>Aug 9, 2013</td>
</tr>
<tr>
<td>WLM wildcards</td>
<td>Workload management (WLM) supports wildcards for adding user groups and query groups to queues. For more information, see Wildcards (p. 198).</td>
<td>Aug 1, 2013</td>
</tr>
<tr>
<td>WLM timeout</td>
<td>To limit the amount of time that queries in a given WLM queue are permitted to use, you can set the WLM timeout value for each queue. For more information, see WLM timeout (p. 199).</td>
<td>Aug 1, 2013</td>
</tr>
<tr>
<td>New COPY options 'auto' and 'epochsecs'</td>
<td>The COPY (p. 276) command performs automatic recognition of date and time formats. New time formats, 'epochsecs' and 'epochmilliseconds' enable COPY to load data in epoch format.</td>
<td>July 25, 2013</td>
</tr>
<tr>
<td>CONVERT_TIMEZONE function</td>
<td>The CONVERT_TIMEZONE function (p. 474) converts a timestamp from one time zone to another.</td>
<td>July 25, 2013</td>
</tr>
<tr>
<td>Change</td>
<td>Description</td>
<td>Date Changed</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
<td>--------------</td>
</tr>
<tr>
<td>FUNC_SHA1 function</td>
<td>The <strong>FUNC_SHA1</strong> function (p. 527) converts a string using the SHA1 algorithm.</td>
<td>July 15, 2013</td>
</tr>
<tr>
<td>max_execution_time</td>
<td>To limit the amount of time queries are permitted to use, you can set the <code>max_execution_time</code> parameter as part of the WLM configuration. For more information, see <strong>Modifying the WLM configuration</strong> (p. 201).</td>
<td>July 22, 2013</td>
</tr>
<tr>
<td>Four-byte UTF-8 characters</td>
<td>The VARCHAR data type now supports four-byte UTF-8 characters. Five-byte or longer UTF-8 characters are not supported. For more information, see <strong>Storage and ranges</strong> (p. 223).</td>
<td>July 18, 2013</td>
</tr>
<tr>
<td>SVL_QERROR</td>
<td>The <strong>SVL_QERROR</strong> system view has been deprecated.</td>
<td>July 12, 2013</td>
</tr>
<tr>
<td>Revised Document History</td>
<td>The Document History page now shows the date the documentation was updated.</td>
<td>July 12, 2013</td>
</tr>
<tr>
<td>STL_UNLOAD_LOG</td>
<td><strong>STL_UNLOAD_LOG</strong> (p. 638) records the details for an unload operation.</td>
<td>July 5, 2013</td>
</tr>
<tr>
<td>JDBC fetch size parameter</td>
<td>To avoid client-side out of memory errors when retrieving large data sets using JDBC, you can enable your client to fetch data in batches by setting the JDBC fetch size parameter. For more information, see <strong>Setting the JDBC fetch size parameter</strong> (p. 196).</td>
<td>June 27, 2013</td>
</tr>
<tr>
<td>UNLOAD encrypted files</td>
<td><strong>UNLOAD</strong> (p. 395) now supports unloading table data to encrypted files on Amazon S3.</td>
<td>May 22, 2013</td>
</tr>
<tr>
<td>Temporary credentials</td>
<td><strong>COPY</strong> (p. 276) and <strong>UNLOAD</strong> (p. 395) now support the use of temporary credentials.</td>
<td>April 11, 2013</td>
</tr>
<tr>
<td>Added clarifications</td>
<td>Clarified and expanded discussions of Designing Tables and Loading Data.</td>
<td>February 14, 2013</td>
</tr>
<tr>
<td>Added Best Practices</td>
<td>Added <strong>Best practices for designing tables</strong> (p. 26) and <strong>Best practices for loading data</strong> (p. 29).</td>
<td>February 14, 2013</td>
</tr>
<tr>
<td>Clarified password constraints</td>
<td>Clarified password constraints for <strong>CREATE USER</strong> and <strong>ALTER USER</strong>, various minor revisions.</td>
<td>February 14, 2013</td>
</tr>
<tr>
<td>New Guide</td>
<td>This is the first release of the <strong>Amazon Redshift Developer Guide</strong>.</td>
<td>February 14, 2013</td>
</tr>
</tbody>
</table>