Amazon Redshift

Database Developer Guide

API Version 2012-12-01

Amazon Web Services

Amazon Redshift: Database Developer Guide

Amazon Web Services

Welcome

Topics

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

  - CLI Reference
- Amazon Redshift Database Developer Guide *(this document)* – If you are a database developer, the Database Developer Guide explains how to design, build, query, and maintain the databases that make up your data warehouse.

If you are transitioning to Amazon Redshift from another relational database system or data warehouse application, you should be aware of important differences in how Amazon Redshift is implemented. For a summary of the most important considerations for designing tables and loading data, see Best practices for designing tables (p. 32) and Best practices for loading data (p. 54). Amazon Redshift is based on PostgreSQL 8.0.2. For a detailed list of the differences between Amazon Redshift and PostgreSQL, see Amazon Redshift and PostgreSQL (p. 111).

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.

If you want to ...	We recommend
Quickly start using Amazon Redshift	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.
Learn about the internal architecture of the Amazon Redshift data warehouse.	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.
Create databases, tables, users, and other database objects.	Getting Started Using Databases (p. 15) is a quick introduction to the basics of SQL development. The Amazon Redshift SQL (p. 110) has the syntax and examples for Amazon Redshift SQL commands and functions and other SQL elements. Best practices for designing tables (p. 32) provides a summary of our recommendations for choosing sort keys, distribution keys, and compression encodings.
Learn how to design tables for optimum performance.	Designing Tables (p. 32) details considerations for applying compression to the data in table columns and choosing distribution and sort keys.
Load data.	Loading Data (p. 54) 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. 54) provides for tips for loading your data quickly and effectively.
Manage users, groups, and database security.	Managing Database Security (p. 26) covers database security topics.

If you want to ...	We recommend
Monitor and optimize system performance.	The System Tables Reference (p. 450) 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.	Amazon Redshift is certified for use with Jaspersoft and MicroStrategy, with additional business intelligence tools coming soon. The SQL Reference (p. 110) 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 Getting Started Guide for step-by-step instructions.

You should also know how to use your SQL client and should have a fundamental understanding of the SQL language.

Amazon Redshift System Overview

Topics

An Amazon Redshift data warehouse is an enterprise-class relational database query and management system.

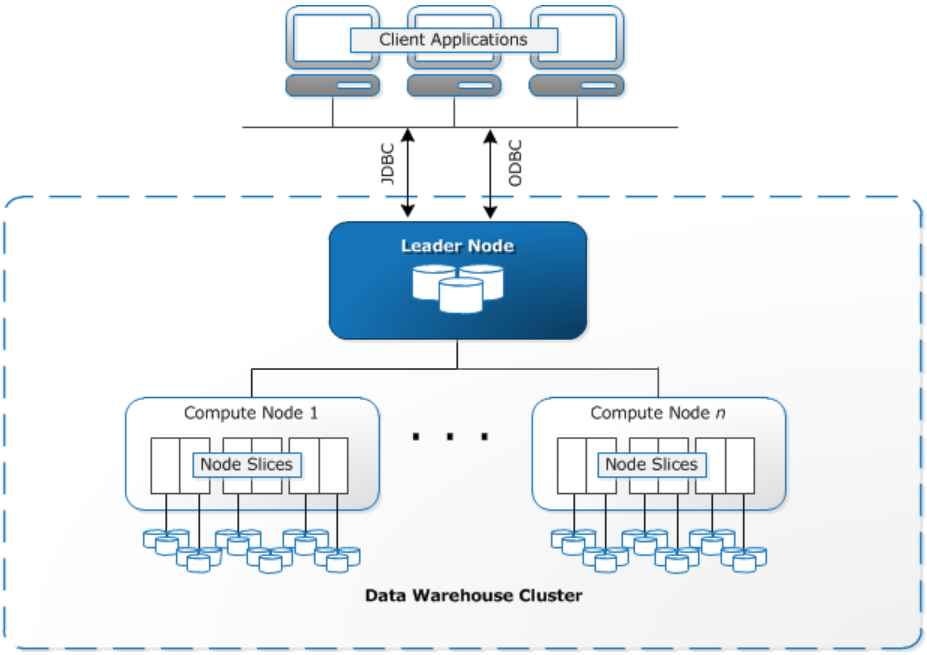
Amazon Redshift supports client connections with many types of applications, including business intelligence (BI), reporting, data, and analytics tools.

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.



Client Applications

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. 111).

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. 112).

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. 110).

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: an XL node with two cores, 15 GiB memory, and 3 disk drives with 2 TB of local attached storage, or an 8XL node with 16 cores, 120 GiB memory, and 24 disk drives with 16 TB of local attached storage. You can start with a single 2 TB XL node and scale up to multiple 16 TB 8XL 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 Choosing the best distribution key (p. 33).

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. 111).

Performance

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

- Massively parallel processing
- Columnar data storage
- Data compression
- Query optimization
- Compiled code

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 Choosing the best distribution key (p. 33).

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. 54).

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 Choosing the best sort key (p. 33).

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. 68).

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. 90).

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, 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.

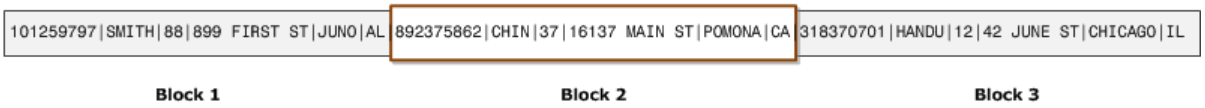
Columnar storage

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.

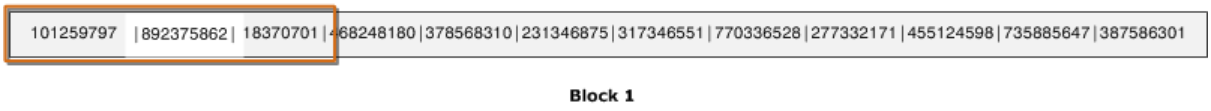
SSN	Name	Age	Addr	City	St
101259797	SMITH	88	899 FIRST ST	JUNO	AL
892375862	CHIN	37	16137 MAIN ST	POMONA	CA
318370701	HANDU	12	42 JUNE ST	CHICAGO	IL



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.

SSN	Name	Age	Addr	City	St
101259797	SMITH	88	899 FIRST ST	JUNO	AL
892375862	CHIN	37	16137 MAIN ST	POMONA	CA
318370701	HANDU	12	42 JUNE ST	CHICAGO	IL



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. 36).

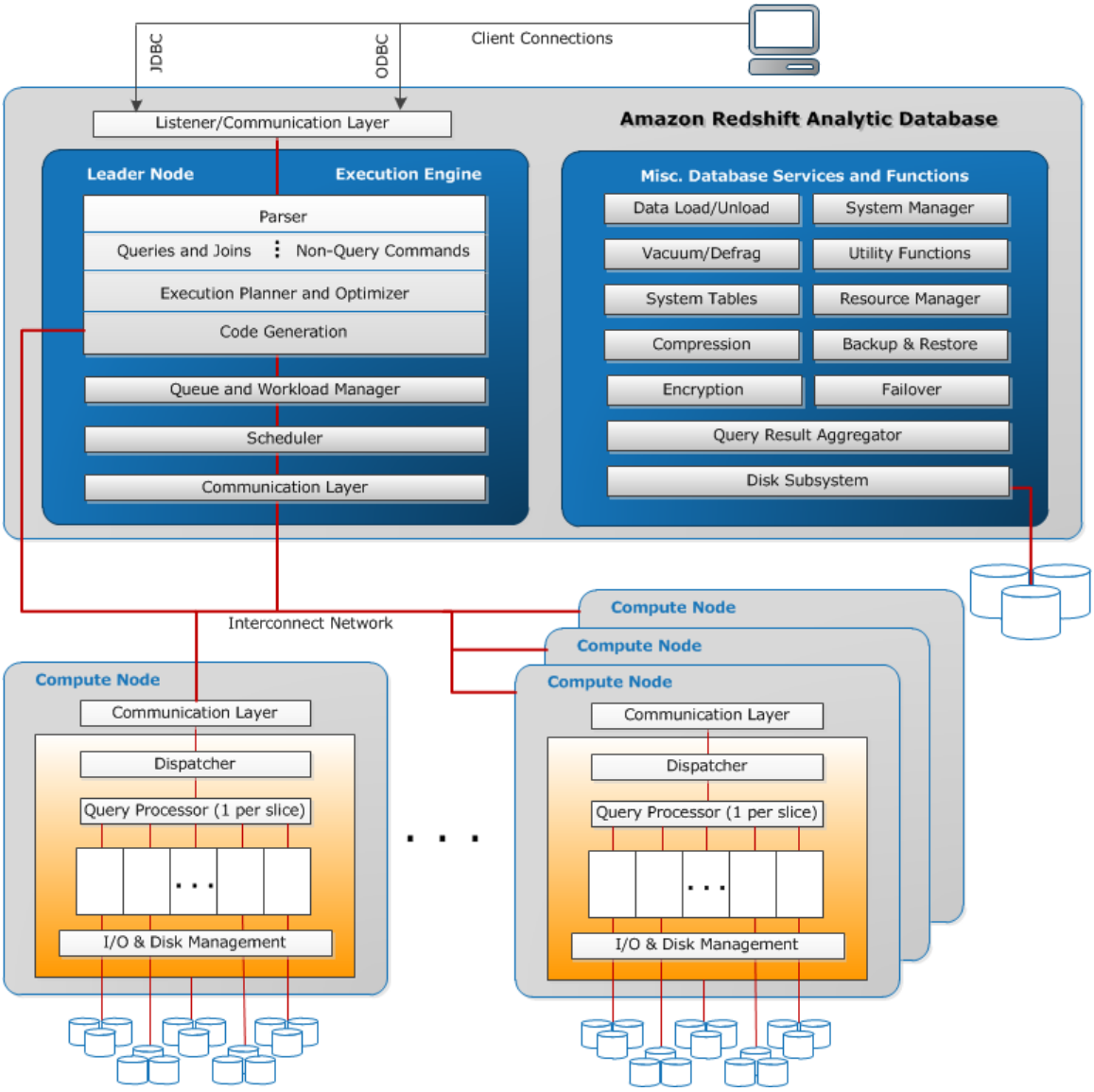
The savings in space for storing data on disk also carries over to retrieving and then storing that data in memory. Since many database operations only need to access or operate on one or a small number of columns at a time, you can save memory space by only retrieving blocks for columns you actually 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

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



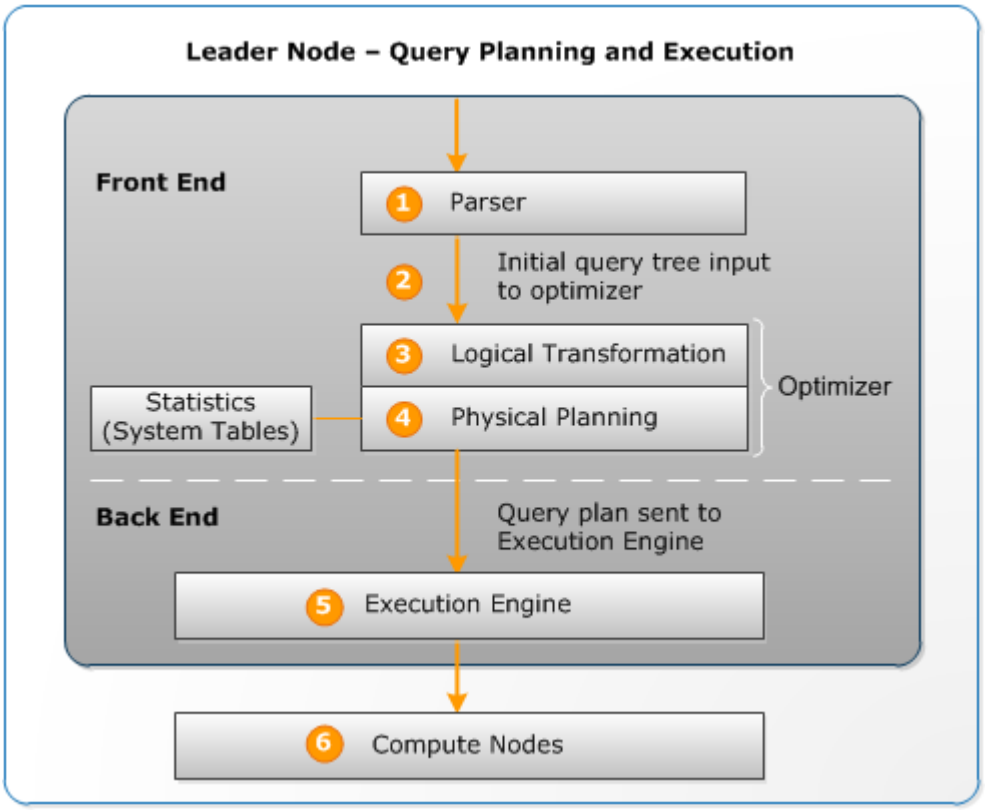
Query processing

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. The section provides an overview of Amazon Redshift query processing. For more information, see Tuning Query Performance (p. 90).

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. 227) 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. 101).
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.

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

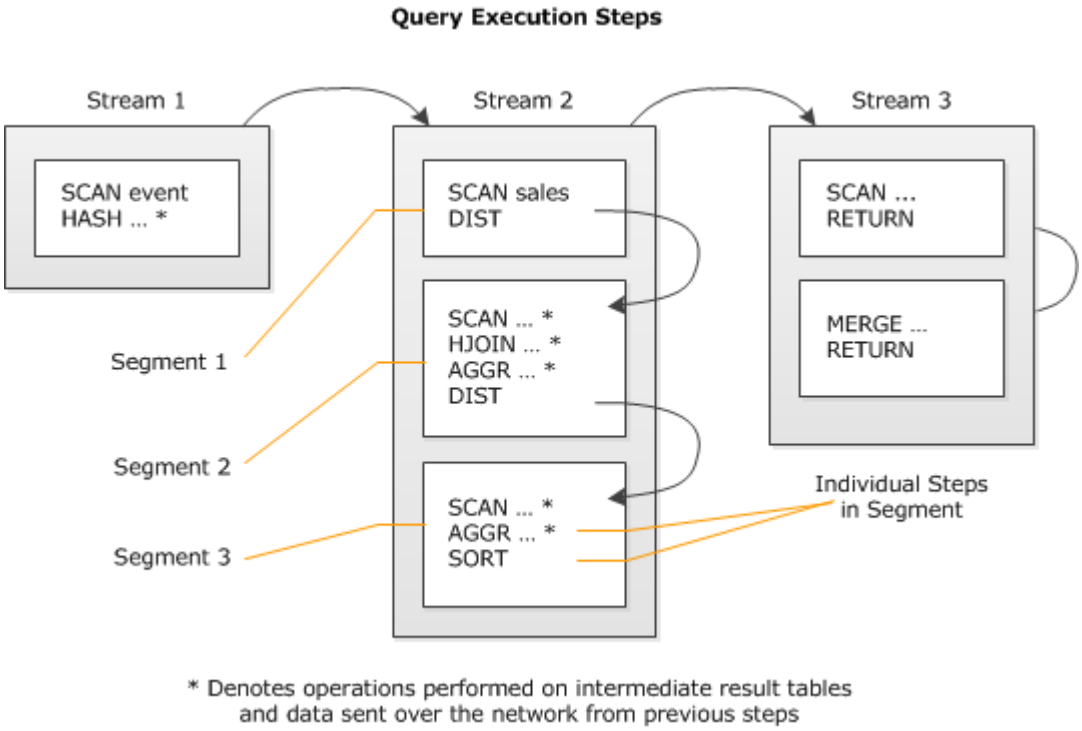
```
QUERY PLAN
XN Merge  (cost=1000451920505.33..1000451920506.77 rows=576 width=27)
  Merge Key: sum(sales.pricepaid)
  ->  XN Network  (cost=1000451920505.33..1000451920506.77 rows=576 width=27)
        Send to leader
       ->  XN Sort  (cost=1000451920505.33..1000451920506.77 rows=576 width=27)

              Sort Key: sum(sales.pricepaid)
              ->  XN HashAggregate  (cost=451920477.48..451920478.92 rows=576
width=27)
                    ->  XN Hash Join DS_DIST_INNER  (cost=47.08..451920458.65
rows=3766 width=27)
                          Inner Dist Key: sales.eventid
                          Hash Cond: ("outer".eventid = "inner".eventid)
                         ->  XN Seq Scan on event  (cost=0.00..87.98 rows=8798
 width=21)
                          ->  XN Hash  (cost=37.66..37.66 rows=3766 width=14)
                                ->  XN Seq Scan on sales  (cost=0.00..37.66
rows=3766 width=14)
```

For more information about how to read an explain plan, see Analyzing the explain plan (p. 91).

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 an 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.

The last segment of a query returns the data. If the return set is aggregated or sorted, the compute nodes each send their piece of the intermediate result to the leader node, which then merges the data so the final result can be sent back to the requesting client.

For more information about the operators used in the explain plan, see EXPLAIN operators (p. 93).

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 15. The maximum total concurrency level for all user-defined queues (not including the Superuser queue) is 15.

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. 102).

Getting Started Using Databases

Topics

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. When you are done, you should delete your cluster. 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. 198) 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 password 'ABCd4321';
```

For information about other command options, see CREATE USER (p. 213) 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. 224) 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:

```
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.

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

The query result should look something like this:

```
schemaname|tablename|column | type  |encoding|distkey|sortkey | notnull
----------+---------+-------+-------+--------+-------+--------+---------
 public   |testtable|testcol|integer|none    |f      |      0 | f
(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. 29) 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. 32).

Insert data rows into a table

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

> **Note**
> The INSERT (p. 235) command inserts individual rows into a database table. For standard bulk loads, use the COPY (p. 179) command. For more information, see Using a COPY command to load data (p. 55).

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

```
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:

```
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. 246) 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 *awssampledb*. 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. 60).

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. 32) 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.

```
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),
 venueseats integer);

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,
 numtickets 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 *awssampledb* 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/tickit/allusers_pipe.txt'
CREDENTIALS 'aws_access_key_id=<your-access-key-id>;aws_secret_ac
cess_key=<your-secret-access-key>'
delimiter '|';
copy venue from 's3://awssampledb/tickit/venue_pipe.txt'
CREDENTIALS 'aws_access_key_id=<your-access-key-id>;aws_secret_ac
cess_key=<your-secret-access-key>'
delimiter '|';
copy category from 's3://awssampledb/tickit/category_pipe.txt'
CREDENTIALS 'aws_access_key_id=<your-access-key-id>;aws_secret_ac
cess_key=<your-secret-access-key>'
delimiter '|';
copy date from 's3://awssampledb/tickit/date2008_pipe.txt'
CREDENTIALS 'aws_access_key_id=<your-access-key-id>;aws_secret_ac
cess_key=<your-secret-access-key>'
delimiter '|';
copy event from 's3://awssampledb/tickit/allevents_pipe.txt'
CREDENTIALS 'aws_access_key_id=<your-access-key-id>;aws_secret_ac
cess_key=<your-secret-access-key>'
delimiter '|' timeformat 'YYYY-MM-DD HH:MI:SS';
copy listing from 's3://awssampledb/tickit/listings_pipe.txt'
CREDENTIALS 'aws_access_key_id=<your-access-key-id>;aws_secret_ac
cess_key=<your-secret-access-key>'
delimiter '|';
copy sales from 's3://awssampledb/tickit/sales_tab.txt'
CREDENTIALS 'aws_access_key_id=<your-access-key-id>;aws_secret_ac
cess_key=<your-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.

```
select count(*) from users;
select count(*) from venue;
select count(*) from category;
select count(*) from date;
select count(*) from event;
select count(*) from listing;
select count(*) from sales;
```

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. 451)

  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. 482)

  These tables are virtual system tables that contain snapshots of the current system data. Snapshot tables have an STV prefix.
- System views (p. 504)

  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. 519)

  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.

```
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;
```

The result will look something like this:

```
 usename  | usesysid | usecreatedb | usesuper | usecatupd |  passwd  | valuntil
| useconfig
----------+----------+-------------+----------+-----------+----------+-------
```

```
---+----------
 masteruser|      100 | t           | t        | f         | ******** |
   |
 rdsdb     |        1 | t           | t        | t         | ******** |
   |
(2 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:

```
 query  |  pid  | elapsed  |                          substring
--------+-------+----------+-----------------------------------------------------
------------
 187752 | 18921 | 18465685 | select query, elapsed, substring from svl_qlog
order by query
 204168 |  5117 |    59603 | insert into testtable values (100);
 187561 | 17046 |  1003052 | select * from pg_table_def where tablename =
'testtable';
 187549 | 17046 |  1108584 | select * from STV_WLM_SERVICE_CLASS_CONFIG
 187468 | 17046 |  5670661 | select * from pg_table_def where schemaname =
'public';
(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 USERS 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:

```
  pid  |   btrim    |         starttime          |      substring
-------+------------+----------------------------+----------------------
 18764 | masteruser | 2013-03-28 18:39:49.355918 | select sellerid, fir
(1 row)
```

To cancel the query with PID 18764, issue the following command:

```
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:

```
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:

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

For information about managing query queues, see Implementing workload management (p. 102).

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:

```
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:

```
drop table testtable;
drop table users;
drop table venue;
drop table category;
drop table date;
drop table event;
drop table listing;
drop table sales;
```

Managing Database Security

Topics

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. 30).

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 Amazon S3 client-side encryption. When you load the data from Amazon S3, the COPY command will decrypt the data as it loads the table. For more information, see Uploading encrypted data to Amazon S3 (p. 62).
- **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. 233) 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. 243) 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. 450).

Superusers can query all PostgreSQL catalog tables. For information, see System catalog tables (p. 519).

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 ROLE 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 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. 213) command.
- To remove an existing user, use the DROP USER (p. 224) command.
- To make changes to a user account, such as changing a password, use the ALTER USER (p. 170) command.
- To view a list of users, query the PG_USER catalog table:

```
select * from pg_user;
```

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. 198) command.
- To add users to or remove users from an existing group, use the ALTER GROUP (p. 163) command.
- To delete a group, use the DROP GROUP (p. 221) 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. 30) 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. 199) command.
- To change the owner of a schema, use the ALTER SCHEMA (p. 164) command.
- To delete a schema and its objects, use the DROP SCHEMA (p. 221) 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. 276) command.

For more information, see the search_path (p. 527) 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. 243) 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:

```
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:

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

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

```
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:

```
create schema webapp;
```

   b. Grant USAGE privileges to the WEBAPPUSERS group:

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

   c. Grant USAGE privileges to the WEBPOWERUSERS group:

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

   d. Grant ALL privileges to the WEBDEVUSERS group:

```
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.

```
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;
```

Designing Tables

Topics

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.

Best practices for designing tables

Topics

As you plan your database, there are key decisions you must make 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.

When you create tables, these are the decisions that will have the greatest impact on query performance:

- Choose the best sort key
- Choose the best distribution key
- Choose the best compression strategy
- Define constraints

The choices you make will depend on the kind of work that your database is doing. For example, there is no one best sort key for all situations.

This section summarizes the most important design decisions and presents best practices for optimizing query performance. Later sections provide more detailed explanations and examples of table design options.

Choosing 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 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 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 Choosing sort keys (p. 50).

Choosing the best distribution key

The distribution key determines how a table's data is distributed across compute nodes.

Good data distribution has two goals:

- To distribute data evenly among the nodes and slices in a cluster.

  Uneven distribution, or data distribution skew, forces some nodes to do more work than others, slowing the whole process down.
- To collocate data for joins to minimize data movement.

If you frequently join a table, specify the join column as the distribution key. If a table joins with multiple other tables, distribute on the foreign key of the largest dimension that the table joins with. If the dimension tables are filtered as part of the joins, compare the size of the data after filtering when you choose the largest dimension. This ensures that the rows involved with your largest joins will generally be distributed to the same physical nodes. Because local joins avoid data movement, they will perform better than network joins.

If you have primarily equality filters on a column, do not choose that column for the distribution key because, effectively, all of your matching data will be stored in a single node. If that column is also your sort key, then the situation is even worse, because all the processing will be concentrated in a single slice within that node.

If your tables are largely denormalized and do not participate in joins, do not specify a distribution key. Amazon Redshift will evenly distribute the data across nodes in a round robin fashion.

See Choosing a data distribution method (p. 45) for more information about choosing and specifying distribution keys.

Defining 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.

Amazon Redshift does not enforce unique, primary-key, and foreign-key constraints. Your application is responsible for ensuring uniqueness and managing the idempotency of DML operations.

See Defining constraints (p. 51) for additional information about how Amazon Redshift uses constraints.

Using the smallest possible column size

You will improve query performance by reducing columns to the minimum possible size. 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. 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.

Using 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. 131).

Specifying 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';
```

Choosing a column compression type

Topics

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. When you create tables in an Amazon Redshift database, you can define a compression type, or *encoding*, for any or all of the columns. Compression reduces the size of data that is read from disk and consequently reduces the amount of disk I/O during processing.

You can apply compression encodings to columns in tables manually when you create the tables, 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. 68).

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. 172) 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:

```
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).

```
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.

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

Compression encodings

Topics

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.

Encoding type	Keyword in CREATE TABLE and ALTER TABLE	Data types
Raw (no compression)	RAW	All
Byte dictionary	BYTEDICT	All except BOOLEAN
Delta	DELTA	SMALLINT, INT, BIGINT, DATE, TIMESTAMP, DECIMAL
	DELTA32K	INT, BIGINT, DATE, TIMESTAMP, DECIMAL
Mostly*n*	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

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

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 the column holds long character strings.

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:

Unique data value	Dictionary index	Size (fixed length, 30 bytes per value)
England	0	30
United States of America	1	30
Venezuela	2	30
Sri Lanka	3	30
Argentina	4	30
Japan	5	30
Total		180

The following table represents the values in the COUNTRY column:

Original data value	Original size (fixed length, 30 bytes per value)	Compressed value (index)	New size (bytes)
England	30	0	1
England	30	0	1
United States of America	30	1	1
United States of America	30	1	1
Venezuela	30	2	1
Sri Lanka	30	3	1

Original data value	Original size (fixed length, 30 bytes per value)	Compressed value (index)	New size (bytes)
Argentina	30	4	1
Japan	30	5	1
Sri Lanka	30	3	1
Argentina	30	2	1
Totals	300		10

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:

Original data value	Original size (bytes)	Difference (delta)	Compressed value	Compressed size (bytes)
1	4		1	1+4 (flag + actual value)
5	4	4	4	1
50	4	45	45	1
200	4	150	150	1+4 (flag + actual value)
185	4	-15	-15	1
220	4	35	35	1

Original data value	Original size (bytes)	Difference (delta)	Compressed value	Compressed size (bytes)
221	4	1	1	1
Totals	28			15

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.

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

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 123,456 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:

Original value	Original INT or BIGINT size (bytes)	MOSTLY8 compressed size (bytes)	MOSTLY16 compressed size (bytes)	MOSTLY32 compressed size (bytes)
1	4	1	2	4
10	4	1	2	4
100	4	1	2	4
1000	4	Same as raw data size	2	4
10000	4		2	4
20000	4		2	4
40000	8		Same as raw data size	4
100000	8			4
2000000000	8			4

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:

Original data value	Original size (bytes)	Compressed value (token)	Compressed size (bytes)
Blue	4	{2,Blue}	5
Blue	4		0
Green	5	{3,Green}	6
Green	5		0
Green	5		0
Blue	4	{1,Blue}	5

Original data value	Original size (bytes)	Compressed value (token)	Compressed size (bytes)
Yellow	6	{4,Yellow}	7
Yellow	6		0
Yellow	6		0
Yellow	6		0
Totals	51		23

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 stores 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. 172) command to view the suggested encodings for existing data. For details about applying automatic compression, see Loading tables with automatic compression (p. 68).

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, we create a table with four columns, each with a different compression encoding: raw, runlength, text32k, and text255. 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.

```
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.

```
create table encodingvenue (
venueraw varchar(100) encode raw,
venuerunlength varchar(100) encode runlength,
venuetext32k varchar(100) encode text32k,
venuetext255 varchar(100) encode text255);
```

3. Insert the same data into all of the columns using an INSERT statement with a SELECT clause.

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

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

```
select count(*) from encodingvenue

  count
----------
 38884394
(1 row)
```

5. Query the STV_BLOCKLIST (p. 483) 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 < 5` in the WHERE clause to exclude the system-generated columns.

```
select col, max(blocknum)
from stv_blocklist b, stv_tbl_perm p
where (b.tbl=p.id) and name ='encodingvenue'
and col < 5
group by name, col
order by col;
```

   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 TEXT255 encoding on the fourth column produced the best results for this data set.

```
 col | max
-----+-----
```

```
   0 | 203
   1 | 204
   2 |  72
   3 |  55
(4 rows)
```

If you have data in an existing table, you can use the ANALYZE COMPRESSION (p. 172) 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 that contains 32 million rows. Notice that ANALYZE COMPRESSION recommends TEXT255 encoding for the VENUENAME column, which agrees with the results of our previous test.

```
analyze compression reallybigvenue;

     Column    | Encoding
---------------+-----------
  venueid      | bytedict
  venuename    | text255
  venuecity    | text255
  venuestate   | bytedict
  venueseats   | bytedict
 (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 runlength,
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:

Column	Data Type	Encoding	Explanation
CUSTKEY	int	delta	CUSTKEY consists of unique, consecutive integer values. Since the differences will be one byte, DELTA is a good choice.

Column	Data Type	Encoding	Explanation
CUSTNAME	varchar(30)	raw	CUSTNAME has a large domain with few repeated values. Any compression encoding would probably be ineffective.
GENDER	varchar(7)	text255	GENDER is very small domain with many repeated values. Text255 works well with VARCHAR columns in which the same words recur.
ADDRESS	varchar(200)	text255	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.
CITY	varchar(30)	text255	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.
STATE	char(2)	raw	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.

Column	Data Type	Encoding	Explanation
ZIPCODE	char(5)	bytedict	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.
START_DATE	date	delta32k	Delta encodings are very useful for datetime columns, especially if the rows are loaded in date order.

Choosing a data distribution method

Topics

The query performance that Amazon Redshift derives from massively parallel processing (MPP) depends on an even distribution of data across all of the nodes and slices in the cluster. When data is evenly distributed, the leader node is better able to evenly distribute the processing load among the slices.

This section discuss the role of nodes and slices in parallel processing and describes how to select and implement the distribution methods best suited to leveraging parallel processing in your database.

Nodes and slices

Think of a cluster as a parallel machine. An Amazon Redshift cluster is made up of a set of nodes. Each node in the cluster has its own operating system, memory, and directly attached storage (disk). These resources are not shared. One node is the *leader node*, which manages the distribution of data and query processing tasks to the *compute nodes*. Each compute node is further divided into a number of *slices*.

The number of slices is equal to the number of processor cores on the node. For example, each XL compute node has two slices, and each 8XL compute node has 16 slices. The slices all participate in parallel query execution, working on data that is distributed as evenly as possible across the nodes. The Data warehouse system architecture (p. 4) section includes an illustration that shows the relationship of the leader node, compute nodes, and slices in an Amazon Redshift cluster.

Distribution methods

When you add data to a table, Amazon Redshift distributes the rows in the table to the cluster slices using one of two methods:

- Even distribution
- Key distribution

Even distribution

Even distribution is the default distribution method. With even distribution, the leader node spreads data rows across the slices in a round-robin fashion, regardless of the values that exist in any particular column. This approach is a good choice when you do not have a clear option for a distribution key.

If your tables are largely denormalized and do not participate in joins, use even distribution.

To specify even distribution, use the DISTSTYLE EVEN option with the CREATE TABLE statement, as the following example shows:

```
create table newusers diststyle even as
select * from users;
```

Key distribution

If you specify a distribution key when you create a table, the leader node distributes the data rows to the slices based on the values in the distribution key column. Matching values from the distribution key column are stored together.

Evaluate the data values in your table and consider how your queries will select rows when you choose the distribution key.

Select a distribution key with these goals in mind:

- To distribute data evenly among the slices in a cluster. Uneven distribution, or data distribution skew, forces some slices to do more work than others, slowing the whole process down.
- To collocate data for joins to minimize data movement.

When all the data for rows that participate in a join are located on the same node, the query engine doesn't need to move as much data across the network.

> **Important**
> Once you have specified a distribution method 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.

Choosing distribution keys

In order to make good choices for distribution and sort keys, you need to understand the query patterns for your Amazon Redshift application.

Distributing on join columns

If the primary key of a table is not frequently used as a joining column, consider not making the primary key the distribution key. Instead, if the table has a foreign key or another column that is frequently used as a join key, consider making that column the distribution key. In making this choice, take pairs of joined tables into account. You might get better results when you specify the joining columns as the distribution keys and the sort keys on both tables. This enables the query optimizer to select a faster merge join instead of a hash join when executing the query.

Aggregation and grouping operations might also benefit from specifying the same column as both the distribution key and the sort key. For example, sort-based aggregation on a column that is both the distribution key and the sort key is likely to be much faster than hash-based aggregation. For information about sort keys, see Choosing the best sort key (p. 33).

For example, consider the schema of the SALES table in the TICKIT database, which has a single-column primary key called SALESID. Although SALESID would yield even distribution, it is not the best choice for the distribution key if you perform frequent JOIN operations. SALESID might be used to restrict rows selected from SALES, but it is never used as a join column to other tables. The SALES table is often joined to other tables in the database, especially LISTING, on the LISTID column. Therefore LISTID is the best distribution and sort key for both the SALES and LISTING tables.

This example highlights the choice of distribution *and* sort keys in order to optimize queries with collocated joins. Collocated joins reduce data movement during query processing because the data required for the join operation is already distributed appropriately when the query is submitted. However, join cost is not the only factor to consider when you choose distribution and sort keys for a table.

Viewing table details

You can use the PG_TABLE_DEF catalog table to view the column details for a table. The following example shows the encoding for the SALES table and the LISTING table. Note that "column" in the SELECT list is a reserved word, so it must be enclosed in double quotes.

```
select "column", type, encoding, distkey, sortkey
from pg_table_def where tablename = 'sales';

   column   |         type         | encoding | distkey | sortkey
------------+----------------------+----------+---------+---------
 salesid    | integer              | none     | f       |       0
 listid     | integer              | none     | t       |       1
 sellerid   | integer              | none     | f       |       2
 buyerid    | integer              | none     | f       |       0
 eventid    | integer              | mostly16 | f       |       0
 dateid     | smallint             | none     | f       |       0
 qtysold    | smallint             | mostly8  | f       |       0
 pricepaid  | numeric(8,2)         | delta32k | f       |       0
 commission | numeric(8,2)         | delta32k | f       |       0
 saletime   | timestamp without... | none     | f       |       0
(10 rows)


select "column", type, encoding, distkey, sortkey
from pg_table_def where tablename = 'listing';

     column     |         type         | encoding | distkey | sortkey
----------------+----------------------+----------+---------+---------
 listid         | integer              | none     | t       |       1
 sellerid       | integer              | none     | f       |       0
 eventid        | integer              | mostly16 | f       |       0
 dateid         | smallint             | none     | f       |       0
 numtickets     | smallint             | mostly8  | f       |       0
 priceperticket | numeric(8,2)         | bytedict | f       |       0
 totalprice     | numeric(8,2)         | mostly32 | f       |       0
 listtime       | timestamp without... | none     | f       |       0
(8 rows)
```

Filtering requirements

If most of your queries are restricted by a WHERE clause that specifies a particular column, you should consider specifying that column as the sort key. For example, your application might run a large number of queries that restrict dates, customer names, or cities. If you define such a column as the sort key, a

different kind of optimization comes into play. If the data is presorted for a column whose value restricts the scope of the query, the process of scanning rows during query processing is faster, especially when the restriction results in a small range of values from a very large domain (for example, a date range of 30 days in a table that spans five years of data). If your application runs this kind of query routinely, the scan optimization might be more important than the collocated join optimization.

For example, the CALDATE column in the DATE table is a good alternative sort key for queries that routinely contain the following kind of range restriction:

```
where caldate between '2008-01-01' and '2008-02-28'
```

See Choosing the best distribution key (p. 33) for guidelines for selecting a good distribution key.

Tables without distribution keys

You should declare a distribution key for most of the tables you create. In some cases, you might choose even distribution instead of key distribution if the table offers no clear choice for a specific distribution column. Another case where you might not declare a distribution key is when you create new tables based on data in existing tables. When you write a CREATE TABLE AS statement, for example, you might be able to rely on the incoming data to set the distribution behavior . For information, see the description and examples for CREATE TABLE AS (p. 209).

Tables without sort keys

If you do not specify a SORTKEY column in a CREATE TABLE or CREATE TABLE AS statement, the table is not sorted. A table might not need a SORTKEY when you know that the data that you initially and incrementally load is pre-sorted. For example, you might have a table that is updated daily or weekly. The only rows that are ever added are rows with new dates or timestamps, so that the table is naturally sorted by time. Tables without sort keys can still benefit from query optimizations such as range-restricted scans over date ranges. For information, see Choosing sort keys (p. 50).

One advantage to creating such a table is that you avoid the need for vacuum operations on the table to resort rows after incremental loads. You can run a faster VACUUM DELETE ONLY operation on the table after any DELETE or UPDATE operations, instead of having to run a full vacuum or a SORT ONLY vacuum. For more information, see Vacuuming tables (p. 78) and the command description for VACUUM (p. 294).

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, which has its primary-key column, USERID, defined as the DISTKEY column:

```
select "column", type, encoding, distkey, sortkey
from pg_table_def where tablename = 'users';

    column     |          type          | encoding | distkey | sortkey
---------------+------------------------+----------+---------+---------
 userid        | integer                | none     | t       |       1
 username      | character(8)           | none     | f       |       0
 firstname     | character varying(30)  | text32k  | f       |       0
```

```
...
```

USERID is a good choice for the distribution column on this table. If you query the SVV_DISKUSAGE system view (which requires that you be logged in as a superuser), you can see that the table is very evenly distributed:

```
select slice, col, num_values, minvalue, maxvalue
from svv_diskusage
where name='users' and col =0
order by slice, col;

slice| col | num_values | minvalue | maxvalue
-----+-----+------------+----------+----------
0    | 0   | 12496      | 4        | 49987
1    | 0   | 12498      | 1        | 49988
2    | 0   | 12497      | 2        | 49989
3    | 0   | 12499      | 3        | 49990
(4 rows)
```

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 2 (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 newusers 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;

slice | col | num_values | minvalue | maxvalue
------+-----+------------+----------+----------
    0 |   0 |      13587 |        5 |    49989
    1 |   0 |      11245 |        2 |    49990
    2 |   0 |      15008 |        1 |    49976
    3 |   0 |      10150 |        4 |    49986
(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 newusers diststyle even as
select * from users;

select slice, col, num_values, minvalue, maxvalue from svv_diskusage
```

```
where name = 'newusers' and col=0 order by slice, col;

slice | col | num_values | minvalue | maxvalue
------+-----+------------+----------+----------
    0 |   0 |      12497 |        4 |    49990
    1 |   0 |      12498 |        8 |    49984
    2 |   0 |      12498 |        2 |    49988
    3 |   0 |      12497 |        1 |    49989
(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.

Data redistribution

You cannot manually redistribute the data in a table after it has been loaded; however, when query plans are generated, they often contain data broadcast or redistribution directives. The optimizer determines where slices or columns of data need to be located to best execute a given query plan, and the data is physically moved during execution.This kind of data movement is a routine part of query processing in a multi-node system.

Redistribution involves either broadcasting an entire table to all of the nodes or sending specific rows to nodes for joining purposes.

When a table being joined is not distributed on the joining column specified in the query, slices have to be redistributed or broadcast to all the compute nodes. When the tables being joined are distributed on the joining columns, *collocated joins* occur: the slices of each table are joined separately on each node.

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. 75).

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 Choosing distribution keys (p. 46).

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 (p. 227) command. For more information, see Analyzing the explain plan (p. 91)

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. 78).

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. 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.

```
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)
                   THEN 1 ELSE t2.min_blocks_per_slice END,
      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.

```
          |         |       |     |has_  |has_ |has_    |pct_skew_|pct_
          |         |       |size_|dist_ |sort_|col_    |across_  |slices_
schemaname|tablename|tableid|in_mb|key   |key  |encoding|slices   |populated
----------+---------+-------+-----+------+-----+--------+---------+---------
public    |category |100553 |  28 |    1 |   1 |      0 |       0 |      100
public    |date     |100555 |  44 |    1 |   1 |      0 |       0 |      100
public    |event    |100558 |  36 |    1 |   1 |      1 |       0 |      100
public    |listing  |100560 |  44 |    1 |   1 |      1 |       0 |      100
public    |nation   |100563 | 175 |    0 |   0 |      0 |       0 |    39.06
public    |region   |100566 |  30 |    0 |   0 |      0 |       0 |     7.81
public    |sales    |100562 |  52 |    1 |   1 |      0 |       0 |      100
public    |skew     |100547 |18978|    0 |   0 |      0 |   15.12 |       50
public    |skew2    |100548 | 353 |    1 |   0 |      0 |       0 |     1.56
public    |venue    |100551 |  32 |    1 |   1 |      0 |       0 |      100
public    |users    |100549 |  82 |    1 |   1 |      1 |       0 |      100
public    |venue    |100551 |  32 |    1 |   1 |      0 |       0 |      100
```

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

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.

Best practices for loading data

Topics

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.

There is no one best method of loading data for all situations. Later sections will provide more detailed explanations and examples of table design options.

Using a COPY command to load data

Use a COPY command to load data in parallel from Amazon S3 or Amazon DynamoDB. 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. 60) and Loading data from an Amazon DynamoDB table (p. 66).

When you run a COPY command to load data into an empty table that uses a sort key, Amazon Redshift sorts your data before storing it on disk, which improves query performance. See Choosing the best sort key (p. 33).

Using a single COPY command

Use a single COPY command to load data for one table from multiple files. Amazon Redshift then automatically loads the data in parallel.

Do not use multiple concurrent COPY commands to load one table from multiple files. Using multiple COPY commands results in 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. 60).

Splitting your data into multiple files

When you use a COPY command to load data from Amazon S3, first split your data into multiple files instead of loading all the data from a single large file.

The COPY command then loads the data in parallel from multiple files, dividing the workload among the nodes in your cluster. 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. 60).

Compressing your data files with GZIP

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

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

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

Using 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.

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

See INSERT (p. 235) for more details and examples.

Using a bulk insert

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

Use the INSERT (p. 235) and CREATE TABLE AS (p. 209) 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.

```
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.

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

Loading data in sort key order

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

Your data will be properly stored in sort order, so a VACUUM might not be needed. 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 effectively stored in sort order because the current day's data is always appended at the end of the previous day's data.

Using 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.

Using a staging table to perform an upsert

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 upsert (update or insert) command to insert and update data from a single data source, you can effectively perform an upsert 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. 74).

Vacuuming 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.

When you run a COPY command to load data into a table that uses a sort key, Amazon Redshift sorts your data before storing it on disk. (For more information, see Loading data in sort key order (p. 57).) If you later add a significant number of new rows to the table, your performance might degrade because new rows are not sorted relative to the existing rows. For more information about how often to vacuum, see Vacuuming tables (p. 78).

Performing a deep copy to avoid long vacuums

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 tradeoff is that you cannot make concurrent updates during a deep copy operation, which you can do during a vacuum.

Follow these steps to perform a deep copy:

1. Create a new table with the same attributes as the current table.

   Use a copy of the CREATE TABLE DDL for the current table to create the new table.

   Alternatively, you can also use a CREATE TABLE … LIKE statement to create the new table, but the new table does not inherit primary key and foreign key constraints, if any exist.

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 with an INSERT INTO … SELECT statement. If the original CREATE TABLE DDL is available, use that instead and skip the first statement.

```
create table likesales (like sales);
insert into likesales (select * from sales);
drop table sales;
alter table likesales rename to sales;
```

If you use CREATE TABLE … LIKE, the new table does not inherit the primary key and foreign key attributes of the original table. If the original CREATE TABLE DDL is not available, and you need to preserve primary key and foreign key attributes, you can preserves those attributes by keeping the original table intact and using TRUNCATE instead of DROP TABLE. The drawback is that the operation takes longer because it uses two inserts.

```
create temp table tempsales as select * from sales;
truncate sales;
insert into sales (select * from tempsales);
drop table tempsales;
```

Increasing 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. 529) for instructions about how to set this parameter and more details about the concurrency implications.

Maintaining 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. 75) for more information and examples.

Using a COPY command to load data

Topics

The COPY command leverages the Amazon Redshift massively parallel processing (MPP) architecture to read and load data in parallel from files on Amazon S3 or from a DynamoDB table.

> **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. 235) or CREATE TABLE AS (p. 209).

To grant or revoke privilege to load data into a table using a COPY command, grant or revoke 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 compress the files using GZIP to save time uploading the files. COPY is then able to speed up the load process by decrypting the files as they are read. 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.

To 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. 179) 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. 188). 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. 119). For information about multi-byte character errors, see Multi-byte character load errors (p. 72)

Loading data from Amazon S3

Topics

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 parallelism 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 method (p. 45).

The 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 (|). All the table columns are loaded by default, or 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. 70).

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

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. A COPY command can load data from multiple files in parallel if the file names share a common prefix, or *prefix key*, for the set.

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. For example, each XL compute node has two slices, and each 8XL compute node has 16 slices. If you have a cluster with two XL nodes, you might split your data into four files. Amazon Redshift does not take file size into account when dividing the workload, so make sure the files are roughly the same size.

Name each file with a common prefix. For example, the `venue.txt` file might be split into four files, as follows:

```
venue.txt.1
venue.txt.2
venue.txt.3
venue.txt.4
```

If you put multiple files in a folder in your bucket, you can specify the bucket name as the prefix and COPY will load all of the files in the bucket.

Uploading files to Amazon S3

After splitting your files, you can upload them to your bucket. You can optionally encrypt the files and compress the files before you load them.

Create an Amazon S3 bucket to hold your data files, 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.*

Important
The Amazon S3 bucket that holds the data files must be created in the same region as your cluster.

If you created your cluster in the US East (N. Virginia) Region, do not specify the US Standard Region when you create your bucket. Amazon S3 automatically routes requests that specify the US Standard Region to facilities in Northern Virginia or the Pacific Northwest, so if you specify the US Standard Region, it is possible that your data will not be located near your cluster. To minimize data latency, use a named endpoint when you create your Amazon S3 bucket.

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
```

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

You can upload data to an Amazon S3 bucket using client-side encryption, then securely load the data using the COPY command with the ENCRYPTED option and a private encryption key.

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 secure key management that asymmetric keys provide. 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.

These are the steps for uploading encrypted files to Amazon S3

1. Generate a one time use envelope symmetric key.
2. Encrypt the file data using this envelope key.
3. Encrypt that envelope key using a master public key or symmetric key.
4. Store this encrypted envelope key with the encrypted file.
5. Store a description of the master key alongside the envelope key to uniquely identify the key used to encrypt the envelope key.

When you configure your client encryption, follow these guidelines

- Use Symmetric Encryption mechanism
- Provide a 256 bit AES symmetric key to the AWS S3 encryption client
- Use Object Metadata storage mode

For more information about uploading encrypted files to Amazon S3, see Using Client-Side Encryption.

See Loading encrypted data files from Amazon S3 (p. 65) for steps to load encrypted files into your Amazon Redshift tables using the COPY command.

Using a COPY command to load from Amazon S3

Topics

Use the COPY command to load a table in parallel from data files on Amazon S3.

The syntax is as follows:

```
copy <table_name> from 's3://<bucket_name>/<object_prefix>'
credentials 'aws_access_key_id=<my aws_access key id>;
aws_secret_access_key=<my aws secret access key>' [<options>];
```

Note
These examples contain line breaks for readability. 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. 200) in the SQL Reference. The values for *<my aws access key id>* and *<my aws secret access key>* are the AWS credentials needed to access the Amazon S3 objects. If these credentials correspond to an IAM user, that IAM user must have 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 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. 188). To load data using temporary access credentials, use the following syntax:

```
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>'
[<options>];
```

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. 179) 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. 60).

For example, the `venue.txt` file might be split into four files, as follows:

```
venue.txt.1
venue.txt.2
venue.txt.3
venue.txt.4
```

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).

```
copy venue from 's3://mybucket/venue'
credentials 'aws_access_key_id=<your-access-key-id>;
aws_secret_access_key=<your-secret-access-key>'
delimiter '|';
```

If no Amazon S3 objects with the key prefix 'venue' exist, the load fails.

Loading GZIP compressed data files from Amazon S3

To load data files that are compressed using GZIP, include the `gzip` option.

```
copy customer from 's3://mybucket/customer.gz'
credentials 'aws_access_key_id=<your-access-key-id>;aws_secret_access_key=<your-
secret-access-key>'
delimiter '|' gzip;
```

If you use the gzip option, ensure that all data files being loaded are GZIP compressed.

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:

```
copy table_name from 's3://mybucket/prefix'
credentials 'aws_access_key_id=<your-access-key-id>;aws_secret_access_key=<your-
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:

```
1  Toyota Park              Bridgeview  IL0
2  Columbus Crew Stadium    Columbus    OH0
3  RFK Stadium              Washington  DC0
4  CommunityAmerica BallparkKansas City KS0
5  Gillette Stadium         Foxborough  MA68756
```

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=<your-access-key-id>;aws_secret_access_key=<your-
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. 119).

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. 62).

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=<Your-Access-Key-ID>;
aws_secret_access_key=<your-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=<Your-Access-Key-ID>;
aws_secret_access_key=<your-secret-access-key>;
master_symmetric_key=<master_key>' delimiter '|' encrypted gzip;
```

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.

The COPY command leverages the Amazon Redshift massively parallel processing (MPP) architecture to read and load data in parallel from Amazon DynamoDB tables. You can take maximum advantage of parallelism by setting distribution keys on your Amazon Redshift tables. For more information, see Choosing a data distribution method (p. 45).

> **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. 179) 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=<your-access-key-id>;aws_secret_access_key=<your-
secret-access-key>'
readratio '<integer>' [options ];
```

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. 188). 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_ac
cess_key=<temporary-secret-access-key>;token=<temporary-token>'
[<options>];
```

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. 179) 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=<your-access-key-id>;aws_secret_access_key=<your-
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. 68).

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. 119).

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. 179) 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
context:   S3 key being read :
location:  step_scan.cpp:1883
process:   xenmaster [pid=22199]
-----------------------------------------------
```

To troubleshoot error messages, see the Load error reference (p. 73).

Loading tables with automatic compression

Topics

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. 172) 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. 35).

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:

```
truncate biglist;
```

2. Load the table with a single COPY command. Although the table is empty, some earlier encoding may have been specified. To ensure that Amazon Redshift performs a compression analysis, set the COMPUPDATE parameter to ON.

```
copy biglist from 's3://mybucket/biglist.txt'
credentials 'aws_access_key_id=<your-access-key-id>;aws_secret_ac
cess_key=<your-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';

   Column      |            Type               | Encoding
---------------+-------------------------------+----------
listid         | integer                       | delta
sellerid       | integer                       | delta32k
eventid        | integer                       | delta32k
dateid         | smallint                      | delta
+numtickets    | smallint                      | delta
priceperticket | numeric(8,2)                  | delta32k
totalprice     | numeric(8,2)                  | mostly32
listtime       | timestamp without time zone | none
```

4. Verify that the expected number of rows were loaded:

```
select count(*) from biglist;

count
---------
3079952
(1 row)
```

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. 200).

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. 202) column is included in the column list, the EXPLICIT_IDS option must also be specified in the COPY (p. 179) 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. 190).

Troubleshooting data loads

Topics

This section provides information about identifying and resolving data loading errors.

The Amazon S3 bucket specified in your COPY command must be in the same region as your 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 ad
dressed using the specified endpoint.
```

Two Amazon Redshift system tables can be helpful in troubleshooting data load issues:

- Query STL_LOAD_ERRORS (p. 460) to discover the errors that occurred during specific loads.
- Query STL_FILE_SCAN (p. 457) 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.

   ```
   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:

   ```
   select * from loadview where table_name='venue';
   ```

   ```
     tbl   | table_name | query |         starttime
   --------+------------+-------+----------------------------
    100551 | venue      | 20974 | 2013-01-29 19:05:58.365391

   |     input      | line_number | colname | err_code |       reason
   +----------------+-------------+-------+----------+--------------------
   | venue_pipe.txt |           1 |     0 |     1214 | Delimiter not found
   ```

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. 128).

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.

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.

Error code	Description
1	The UTF-8 byte sequence exceeds the four-byte maximum supported by VARCHAR.
2	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.
3	The UTF-8 single-byte character is out of range. The starting byte must not be 254, 255 or any character between 128 and 191 (inclusive).
4	The value of the trailing byte in the byte sequence is out of range. The continuation byte must be between 128 and 191 (inclusive).
5	The UTF-8 character is reserved as a surrogate. Surrogate code points (U+D800 through U+DFFF) are invalid.
6	The character is not a valid UTF-8 character (code points 0xFDD0 to 0xFDEF).
7	The character is not a valid UTF-8 character (code points 0xFFFE and 0xFFFF).
8	The byte sequence exceeds the maximum UTF-8 code point.
9	The UTF-8 byte sequence does not have a matching code point.

Load error reference

If any errors occur while loading data from a file, query the STL_LOAD_ERRORS (p. 460) table to identify the error and determine the possible explanation. The following table lists all error codes that might occur during data loads:

Load error codes

Error code	Description
1200	Unknown parse error. Contact support.
1201	Field delimiter was not found in the input file.
1202	Input data had more columns than were defined in the DDL.
1203	Input data had fewer columns than were defined in the DDL.
1204	Input data exceeded the acceptable range for the data type.
1205	Date format is invalid. See DATEFORMAT and TIMEFORMAT strings (p. 188) for valid formats.
1206	Timestamp format is invalid. See DATEFORMAT and TIMEFORMAT strings (p. 188) for valid formats.
1207	Data contained a value outside of the expected range of 0-9.
1208	FLOAT data type format error.
1209	DECIMAL data type format error.
1210	BOOLEAN data type format error.
1211	Input line contained no data.
1212	Load file was not found.
1213	A field specified as NOT NULL contained no data.
1214	VARCHAR field error.
1215	CHAR field error.
1220	String contains invalid or unsupported UTF-8 codepoints.

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. 179) 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. 235) or CREATE TABLE AS (p. 209).

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. 75) and Vacuuming tables (p. 78).

Updating and inserting new data

You can efficiently add new data to existing tables by using a combination of updates and inserts (also known as an *upsert*) by using a staging table.

While Amazon Redshift does not support a single upsert command to insert and update data from a single data source, you can effectively perform an upsert 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.

To perform an upsert operation using staging tables, follow these steps:

1. Create a staging table that will hold the new data.
2. For the staging table, specify the same distribution key as your target table uses. That way, the joins between the two tables will be collocated. For example, if the target table uses a primary key column as the distribution key, specify the corresponding primary key column as the distribution key for the staging table.
3. In the UPDATE and INSERT statements, add a redundant join predicate between the foreign key columns in addition to the join on the primary keys to achieve a collocated join. However, this only works if the column you're using to collocate your joins is not updated as part of the upsert operation. For example, suppose your WHERE clause would normally join on the primaryKey column.

   ```
   WHERE target.primaryKey = staging.primaryKey
   ```

   Add a redundant join to also join on the distribution key, as the following example shows.

   ```
   WHERE target.primaryKey = staging.primaryKey AND target.distKey = staging.dis
   tKey
   ```

4. If your target table is sorted by timestamp, and your updates are no older than a given threshold, add a predicate (target.timestamp > threshold) to take advantage of range restricted scans on the target table.

The following example illustrates an upsert operation. Assume TARGET is the table you want to update, and STAGING is the table that contains the data that for your upsert. Note the redundant join predicates (`AND target.distKey = s.distKey`) on the distribution keys of both tables.

The first statement updates existing records in the TARGET table with new values from the STAGING table, for all matching rows on the TARGET and STAGING tables.

```
UPDATE target SET col1 = s.col1, col2 = s.col2
 FROM staging s
 WHERE target.primaryKey = s.primaryKey AND target.distKey = s.distKey;
```

The next statement inserts new rows from the STAGING table that do not yet exist in the TARGET table.

```
INSERT INTO target
SELECT s.* FROM staging s LEFT JOIN target t
ON s.primaryKey = t.primaryKey AND s.distKey = t.distKey
WHERE t.primaryKey IS NULL;
```

Analyzing tables

Topics

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. 171) 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. 171) 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

- Query predicates

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';

column         |        type        | encoding | distkey | sortkey
---------------+--------------------+----------+---------+---------
listid         | integer            | none     | t       | 1
sellerid       | integer            | none     | f       | 0
eventid        | integer            | mostly16 | f       | 0
dateid         | smallint           | none     | f       | 0
numtickets     | smallint           | mostly8  | f       | 0
priceperticket | numeric(8,2)       | bytedict | f       | 0
totalprice     | numeric(8,2)       | mostly32 | f       | 0
listtime       | timestamp with...  | none     | f       | 0
```

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:

- 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:

```
redshift_fetch_sample: select * from table_name
```

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:

```
select query, rtrim(querytxt), starttime
from stl_query
where querytxt like 'padb_fetch_sample%' and querytxt like '%sales%'
order by query desc;

query |                      rtrim                       |       starttime
------+--------------------------------------------------+---------------------
81 | redshift_fetch_sample: select * from sales        | 2012-04-18 12:...
80 | redshift_fetch_sample: select * from sales        | 2012-04-18 12:...
79 | redshift_fetch_sample: select count(*) from sales | 2012-04-18 12:...
(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:

```
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 'red
shift_fetch_sample%' )
order by xid desc, starttime;

xid  |  starttime   | secs |                  substring
-----+--------------+------+-----------------------------------------
1338 | 12:04:28.511 |    4 | Analyze date
1338 | 12:04:28.511 |    1 | redshift_fetch_sample: select count(*) from
1338 | 12:04:29.443 |    2 | redshift_fetch_sample: select * from date
1338 | 12:04:31.456 |    1 | redshift_fetch_sample: select * from date
1337 | 12:04:24.388 |    1 | redshift_fetch_sample: select count(*) from
1337 | 12:04:24.388 |    4 | Analyze sales
1337 | 12:04:25.322 |    2 | redshift_fetch_sample: select * from sales
1337 | 12:04:27.363 |    1 | redshift_fetch_sample: select * from sales
...
```

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

Run the VACUUM command following a significant number of deletes or updates.

> **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.

To perform an update, Amazon Redshift deletes the original row and appends the updated row, so every update is effectively a delete and an insert.

Amazon Redshift does not automatically reclaim and reuse space that is freed when you delete rows from tables or update rows in tables. Too many deleted rows can cost unnecessary processing. The VACUUM command reclaims space following deletes, which improves performance as well as increasing available storage. To clean up tables after a bulk delete, a load, or a series of incremental updates, you need to run the VACUUM (p. 294) command, either against the entire database or against individual tables.

In addition to reclaiming unused disk space, the VACUUM command also ensures that new data in tables is fully sorted on disk. When data is initially loaded into a table, the data is sorted according to the SORTKEY specification in the CREATE TABLE 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.

The VACUUM command merges new rows with existing sorted rows, so rows don't need to be sorted on demand during query execution. If large numbers of rows remain unsorted on disk, query performance might be degraded for operations that rely on sorted data, such as merge joins. A large unsorted region also results in longer vacuum times.

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.

- 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.
- Run VACUUM during maintenance windows or time periods when you expect minimal activity on the cluster.

You can perform queries and write operations while vacuum operations are in progress. Users can access all of the tables in the database, including the tables that are being vacuumed. The only operations that are blocked on tables being vacuumed are DDL operations. For example, if you attempt to rename a table that is being vacuumed, the ALTER TABLE command will fail.

The vacuum operation proceeds in incremental steps. If the operation fails for some reason, or if Amazon Redshift goes offline during the vacuum, the partially vacuumed table or database will be in a consistent state. You will need to manually restart the vacuum operation, but in most cases, the vacuum operation does not have to be fully repeated. Merged rows that were already committed before the failure do not need to be vacuumed again.

You can run a full vacuum, a delete only vacuum, or sort only vacuum.

- **Full vacuum** Reclaim space and sort rows. We recommend a full vacuum for most applications where reclaiming space and resorting rows are equally important. If you need to run a full vacuum, it is more

efficient to do everything in one operation than to run back-to-back DELETE ONLY and SORT ONLY vacuum operations.

- **DELETE ONLY** Reclaim space only. A DELETE ONLY vacuum reduces the elapsed time for vacuum operations 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.
- **SORT ONLY** Sort rows only. A SORT ONLY vacuum reduces the elapsed time for vacuum operations when reclaiming disk space is not important but resorting new rows is. For example, you might perform a SORT ONLY vacuum operation if your application does not have disk space constraints but does depend on query optimizations associated with keeping table rows sorted.

Managing the size of the unsorted region

When you run an initial load into an empty table (either freshly created or truncated), all of the data is loaded directly into the sorted region, so no vacuum is required. 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 techniques:

- 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.
- If you need to load the same table with multiple COPY operations, run the largest load first.
- If you are loading a small number of rows into a table for test purposes, when you are done, truncate the table and reload those rows as part of the subsequent production load operation. Alternatively, drop the table and re-create it.
- 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.
- 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 tradeoff 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 to avoid long vacuums (p. 57).

Managing concurrent write operations

Topics

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. 174) and END (p. 226) 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 Amazon Redshift 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.

```
begin
[waits]
delete from listing where ;
end;
```

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:

```
begin;
delete one row from USERS table;
copy ;
select count(*) from users;
analyze ;
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:

```
begin;
[waits]
copy users from ...;
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

To unload data from database tables to a set of files on an Amazon S3 bucket, you can use the UNLOAD (p. 282) 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.

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. 246) command reference. For more information about using a LIMIT clause, see the usage notes for the UNLOAD command.

For example, the following UNLOAD command sends the contents of the VENUE table to the Amazon S3 bucket `s3://mybucket/tickit/venue/`.

```
unload ('select * from venue')
to 's3://mybucket/tickit/venue_'
```

```
credentials 'aws_access_key_id=<your-access-key-id>;aws_secret_access_key=<your-
secret-access-key>';
```

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. 188). To unload data using temporary access credentials, use the following syntax:

```
unload ('select * from venue')
to 's3://mybucket/tickit/venue_'
credentials 'aws_access_key_id=<temporary-access-key-id>;aws_secret_ac
cess_key=<temporary-secret-access-key>;token=<temporary-token>'
[<options>];
```

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.

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. For example:

```
venue_0000_part_00
venue_0001_part_00
venue_0002_part_00
venue_0003_part_00
```

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 with query ID 2320:

```
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:

```
 query |             path
-------+--------------------------------------
  2320 | s3://my-bucket/venue0000_part_00
  2320 | s3://my-bucket/venue0001_part_00
  2320 | s3://my-bucket/venue0002_part_00
  2320 | s3://my-bucket/venue0003_part_00
(4 rows)
```

If amount of data is very large, Amazon Redshift might split the files into multiple parts per slice. For example:

```
0000_part_00
0000_part_01
0000_part_02
0001_part_00
0001_part_01
0001_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=<your-access-key-id>;aws_secret_access_key=<your-secret-ac
cess-key>';
```

If you include a prefix in the s3 path string, UNLOAD will use that prefix for the file names.

```
unload ('select venuename, venuecity from venue where venuestate=\'OH\' ')
to 's3://mybucket/tickit/venue/OH_'
credentials
'aws_access_key_id=<your-access-key-id>;aws_secret_access_key=<your-secret-ac
cess-key>';
```

The file names created by the previous example include the prefix 'OH' prefix.

```
OH_0000_part_00
OH_0001_part_00
OH_0002_part_00
OH_0003_part_00
```

By default, UNLOAD will fail rather than overwrite existing files in the destination bucket. To overwrite the existing files, specify the ALLOWOVERWRITE option.

```
unload ('select * from venue') to 's3://mybucket/venue_pipe_'
credentials 'aws_access_key_id=<your-access-key-id>;aws_secret_access_key=<your-
secret-access-key>'
allowoverwrite;
```

Unloading encrypted data files

You can create encrypted data files on 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 on 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.

```
unload ('select venuename, venuecity from venue')
to 's3://mybucket/encrypted/venue_' credentials
'aws_access_key_id=<your-access-key-id>;aws_secret_access_key=<your-secret-ac
cess-key>;master_symmetric_key=<master_key>' encrypted;
```

To unload encrypted data files that are GZIP compressed, include the GZIP option along with the master key value and the ENCRYPTED option.

```
unload ('select venuename, venuecity from venue')
to 's3://mybucket/encrypted/venue_' credentials
'aws_access_key_id=<your-access-key-id>;aws_secret_access_key=<your-secret-ac
cess-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.

```
copy venue from 's3://mybucket/encrypted/venue_' credentials
'aws_access_key_id=<your-access-key-id>;aws_secret_access_key=<your-secret-ac
cess-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:

```
unload ('select * from venue')
to 's3://mybucket/tickit/venue/comma'
credentials 'aws_access_key_id=<your-access-key-id>;aws_secret_access_key=<your-
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/tickit/venue/tab'
credentials 'aws_access_key_id=<your-access-key-id>;aws_secret_access_key=<your-
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/tickit/venue/fw'
credentials 'aws_access_key_id=<your-access-key-id>;aws_secret_access_key=<your-
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
200Al 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. 179) 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, truncated, and reloaded.

```
unload  ('select * from venue order by venueid')
to 's3://mybucket/tickit/venue/reload_'
credentials 'aws_access_key_id=<your-access-key-id>;aws_secret_access_key=<your-
secret-access-key>'
delimiter '|';

truncate venue;
```

```
copy venue from 's3://mybucket/tickit/venue/reload_'
credentials 'aws_access_key_id=<your-access-key-id>;aws_secret_access_key=<your-
secret-access-key>'
delimiter '|';
```

After it is reloaded, the VENUE table looks like this:

```
select * from venue order by venueid limit 5;

 venueid |         venuename          |  venuecity  | venuestate | venueseats
---------+----------------------------+-------------+------------+-----------
       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
(5 rows)
```

Tuning Query Performance

Topics

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. 32) 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.

Analyzing the explain plan

Topics

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. 227) 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. 171) 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. 513) 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. 97).

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:

```
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)
```

- Query the STL_EXPLAIN (p. 455) 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:

```
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;

query | nodeid | parentid |      substring      |     substring
------+--------+----------+---------------------+--------------------
10 |   1 |   0 | XN Aggregate  (cost=6350.30... |
10 |   2 |   1 |   ->  XN Merge Join DS_DIST_NO | Merge Cond: ("outer"
10 |   3 |   2 |         ->  XN Seq Scan on lis |
10 |   4 |   2 |         ->  XN Seq Scan on sal |
(4 rows
```

Simple EXPLAIN example

The following example shows the EXPLAIN output for a simple GROUP BY query against the EVENT table:

```
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:

cost
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:

- What is the relative cost of returning the first row (startup cost)? (0.00 for both operators in this example)
- What is the relative cost of completing the operation? (87.98 for the scan in this example)

> **Note**
> Cost is not an estimated time value. It is not measured in seconds or milliseconds.

rows
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).

> **Note**
> The rows estimate is based on the available statistics generated by the ANALYZE (p. 171) 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. 227) 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_NONE
No tables are distributed: collocated joins are possible because corresponding slices are joined without moving data between nodes.

DS_DIST_INNER
The inner table is distributed.

DS_DIST_BOTH
Both tables are distributed.

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_DIST_NONE  (cost=0.00..26.65 rows=740 width=104)
         Merge Cond: ("outer".listid = "inner".listid)
         ->  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)
   ->  XN Hash  (cost=2.00..2.00 rows=200 width=336)
         ->  XN Seq Scan on event  (cost=0.00..2.00 rows=200 width=336)
(11 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;
                                          QUERY PLAN
------------------------------------------------------------------------------
---
 XN Merge(cost=1000088800178.99..1000088800179.49 rows=200 width=330)
 Merge Key: sum(sales.pricepaid)
 ->XN Network (cost=1000088800178.99..1000088800179.49 rows=200 width=330)
   Send to leader
   -> XN Sort(cost=1000088800178.99..1000088800179.49 rows=200 width=330)
      Sort Key: sum(sales.pricepaid)
      -> XN HashAggregate(cost=88800170.85..88800171.35 rows=200 width=330)
        -> XN Hash Join DS_DIST_INNER(cost=9.25..880016.15 rows=740 width=330)
```

```
               Inner Dist Key: sales.eventid
               Hash Cond: ("outer".eventid = "inner".eventid)
               -> XN Seq Scan on event(cost=0.00..2.00 rows=200 width=322)
               -> XN Hash  (cost=7.40..7.40 rows=740 width=16)
                    -> XN Seq Scan on sales(cost=0.00..7.40 rows=740 width=16)
(13 rows)
```

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 (p. 513) and SVL_QUERY_REPORT (p. 509). 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

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:

```
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:

```
query| elapsed |                            substring
-----+---------+-----------------------------------------------------
1026 | 9574270 | select s.seg, s.maxtime, s.label,
                  s.is_diskbased from query_
1025 | 18672594| select t1.c1 x1, t2.c2 x2 from tbig t1, tbig
                  t2 where t1.c1
1024 | 84266   | select count(*) as underrepped from ( select
                  count(*) as a f
1023 | 83217   | select system_status from stv_gui_status
1022 | 39236   | select * from stv_sessions
(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:

```
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:

```
query| step|  rows  |  workmem  |  label        | is_diskbased
-----+-----+--------+-----------+---------------+--------------
1025 |  0  |16000000|  43205240 |scan tbl=9     | f
1025 |  2  |16000000|  43205240 |hash tbl=142   | t
1025 |  0  |16000000|  55248    |scan tbl=116536| f
1025 |  2  |16000000|  55248    |dist           | f
(4 rows)
```

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 cluster (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. 529) 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:

```
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:

   ```
   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:

   ```
                         QUERY PLAN
   ------------------------------------------------------------
   XN Hash Join DS_DIST_INNER (cost=200000.00..40836025765.37
   rows=90576537 width=8)
     Hash Cond: ("outer".c1 = "inner".c2)
     -> XN Seq Scan on tbig t1 (cost=0.00..160000.00 rows=16000000 width=4)
       -> XN Hash (cost=160000.00..160000.00 rows=16000000 width=4)
         -> XN Seq Scan on tbig t2 (cost=0.00..160000.00 rows=16000000 width=4)
   (5 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:

   ```
   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.

   ```
    query | elapsed | substring
   -------+---------+-----------------------------------------------
   1033   | 4592158 | select t1.c1 x1, t2.c2 x2 from tbig t1, tbig t2
   1032   | 477285  | select query, step, rows, memory, label,
   1031   | 23742   | select query, elapsed, substring from svl_qlog
   1030   | 900222  | select * from svl_query_summary limit 5;
   1029   | 1304248 | select query, step, rows, memory, label,
   (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:

```
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:

```
query| step| rows | workmem | label | is_diskbased
-----+-----+--------+-----------+---------------+--------------
1033 | 0 |16000000| 141557760 |scan tbl=9     | f
1033 | 2 |16000000| 135266304 |hash tbl=142   | t
1033 | 0 |16000000| 128974848 |scan tbl=116536| f
1033 | 2 |16000000| 122683392 |dist           | f
(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. 102).

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.

```
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:

```
 node  | diskno | used | capacity
-------+--------+------+----------
     0 |      0 |  245 |   1906185
     0 |      1 |  200 |   1906185
     0 |      2 |  200 |   1906185
(3 rows)
```

For more information, see STV_PARTITIONS (p. 490). 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:

```
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:

```
  tbl   | count
--------+-------
 100597 |   100
(1 row)
```

Each data block occupies 1 MB, so 100 blocks represents 100 MB of storage.

For more information, see STV_BLOCKLIST (p. 483) and STV_TBL_PERM (p. 495).

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 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. You should always run the query a second time to evaluate its performance.

The overhead cost might be especially noticeable when you run ad hoc queries. The compiled code is cached and shared across sessions in a cluster, so subsequent executions of the same query, even with different query parameters and in different sessions, will run faster because they can skip the initial generation and compilation steps.

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.

Setting the JDBC fetch size parameter

Note
Fetch size is not supported for ODBC.

By default, the JDBC driver collects all the results for at 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

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 a fixed, equal share of server memory to each queue defined in the WLM configuration. 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. 107).

For each queue, you specify

- Concurrency level
- User groups
- Query groups
- Wildcards
- WLM timeout

Concurrency 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 15 queries concurrently. The maximum total concurrency level for all user-defined queues, not including the reserved Superuser queue, is 15. Amazon Redshift allocates an equal, fixed share of memory to each query slot in a queue.

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 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. 528) 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

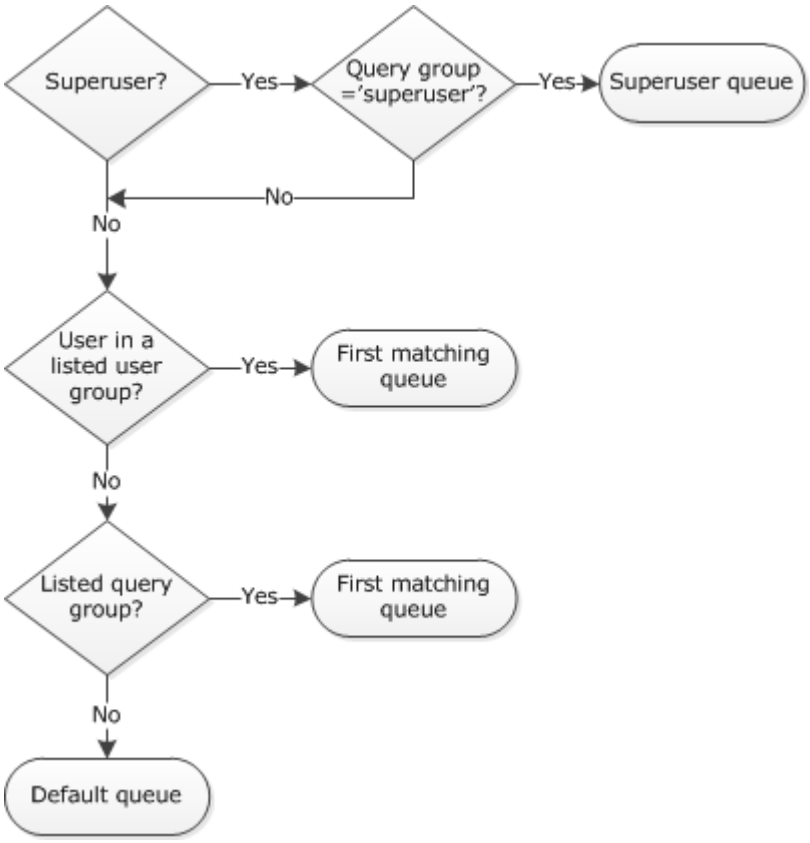
The last queue defined in the WLM configuration is the default queue. You can set the concurrency level and the timeout for the default queue, but it cannot include user groups or query groups. The default queue counts against the limit of eight query queues and the limit of 15 concurrent queries.

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.

WLM queue assignment rules

When a user runs a query, WLM assigns the query to the first matching queue, based on these rules.



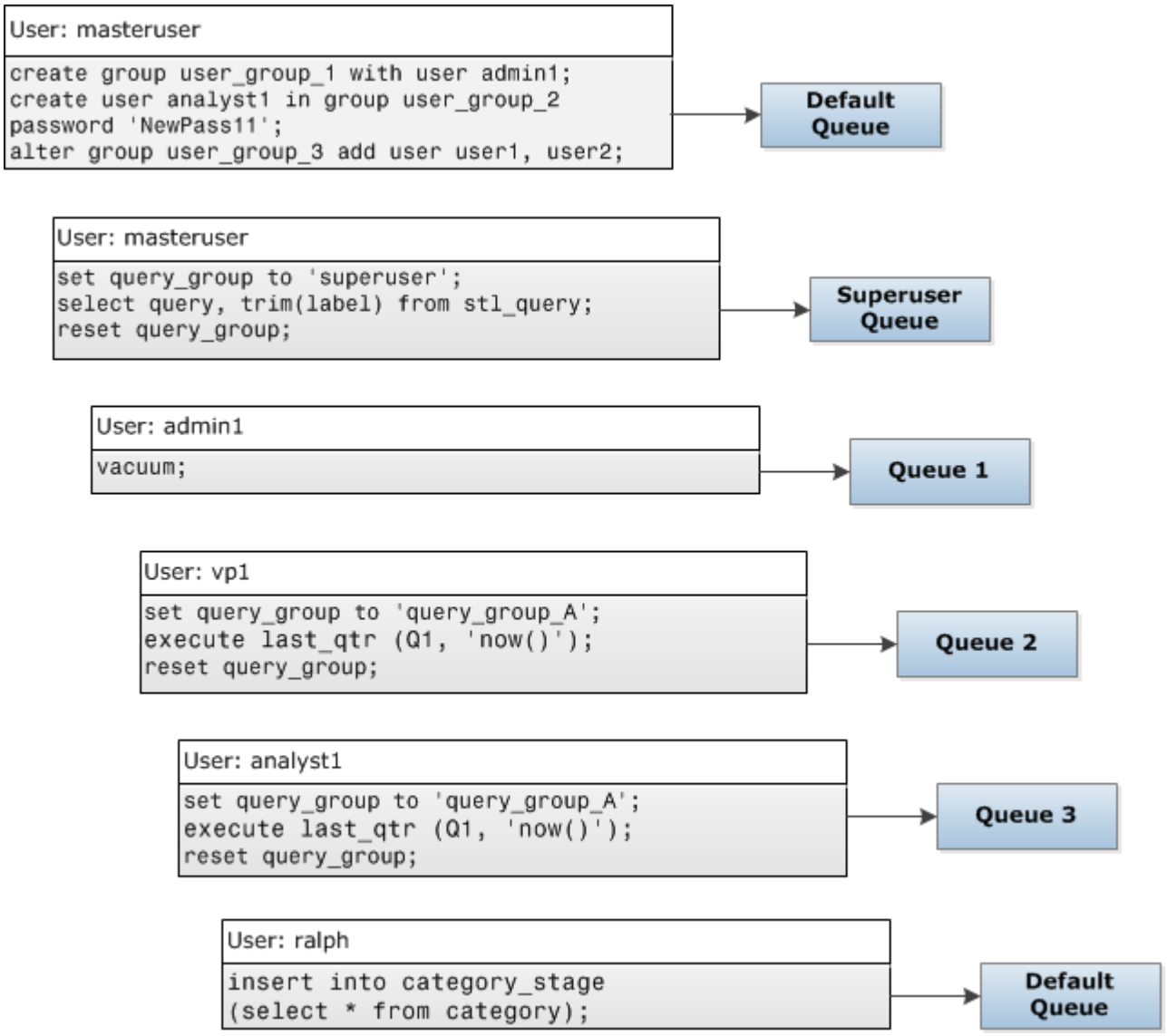
1. If a user logged in as a superuser and runs a query in the query group labeled 'superuser', the query is assigned to the Superuser queue.
2. If a user belongs to a listed user group, the query is assigned to the first corresponding queue.
3. If a user runs a query within a listed query group, the 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.

Queue	Concurrency	User Groups	Query Groups
Superuser	1		superuser
1	2	user_group_1	
2	2		query_group_A
3	4	user_group_2	query_group_B
Default	4		

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. 107) 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. 213), CREATE GROUP (p. 198), and ALTER GROUP (p. 163).

```
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. 276) and RESET (p. 242) 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. 527) 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.

```
set query_group to 'superuser';
analyze;
vacuum;
reset query_group;
```

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 name	Description
STL_WLM_ERROR (p. 478)	Contains a log of WLM-related error events.
STL_WLM_QUERY (p. 478)	Lists queries that are being tracked by WLM.
STV_WLM_CLASSIFICATION_CONFIG (p. 498)	Shows the current classification rules for WLM.
STV_WLM_QUERY_QUEUE_STATE (p. 499)	Records the current state of the query queues.
STV_WLM_QUERY_STATE (p. 500)	Provides a snapshot of the current state of queries that are being tracked by WLM.
STV_WLM_QUERY_TASK_STATE (p. 501)	Contains the current state of query tasks.
STV_WLM_SERVICE_CLASS_CONFIG (p. 502)	Records the service class configurations for WLM.
STV_WLM_SERVICE_CLASS_STATE (p. 503)	Contains the current state of the service classes.

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(ex
ec_start_time) from stl_wlm_query);

task
------
137
(1 row)
```

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;
```

```
xid |task|query|service_| wlm_start_  |  state  |queue_ | exec_
    |    |     |class   | time        |         |time   | time
----+----+-----+--------+-------------+---------+-------+--------
2645| 84 | 98  | 3      | 2010-10-... |Returning|   0   | 3438369
2650| 85 | 100 | 3      | 2010-10-... |Waiting  |   0   | 1645879
2660| 87 | 101 | 2      | 2010-10-... |Executing|   0   | 916046
2661| 88 | 102 | 1      | 2010-10-... |Executing|   0   | 13291
(4 rows)
```

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.

SQL Reference

Topics

Amazon Redshift SQL

Topics

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. 296) for a list of functions that run exclusively on the leader node.

Examples

The NOW 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.

```
select now();
```

Result

```
now
-------------------------------
2013-01-02 18:05:36.621916+00
(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.

```
select NOW(), starttime from stv_sessions;
```

Result

```
INFO:  Function "now()" not supported.
ERROR:  Specified types or functions (one per INFO message) not supported on
Redshift tables.
```

Amazon Redshift and PostgreSQL

Topics

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

Because Amazon Redshift is based on PostgreSQL 8.0.2, we recommend using the PostgreSQL 8.x JDBC and ODBC drivers to ensure compatibility, however the PostrgreSQL 9.x JDBC and ODBC drivers are backward compatible. For information about installing JDBC and ODBC drivers, see Download the Client Tools and the Drivers.

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. 102).

Amazon Redshift does not recognize the JDBC maxRows parameter. Instead, specify a LIMIT (p. 270) clause to restrict the result set. You can also use an OFFSET (p. 270) 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. 159) to understand the often subtle differences.

One example in particular is the VACUUM (p. 294) 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. 78) 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. 450) for more information.

The following list includes some examples of SQL features that are implemented differently in Amazon Redshift.

- CREATE TABLE (p. 200)

  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. 165)

  ALTER COLUMN actions are not supported.

  ADD COLUMN supports adding only one column in each ALTER TABLE statement.
- COPY (p. 179)

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. 54) section and the COPY command reference for details.

- SELECT (p. 246)

  ORDER BY ... NULLS FIRST/LAST is not supported.
- INSERT (p. 235), UPDATE (p. 290), and DELETE (p. 218)

  WITH is not supported.
- VACUUM (p. 294)

  The parameters for VACUUM are entirely different.

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. 159) 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.
- Cursors
- 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
- Indexes
- NULLS clause in Window functions
- Collations

  Amazon Redshift does not support locale-specific or user-defined collation sequences. See Collation sequences (p. 140).
- 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. 110).

For a list of the supported data types, see Data types (p. 119).

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. 159) to understand the often subtle differences.

For more information, see SQL functions supported on the leader node (p. 110).

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()
  - VARIANCE()
- 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_NULLABLE()
    - REGEXP_MATCHES()
    - REGEXP_REPLACE()
    - REGEXP_SPLIT_TO_ARRAY()
    - REGEXP_SPLIT_TO_TABLE()
    - SPLIT_PART()
    - 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
    - ROW_NUMBER()
    - PERCENT_RANK()
    - CUME_DIST()
- XML functions

Using SQL

Topics

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.

Character	Description
CAPS	Words in capital letters are key words.
[ ]	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. 246).
{ }	Braces indicate that you are required to choose one of the arguments inside the braces.
|	Pipes indicate that you can choose between the arguments.
italics	Words in italics indicate placeholders. You must insert the appropriate value in place of the word in italics.
. . .	An ellipsis indicates that you can repeat the preceding element.
'	Words in single quotes indicate that you must type the quotes.

Basic elements

Topics

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 72 characters in length
- 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:

Syntax	Result	Discussion
"group"	group	GROUP is a reserved word, so usage of it within an identifier requires double quotes.
""WHERE""	"where"	WHERE is also a reserved word. To include quotation marks in the string, escape them with additional quotation marks.
"This name"	this name	Double quotes are required in order to preserve the space.
"This ""IS IT"""	this "is it"	The quotes surrounding IS IT must each be preceded by an extra quote in order to become part of the name.

To create a table named group with a column named this "is it":

```
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

- Type compatibility and conversion (p. 137)

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, and when stored procedures and functions 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.

Data type	Aliases	Description
SMALLINT	INT2	Signed two-byte integer
INTEGER	INT, INT4	Signed four-byte integer
BIGINT	INT8	Signed eight-byte integer
DECIMAL	NUMERIC	Exact numeric of selectable precision
REAL	FLOAT4	Single precision floating-point number
DOUBLE PRECISION	FLOAT8	Double precision floating-point number
BOOLEAN	BOOL	Logical Boolean (true/false)
CHAR	CHARACTER	Fixed-length character string
VARCHAR	CHARACTER VARYING	Variable-length character string with a user-defined limit
DATE		Calendar date (year, month, day)
TIMESTAMP		Date and time (without time zone)

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)
- 0xFDD0 - 0xFDEF, 0xFFFE, 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

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.

Name	Storage	Range
SMALLINT or INT2	2 bytes	-32768 to +32767
INTEGER, INT, or INT4	4 bytes	-2147483648 to +2147483647
BIGINT or INT8	8 bytes	-9223372036854775807 to 9223372036854775807

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.

Storage	Range
Variable, up to 128 bits for uncompressed DECIMAL types with greater than 19 digits precision.	128-bit signed integers with up to 38 digits of precision.

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.

```
insert into sales
values (0, 8, 1, 1, 2000, 14, 5, 4323.8951, 11.00, null);

select pricepaid, salesid from sales where salesid=0;

pricepaid | salesid
-----------+---------
4323.90 |       0
(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. 429).
- 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.

Name	Storage	Range
REAL or FLOAT4	4 bytes	6 significant digits of precision

Name	Storage	Range
DOUBLE PRECISION, FLOAT8, or FLOAT	8 bytes	15 significant digits of precision

For example, note the results of the following inserts into a REAL column:

```
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. 297).

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.

	INT2	INT4	INT8	DECIMAL	FLOAT4	FLOAT8
INT2	INT2	INT4	INT8	DECIMAL	FLOAT8	FLOAT8
INT4	INT4	INT4	INT8	DECIMAL	FLOAT8	FLOAT8
INT8	INT8	INT8	INT8	DECIMAL	FLOAT8	FLOAT8
DECIMAL	DECIMAL	DECIMAL	DECIMAL	DECIMAL	FLOAT8	FLOAT8
FLOAT4	FLOAT8	FLOAT8	FLOAT8	FLOAT8	FLOAT4	FLOAT8
FLOAT8	FLOAT8	FLOAT8	FLOAT8	FLOAT8	FLOAT8	FLOAT8

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, `p1` and `s1` represent the precision and scale of the first operand in a calculation and `p2` and `s2` 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.)

Operation	Result precision and scale
+ or -	Scale = `max(s1,s2)` Precision = `max(p1-s1,p2-s2)+1+scale`
*	Scale = `s1+s2` Precision = `p1+p2+1`
/	Scale = `max(4,s1+p2-s2+1)` Precision = `p1-s1+ s2+scale`

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:

```
Precision = 8-2 + 2 + max(4,2+8-2+1)
= 6 + 2 + 9 = 17

Scale = max(4,2+8-2+1) = 9

Result = 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:

```
Scale = max(s1,s2)
Precision = min(max(p1-s1,p2-s2)+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:

```
Precision = min(max(7-2,15-3) + max(2,3), 19)
= 12 + 3 = 15

Scale = max(2,3) = 3

Result = DECIMAL(15,3)
```

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(scale - (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:

  ```
  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. 429). 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 100000000000000000000, cast the expression as follows:

  ```
  select salesid, pricepaid::decimal(38,2) * 100000000000000000000
  as value from sales where salesid=2;

  salesid |           value
  ---------+--------------------------
  2 | 760000000000000000000.00
  (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 | E
e or E indicates that the number is specified in scientific notation.

Examples

Valid floating-point literals include:

```
3.14159
-37.
2.0e19
-2E-19
```

Examples with numeric types

CREATE TABLE statement

The following CREATE TABLE statement demonstrates the declaration of different numeric data types:

```
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.

```
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 the a decimal value into an INT column.

```
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 that the column.

```
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

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.

Name	Storage	Range (width of column)
CHAR or CHARACTER	Length of string, including trailing blanks (if any)	4096 bytes

Name	Storage	Range (width of column)
VARCHAR or CHARACTER VARYING	4 bytes + total bytes for characters, where each character can be 1 to 4 bytes.	65535 bytes (64K -1)

Note
The CREATE TABLE syntax supports the MAX keyword for character data types. For example:

```
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.

```
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.

```
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:

```
select length(address1) from address;

length
--------
23
25
(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:

```
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:

```
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

Datetime data types include DATE and TIMESTAMP.

Storage and ranges

Name	Storage	Range	Resolution
DATE	4 bytes	4713 BC to 5874897 AD	1 day
TIMESTAMP	8 bytes	4713 BC to 5874897 AD	1 microsecond

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:

```
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;

start_date |  end_date
-----------------------
2008-06-01 | 2008-12-31
2008-06-01 | 2008-12-31
```

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:

```
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
---------------------
```

```
2008-12-31 09:59:59
2008-12-31 18:20:00
(2 rows)
```

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.

Input date	Full date
January 8, 1999	January 8, 1999
1999-01-08	January 8, 1999
1/8/1999	January 8, 1999
01/02/00	January 2, 2000
2000-Jan-31	January 31, 2000
Jan-31-2000	January 31, 2000
31-Jan-2000	January 31, 2000
20080215	February 15, 2008
080215	February 15, 2008
2008.366	December 31, 2008 (3-digit part of date must be between 001 and 366)

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.

Input timestamps (concatenated dates and times)	Description (of time part)
20080215 04:05:06.789	4:05 am and 6.789 seconds
20080215 04:05:06	4:05 am and 6 seconds
20080215 04:05	4:05 am exactly
20080215 040506	4:05 am and 6 seconds
20080215 04:05 AM	4:05 am exactly; `AM` is optional
20080215 04:05 PM	4:05 pm exactly; hour value must be < 12.
20080215 16:05	4:05 05 pm exactly

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.

	Description
now	Evaluates to the start time of the current transaction and returns a timestamp with microsecond precision.
today	Evaluates to the appropriate date and returns a timestamp with zeroes for the timeparts.
tomorrow	
yesterday	

The following examples show how `now` and `today` work in conjunction with the DATEADD function:

```
select dateadd(day,1,'today');

date_add
---------------------
2009-11-17 00:00:00
(1 row)

select dateadd(day,1,'now');

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';
```

```
dateplus
---------------------
2008-12-31 12:00:00
(1 row)
```

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.

State	Valid literal values	Storage
True	`TRUE 't' 'true' 'y' 'yes' '1'`	1 byte
False	`FALSE 'f' 'false' 'n' 'no' '0'`	1 byte
Unknown	`NULL`	1 byte

Examples

You could use a BOOLEAN column to store an "Active/Inactive" state for each customer in a CUSTOMER table:

```
create table customer(
custid int,
active_flag boolean default true
);
```

```
insert into customer values(100, default);
```

```
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:

```
select firstname, lastname, likesports, liketheatre
from users
where likesports is true and liketheatre is false
order by userid limit 10;

firstname |  lastname  | likesports | liketheatre
-----------+------------+------------+-------------
```

```
Lars      | Ratliff    | t          | f
Mufutau   | Watkins    | t          | f
Scarlett  | Mayer      | t          | f
Shafira   | Glenn      | t          | f
Winifred  | Cherry     | t          | f
Chase     | Lamb       | t          | f
Liberty   | Ellison    | t          | f
Aladdin   | Haney      | t          | f
Tashya    | Michael    | t          | f
Lucian    | Montgomery | t          | f
(10 rows)
```

This example selects users from the USERS table for whom is it unknown whether they like rock music:

```
select firstname, lastname, likerock
from users
where likerock is unknown
order by userid limit 10;

firstname | lastname | likerock
----------+----------+----------
Rafael    | Taylor   |
Vladimir  | Humphrey |
Barry     | Roy      |
Tamekah   | Juarez   |
Mufutau   | Watkins  |
Naida     | Calderon |
Anika     | Huff     |
Bruce     | Beck     |
Mallory   | Farrell  |
Scarlett  | Mayer    |
(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. 138)). 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. 429).

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:

From type	To type
BIGINT (INT8)	VARCHAR
	CHAR
	SMALLINT (INT2)
	INTEGER (INT, INT4)
DATE	VARCHAR
	CHAR

From type	To type
DECIMAL (NUMERIC)	VARCHAR
	CHAR
	SMALLINT (INT2)
	INTEGER (INT, INT4)
	BIGINT (INT8)
DOUBLE PRECISION (FLOAT8)	DECIMAL (NUMERIC)
	VARCHAR
	CHAR
	REAL (FLOAT4)
	SMALLINT (INT2)
	INTEGER (INT, INT4)
	BIGINT (INT8)
INTEGER (INT, INT4)	VARCHAR
	CHAR
	SMALLINT (INT2)
REAL (FLOAT4)	DECIMAL (NUMERIC)
	VARCHAR
	CHAR
	SMALLINT (INT2)
	INTEGER (INT, INT4)
	BIGINT (INT8)
SMALLINT (INT2)	VARCHAR
	CHAR
TIMESTAMP	DATE
	VARCHAR
	CHAR

The following table lists the data types that can be converted implicitly in expressions only:

From type	To type
BIGINT (INT8)	BOOLEAN
	DECIMAL (NUMERIC)
	DOUBLE PRECISION (FLOAT8)
	REAL (FLOAT4)
CHAR	VARCHAR
	CHAR
DATE	TIMESTAMP
DECIMAL (NUMERIC)	DECIMAL (NUMERIC)
	DOUBLE PRECISION (FLOAT8)
	REAL (FLOAT4)
INTEGER (INT, INT4)	BOOLEAN
	DECIMAL (NUMERIC)
	DOUBLE PRECISION (FLOAT8)
	REAL (FLOAT4)
	BIGINT (INT8)
REAL (FLOAT4)	DOUBLE PRECISION (FLOAT8)
SMALLINT (INT2)	BOOLEAN
	DECIMAL (NUMERIC)
	DOUBLE PRECISION (FLOAT8)
	REAL (FLOAT4)
	INTEGER (INT, INT4)
	BIGINT (INT8)
TIMESTAMP	TIMESTAMP
VARCHAR	DECIMAL (NUMERIC)
	VARCHAR
	CHAR (from single-byte VARCHAR only)

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

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

Examples of simple expressions include:

```
5+12
dateid
sales.qtysold * 100
sqrt (4)
max (qtysold)
(select max (qtysold) from sales)
```

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

(compound_expression)
Compound expressions may be nested using parentheses.

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:

```
sum(abs(qtysold))
```

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;

venueid |        venuename        | venuecity | venuestate | venueseats
---------+-------------------------+-----------+------------+------------
28 | American Airlines Arena | Miami     | FL         |          0
54 | St. Pete Times Forum    | Tampa     | FL         |          0
91 | Raymond James Stadium   | Tampa     | FL         |      65647
(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;
qtr  | avg_saleprice_per_qtr | avg_saleprice_yearly
-------+----------------------+----------------------
1     |               647.64 |               642.28
2     |               646.86 |               642.28
3     |               636.79 |               642.28
4     |               638.26 |               642.28
(4 rows)
```

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

```
abs (variable)
select avg (qtysold + 3) from sales;
select dateadd (day,30,caldate) as plus30days from date;
```

Conditions

Topics

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

```
comparison_condition
| logical_condition
| range_condition
| pattern_matching_condition
| null_condition
| EXISTS_condition
| IN_condition
```

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:

Operator	Syntax	Description
<	`a < b`	Value a is less than value b.
>	`a > b`	Value a is greater than value b.
<=	`a <= b`	Value a is less than or equal to value b.
>=	`a >= b`	Value a is greater than or equal to value b.
=	`a = b`	Value a is equal to value b.
<> or !=	`a <> b or a != b`	Value a is not equal to value b.
ANY | SOME	`a = ANY(subquery)`	Value a is equal to any value returned by the subquery.
ALL	`a <> ALL or != ALL(subquery)`	Value a is not equal to any value returned by the subquery.
IS TRUE | FALSE | UNKNOWN	`a IS TRUE`	Value a is Boolean TRUE.

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. 158) 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. 136) data type.

Examples

Here are some simple examples of comparison conditions:

```
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:

```
select venueid, venuename, venueseats from venue
where venueseats > 10000
order by venueseats desc;

venueid |           venuename            | venueseats
---------+--------------------------------+------------
83 | FedExField                     |      91704
 6 | New York Giants Stadium        |      80242
79 | Arrowhead Stadium              |      79451
78 | INVESCO Field                  |      76125
69 | Dolphin Stadium                |      74916
67 | Ralph Wilson Stadium           |      73967
76 | Jacksonville Municipal Stadium |      73800
89 | Bank of America Stadium        |      73298
72 | Cleveland Browns Stadium       |      73200
86 | Lambeau Field                  |      72922
...
(57 rows)
```

This example selects the users (USERID) from the USERS table who like rock music:

```
select userid from users where likerock = 't' order by 1 limit 5;

userid
--------
3
5
6
13
16
(5 rows)
```

This example selects the users (USERID) from the USERS table where it is unknown whether they like rock music:

```
select firstname, lastname, likerock
from users
where likerock is unknown
order by userid limit 10;
```

```
firstname | lastname | likerock
----------+----------+----------
Rafael    | Taylor   |
Vladimir  | Humphrey |
Barry     | Roy      |
Tamekah   | Juarez   |
Mufutau   | Watkins  |
Naida     | Calderon |
Anika     | Huff     |
Bruce     | Beck     |
Mallory   | Farrell  |
Scarlett  | Mayer    |
(10 rows
```

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 E1 and E2 represent expressions:

E1	E2	E1 AND E2	E1 OR E2	NOT E2
TRUE	TRUE	TRUE	TRUE	FALSE
TRUE	FALSE	FALSE	TRUE	TRUE
TRUE	UNKNOWN	UNKNOWN	TRUE	UNKNOWN
FALSE	TRUE	FALSE	TRUE	
FALSE	FALSE	FALSE	FALSE	
FALSE	UNKNOWN	FALSE	UNKNOWN	
UNKNOWN	TRUE	UNKNOWN	TRUE	
UNKNOWN	FALSE	FALSE	UNKNOWN	
UNKNOWN	UNKNOWN	UNKNOWN	UNKNOWN	

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:

```
select userid, username from users
where likevegas = 1 and likesports = 1
order by userid;

userid | username
--------+----------
1 | JSG99FHE
67 | TWU10MZT
87 | DUF19VXU
92 | HYP36WEQ
109 | FPL38HZK
120 | DMJ24GUZ
123 | QZR22XGQ
130 | ZQC82ALK
133 | LBN45WCH
144 | UCX04JKN
165 | TEY68OEB
169 | AYQ83HGO
184 | TVX65AZX
...
(2128 rows)
```

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.

```
select userid, username from users
where likevegas = 1 or likesports = 1
order by userid;

userid | username
--------+----------
1 | JSG99FHE
2 | PGL08LJI
3 | IFT66TXU
5 | AEB55QTM
6 | NDQ15VBM
9 | MSD36KVR
10 | WKW41AIW
13 | QTF33MCG
15 | OWU78MTR
16 | ZMG93CDD
22 | RHT62AGI
27 | KOY02CVE
29 | HUH27PKK
...
(18968 rows)
```

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;

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
(4 rows)
```

Pattern-matching conditions

Topics

- LIKE (p. 150)
- SIMILAR TO (p. 152)
- POSIX operators (p. 155)

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)%';
select 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 wildcard characters in the pattern. The default is a backslash (\).

If *pattern* does not contain wildcards, 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:

Operator	Description
%	Matches any sequence of zero or more characters.
_	Matches any single character.

Examples

The following table shows examples of pattern matching using LIKE:

Expression	Returns
`'abc' LIKE 'abc'`	True
`'abc' LIKE 'a%'`	True
`'abc' LIKE '_B_'`	False
`'abc' ILIKE '_B_'`	True
`'abc' LIKE 'c%'`	False

The following example finds all cities whose names start with "E":

```
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" :

```
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:

```
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 character (\) to search for strings that include "_":

```
select tablename, "column" from pg_table_def
where "column" like '%start\_time'
limit 5;

   tablename    |          column
----------------+-------------------------
 stl_s3client   | start_time
 stl_unload_log | start_time
 stl_wlm_query  | service_class_start_time
 stl_wlm_query  | queue_start_time
 stl_wlm_query  | exec_start_time
(5 rows)
```

The following example specifies '^' as the escape character, then uses the escape character to search for strings that include "_":

```
select tablename, "column" from pg_table_def
where "column" like '%start^_time' escape '^'
limit 5;

   tablename    |          column
----------------+-------------------------
 stl_s3client   | start_time
 stl_unload_log | start_time
 stl_wlm_query  | service_class_start_time
 stl_wlm_query  | queue_start_time
 stl_wlm_query  | exec_start_time
(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. 150) 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:

```
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 representing a SQL standard regular expression pattern..

escape_char
A character expression that will escape wildcard characters in the pattern. The default is a backslash (\).

If *pattern* does not contain wildcards, 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:

Operator	Description
%	Matches any sequence of zero or more characters.
_	Matches any single character.
|	Denotes alternation (either of two alternatives).
*	Repeat the previous item zero or more times.
+	Repeat the previous item one or more times.
?	Repeat the previous item zero or one time.
{m}	Repeat the previous item exactly *m* times.
{m,}	Repeat the previous item *m* or more times.
{m,n}	Repeat the previous item at least *m* and not more than *n* times.
()	Parentheses group items into a single logical item.
[...]	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:

Expression	Returns
'abc' SIMILAR TO 'abc'	True
'abc' SIMILAR TO '_a_'	True
'abc' SIMILAR TO '_A_'	False
'abc' SIMILAR TO '%(b|d)%'	True
'abc' SIMILAR TO '(b|c)%'	True
'AbcAbcdefgefg12efgefg12' SIMILAR TO '((Ab)?c)+d((efg)+(12))+'	True
'aaaaaab11111xy' SIMILAR TO 'a{6}_ [0-9]{5}(x|y){2}'	True
'$0.87' SIMILAR TO '$[0-9]+(.[0-9][0-9])?'	True

The following example finds all cities whose names contain "E" or "H":

```
select distinct city from users
where city similar to '%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 default escape character (\) to search for strings that include "_":

```
select tablename, "column" from pg_table_def
where "column" similar to '%start\_time'
limit 5;
   tablename    |          column
----------------+-------------------------
 stl_s3client   | start_time
 stl_unload_log | start_time
 stl_wlm_query  | service_class_start_time
 stl_wlm_query  | queue_start_time
 stl_wlm_query  | exec_start_time
(5 rows)
```

The following example specifies '^' as the escape character, then uses the escape character to search for strings that include "_":

```
select tablename, "column" from pg_table_def
where "column" similar to '%start^_time' escape '^'
limit 5;
   tablename    |          column
----------------+-------------------------
 stl_s3client   | start_time
 stl_unload_log | start_time
 stl_wlm_query  | service_class_start_time
 stl_wlm_query  | queue_start_time
 stl_wlm_query  | exec_start_time
(5 rows)
```

POSIX operators

POSIX regular expressions provide a more powerful means for pattern matching than the LIKE (p. 150) and SIMILAR TO (p. 152) operators. POSIX regular expression patterns can match any portion of the 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:

```
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.

SIMILAR TO
SIMILAR TO performs a case-sensitive pattern match for the entire string in *expression*.

pattern
A valid UTF-8 character representing a SQL standard regular expression pattern..

If *pattern* does not contain wildcards, 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*.

All of the character expressions can be CHAR or VARCHAR data types. If they differ, Amazon Redshift converts them to the data type of *expression*.

POSIX regular expressions use the same metacharacters as SIMILAR TO (p. 152), except for the wildcard characters shown in the following table:

POSIX	SIMILAR TO	Description
. (dot)	_ (underscore)	Matches a single character.

POSIX	SIMILAR TO	Description
.*(dot and star)	% (percent)	Matches any sequence of zero or more characters.
^ (caret)	Not supported	Matches the beginning of the string.
>=	Not supported	Matches the end of the string.

Examples

The following table shows examples of pattern matching using SIMILAR TO:

Expression	Returns
'abc' ~ 'abc'	True
'abc' ~ 'a'	True
'abc' ~ 'A'	False
'abc' ~ '.(b|d).*'*	True
'abc' ~ '(b|c).'*	True
'AbcAbcdefgefg12efgefg12' ~ '((Ab)?c)+d((efg)+(12))+'	True
'aaaaaab11111xy' ~ 'a{6}.[[1]]{5}(x|y){2}'	True
“$0.87' ~ '$[0-9]+(\.[0-9][0-9])?'	True

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:

```
select count(*) from sales
where qtysold between 2 and 4;

count
--------
104021
(1 row)
```

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:

```
select count(*) from sales
where qtysold between 4 and 2;

count
-------
0
(1 row)
```

However, applying the NOT modifier will invert the logic and produce a count of all rows:

```
select count(*) from sales
where qtysold not between 4 and 2;

count
--------
172456
(1 row)
```

The following query returns a list of venues with 20000 to 50000 seats:

```
select venueid, venuename, venueseats from venue
where venueseats between 20000 and 50000
order by venueseats desc;

venueid |       venuename               | venueseats
---------+-------------------------------+------------
116 | Busch Stadium                 |      49660
106 | Rangers BallPark in Arlington |      49115
96 | Oriole Park at Camden Yards   |      48876
...
(22 rows)
```

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:

```
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:

```
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:

```
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:

```
explain select * from sales
where salesid in (1,2,3,4,5,6,7,8,9,10,11);
QUERY PLAN
------------------------------------------------------------------------
XN Seq Scan on sales  (cost=0.00..6035.96 rows=86228 width=53)
Filter: (salesid = ANY ('{1,2,3,4,5,6,7,8,9,10,11}'::integer[]))
(2 rows)
```

SQL Commands

Topics

- TRUNCATE (p. 281)
- UNLOAD (p. 282)
- UPDATE (p. 290)
- VACUUM (p. 294)

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. 111).

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. 245) 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:

```
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:

```
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:

```
select * from movie_gross;
```

The command output shows that both rows are successfully inserted:

```
name           |  gross
-------------------------+----------
Raiders of the Lost Ark | 23400000
Star Wars               | 10000000
(2 rows)
```

This command now rolls back the data changes to where the transaction began:

```
abort;
```

Selecting data from the table now shows an empty table:

```
select * from movie_gross;

name | gross
------+-------
(0 rows)
```

ALTER DATABASE

Changes the attributes of a database.

Synopsis

```
ALTER DATABASE database_name
{
RENAME TO new_name |
OWNER TO new_owner |
}
```

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 tickit rename to newtickit;
ERROR:  current database may not be renamed
```

RENAME TO
Renames the specified database. 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. 233) 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 tickit_sandbox rename to tickit_test;
```

The following example changes the owner of the TICKIT database (the current database) to DWUSER:

```
alter database tickit 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

```
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.

new_name
New name of the user group.

Examples

The following example adds a user named DWUSER to the ADMIN_GROUP group:

```
alter group admin_group
add user dwuser;
```

The following example renames the group ADMIN_GROUP to ADMINISTRATORS:

```
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. 199) for more information about schemas.

Synopsis

```
ALTER SCHEMA schema_name
{
RENAME TO new_name |
OWNER TO new_owner
}
```

Parameters

schema_name
Name of the database schema to be altered.

RENAME TO
Renames the schema.

new_name
The new name of the schema.

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:

```
alter schema sales
rename to us_sales;
```

The following example gives ownership of the US_SALES schema to the user DWUSER:

```
alter schema us_sales
owner to dwuser;
```

ALTER TABLE

Topics

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

```
ALTER TABLE table_name
{
ADD table_constraint |
DROP table_constraint [ RESTRICT | CASCADE ] |
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. 200) 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. 200) 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.

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. 119)

- 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. 36) 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:

```
select "column", type, encoding, distkey, sortkey
from pg_table_def where tablename = 'users';

    column     |          type          | encoding | distkey | sortkey
---------------+------------------------+----------+---------+---------
 userid        | integer                | none     | t       |       1
 username      | character(8)           | none     | f       |       0
 firstname     | character varying(30)  | text32k  | f       |       0
 lastname      | character varying(30)  | text32k  | f       |       0
 city          | character varying(30)  | text32k  | f       |       0
 state         | character(2)           | none     | f       |       0
 email         | character varying(100) | text255  | f       |       0
 phone         | character(14)          | none     | f       |       0
 likesports    | boolean                | none     | f       |       0
 liketheatre   | boolean                | none     | f       |       0
 likeconcerts  | boolean                | none     | f       |       0
 likejazz      | boolean                | none     | f       |       0
 likeclassical | boolean                | none     | f       |       0
 likeopera     | boolean                | none     | f       |       0
 likerock      | boolean                | none     | f       |       0
 likevegas     | boolean                | none     | f       |       0
 likebroadway  | boolean                | none     | f       |       0
```

```
 likemusicals  | boolean                | none     | f       |       0
 (18 rows)
```

Now add the feedback_score column:

```
alter table users
add column feedback_score int
default NULL;
```

Select the FEEDBACK_SCORE column from USERS to verify that it was added:

```
select feedback_score from users limit 5;

feedback_score
----------------

(5 rows)
```

Drop the column to reinstate the original DDL:

```
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:

```
alter table users
add column feedback_score int
default NULL;
```

Next, create a view from the USERS table called USERS_VIEW:

```
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):

```
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:

```
alter table users
drop column feedback_score cascade;
```

ALTER USER

Changes a database user account.

Synopsis

```
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 gives the user the privilege to create accounts. NOCREATEUSER is the default.

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.

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.

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. 527) 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. 75).

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. 40) 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:

```
analyze compression listing;

  Table  |     Column     | Encoding
---------+----------------+----------
listing | listid         | delta
listing | sellerid       | delta32k
listing | eventid        | delta32k
listing | dateid         | bytedict
listing | numtickets     | bytedict
listing | priceperticket | delta32k
listing | totalprice     | mostly32
listing | listtime       | raw
```

Analyze the QTYSOLD, COMMISSION, and SALETIME columns in the SALES table:

```
analyze compression sales(qtysold, commission, saletime);
Table |   Column   | Encoding
```

```
------+------------+----------
sales | salesid    | N/A
sales | listid     | N/A
sales | sellerid   | N/A
sales | buyerid    | N/A
sales | eventid    | N/A
sales | dateid     | N/A
sales | qtysold    | bytedict
sales | pricepaid  | N/A
sales | commission | delta32k
sales | saletime   | raw
```

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. 281) 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. 79). 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:

```
select pid, starttime, duration,
trim(user_name) as user,
trim (query) as querytxt
from stv_recents
where status = 'Running';

pid |         starttime          | duration |   user   |    querytxt
-----+----------------------------+----------+----------+-----------------
802 | 2008-10-14 09:19:03.550885 |      132 | dwuser | select
venuename from venue where venuestate='FL', where venuecity not in
('Miami' , 'Orlando');
834 | 2008-10-14 08:33:49.473585 |  1250414 | dwuser | select *
from listing;
964 | 2008-10-14 08:30:43.290527 |   326179 | dwuser | select
sellerid from sales where qtysold in (8, 10);
```

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:

```
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:

```
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. 178)COMMIT, END (p. 226) and ROLLBACK (p. 245)ROLLBACK 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. 216), FETCH (p. 231).

Synopsis

```
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:

```
close movie_cursor;
commit;
```

COMMENT

Creates or changes a comment about a database object.

Synopsis

```
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.

Example

The following example adds a descriptive comment to the EVENT table:

```
comment on table
event is 'Contains listings of individual events.';
```

Result

```
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

Loads data into a table from flat files located in an Amazon S3 bucket or from an Amazon DynamoDB table. 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.

Copy from Amazon S3 Synopsis

```
COPY table_name [ (column1 [,column2, ...]) ]
FROM 's3://objectpath'
[ WITH ] CREDENTIALS [AS] 'aws_access_credentials'
[ option [ ... ] ]

where option is

{ FIXEDWIDTH 'fixedwidth_spec'
| [DELIMITER [ AS ] 'delimiter_char']
  [CSV [QUOTE [ AS ] 'quote_character']}

| ENCRYPTED
| GZIP
| REMOVEQUOTES
| EXPLICIT_IDS
| ACCEPTINVCHARS [ 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
| TRUNCATECOLUMNS
| FILLRECORD
| TRIMBLANKS
| NOLOAD
| NULL [ AS ] 'null_string'
| EMPTYASNULL
| BLANKSASNULL
```

```
| COMPROWS numrows
| COMPUPDATE [ { ON | TRUE } | { OFF | FALSE } ]
| STATUPDATE [ { ON | TRUE } | { OFF | FALSE } ]
| ESCAPE
| ROUNDEC
```

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

| EXPLICIT_IDS
| MAXERROR [ AS ] error_count
| DATEFORMAT [ AS ] { 'dateformat_string' | 'auto' }
| TIMEFORMAT [ AS ] { 'timeformat_string' | 'auto' | 'epochsecs' | 'epochmilli
secs' }
| ACCEPTANYDATE
| TRUNCATECOLUMNS
| TRIMBLANKS
| NOLOAD
| EMPTYASNULL
| BLANKSASNULL
| COMPROWS numrows
| COMPUPDATE [ { ON | TRUE } | { OFF | FALSE } ]
| STATUPDATE [ { ON | TRUE } | { OFF | FALSE } ]
| ROUNDEC
```

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 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

You can copy data from files or Amazon DynamoDB to tables. Use the UNLOAD (p. 282) command to copy data from a table to a set of files.

's3://*objectpath*'

The path to the Amazon S3 objects that contain the data. For example, `'s3://mybucket/cust.txt'`. *s3://objectpath* can be a reference to a single file or it can reference a set of objects or folders using a 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. If a key prefix references multiple folders, all of the files in the folders will be loaded.

> **Important**
> The Amazon S3 bucket that holds the data files must be created in the same region as your cluster.

The maximum size of a single input row is 4 megabytes.

'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. 66).

An Amazon DynamoDB table name is unique to an AWS account, which is identified by the AWS access credentials.

WITH

This keyword is optional.

CREDENTIALS [AS] '*aws_access_credentials*'

The AWS account access credentials for the Amazon S3 bucket or Amazon DynamoDB table that contains the data. 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. 188) in Usage notes.

> **Note**
> These examples contain line breaks for readability. Do not include line breaks or spaces in your *aws_access_credentials* string.

The *aws_access_credentials* string must not contain spaces.

If these credentials correspond to an IAM user, that IAM user must have permission to GET and LIST the Amazon S3 objects that are being loaded or permission to SCAN and DESCRIBE the Amazon DynamoDB table that is being loaded. The *aws_access_credentials* string is in the following format:

```
'aws_access_key_id=<your-access-key-id>;
aws_secret_access_key=<your-secret-access-key>'
```

If only access key and secret access key are required, the *aws_access_credentials* string is in the following format:

```
'aws_access_key_id=<your-access-key-id>;
aws_secret_access_key=<your-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 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 format

```
'aws_access_key_id=<your-access-key-id>;
aws_secret_access_key=<your-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. The format for *fixedwidth_spec* is shown below:

```
'colLabel1:colWidth1,colLabel:colWidth2, ...'
```

DELIMITER [AS] ['*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 '\ddd', 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 `A "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.

ENCRYPTED
Specifies that the input files on Amazon S3 are encrypted. See Loading encrypted data files from Amazon S3 (p. 65). 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.

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.

Input string	Loaded value with REMOVEQUOTES option
"The delimiter is a pipe (|) character"	The delimiter is a pipe (|) character
'Black'	Black
"White"	White
Blue'	Blue'
'Blue	*Value not loaded: error condition*
"Blue	*Value not loaded: error condition*
' ' 'Black' ' '	' 'Black' '
' '	*<white space>*

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. 72).

COPY returns the number of rows that contained invalid UTF-8 characters, and it adds an entry to the STL_REPLACEMENTS (p. 466) 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. For more information, see Using Automatic Recognition with DATEFORMAT and TIMEFORMAT (p. 189).

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. 188) .

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. 188).

If you want Amazon Redshift to automatically recognize and convert the time format in your source data, specify `'auto'`. For more information, see Using Automatic Recognition with DATEFORMAT and TIMEFORMAT (p. 189).

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.

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'.

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) are mutually exclusive options that 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. 68).

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. 75).

Whenever ingesting data into a nonempty table significantly changes the size of the table, we recommend updating statistics either by running an ANALYZE (p. 171) 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
By default, the COPY command does not round up numeric values whose scale exceeds the scale of the column. If you want these values to be rounded up, use the ROUNDEC option. For example, when the ROUNDEC option is used and a value of `20.259` is loaded into a DECIMAL(8,2) column, the value that is stored is `20.26`. If the ROUNDEC option is not used, the value that is stored is `20.25`. The INSERT command behaves the same as the COPY command with the ROUNDEC option.

Usage notes

Topics

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. 120).

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. 528)) 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. 70).

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>';
```

> **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.

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":

Datepart/timepart	Meaning
YY	Year without century
YYYY	Year with century
MM	Month as a number

Datepart/timepart	Meaning
MON	Month as a name (abbreviated name or full name)
DD	Day of month as a number
HH or HH24	Hour (24-hour clock)
HH12	Hour (12-hour clock)
MI	Minutes
SS	Seconds
AM or PM	Meridian indicator (for 12-hour clock)

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:

COPY syntax	Example of valid input string
DATEFORMAT AS 'MM/DD/YYYY'	03/31/2003
DATEFORMAT AS 'MON DD, YYYY'	March 31, 2003
TIMEFORMAT AS 'MM.DD.YYYY HH:MI:SS'	03.31.2003 18:45:05 03.31.2003 18:45:05.123456

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=<your-access-key-id>; aws_secret_access_key=<your-
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. 188). In addition, the `'auto'` option recognizes the following formats:

Format	Example of valid input string
Julian	J2451187
BC	Jan-08-95 BC
YYYYMMDD HHMISS	19960108 040809

Format	Example of valid input string
YYMMDD HHMISS	960108 040809
YYYY.DDD	1996.008
YYYY.DDD YYYY-MM-DD HH:MI:SS.SSS	1996-01-08 04:05:06.789

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;

        test          |      timestamp      | date
----------------------+---------------------+------------
J2345678 04:05:06.789   1710-02-23 04:05:06 1710-02-23
```

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

Load FAVORITEMOVIES from an Amazon DynamoDB table

Note
These examples contain line breaks for readability. Do not include line breaks or spaces in your *aws_access_credentials* string.

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.

```
copy favoritemovies from 'dynamodb://ProductCatalog'
credentials 'aws_access_key_id=<your-access-key-id>;
aws_secret_access_key=<your-secret-access-key>'
readratio 50;
```

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 ('|').

```
copy listing
from 's3://mybucket/data/listings_pipe.txt'
credentials 'aws_access_key_id=<your-access-key-id>;
aws_secret_access_key=<your-secret-access-key>';
```

Load LISTING using temporary credentials

The following example uses the TOKEN option to specify temporary session credentials:

```
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`.

```
copy event
from 's3://mybucket/data/allevents_pipe.txt'
credentials 'aws_access_key_id=<your-access-key-id>;
aws_secret_access_key=<your-secret-access-key>'
removequotes
emptyasnull
blanksasnull
maxerror 5
delimiter '|'
timeformat 'YYYY-MM-DD HH:MI:SS';
```

Load VENUE from a fixed-width data file

```
copy venue
from 's3://mybucket/data/venue_fw.txt'
credentials 'aws_access_key_id=<your-access-key-id>;
aws_secret_access_key=<your-secret-access-key>'
fixedwidth 'venueid:3,venuename:25,venuecity:12,venuestate:2,venueseats:6';
```

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:

```
1  Toyota Park              Bridgeview  IL0
2  Columbus Crew Stadium    Columbus    OH0
3  RFK Stadium              Washington  DC0
4  CommunityAmerica BallparkKansas City KS0
5  Gillette Stadium         Foxborough  MA68756
```

Load CATEGORY from a CSV file

Suppose the file category_csv.txt contains the following text:

```
12,Shows,Musicals,Musical theatre
13,Shows,Plays,All "non-musical" theatre
14,Shows,Opera,All opera, light, and "rock" opera
15,Concerts,Classical,All symphony, concerto, and choir concerts
```

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.

```
12,Shows,Musicals,Musical theatre
13,Shows,Plays,"All ""non-musical"" theatre"
14,Shows,Opera,"All opera, light, and ""rock"" opera"
15,Concerts,Classical,"All symphony, concerto, and choir concerts"
```

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=<your-access-key-id>;
aws_secret_access_key=<your-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:

```
12,Shows,Musicals,Musical theatre
13,Shows,Plays,%All "non-musical" theatre%
14,Shows,Opera,%All opera, light, and "rock" opera%
15,Concerts,Classical,%All symphony, concerto, and choir concerts%
```

The following COPY command uses QUOTE AS to load `category_csv.txt`:

```
copy category
from 's3://mybucket/data/category_csv.txt '
credentials 'aws_access_key_id=<your-access-key-id>;
aws_secret_access_key=<your-secret-access-key>'
csv quote as '%';
```

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:

```
copy venue
from 's3://mybucket/data/venue.txt'
credentials 'aws_access_key_id=<your-access-key-id>;
```

```
aws_secret_access_key=<your-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:

```
copy time
from 's3://mybucket/data/timerows.gz'
credentials 'aws_access_key_id=<your-access-key-id>;
aws_secret_access_key=<your-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`.

```
copy timestamp1
from 's3://mybucket/data/time.txt'
credentials 'aws_access_key_id=<your-access-key-id>;
aws_secret_access_key=<your-secret-access-key>'
timeformat 'YYYY-MM-DD HH:MI:SS';
```

The result of this copy is as follows:

```
select * from timestamp1;
c1
----------------------------
2009-01-12 14:15:57.119568
(1 row)
```

Load LISTING from an Amazon S3 bucket

```
copy listing
from 's3://mybucket/data/listing/'
credentials 'aws_access_key_id=<your-access-key-id>;
aws_secret_access_key=<your-secret-access-key>';
```

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:

```
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:

```
1|Toyota Park|Bridgeview|IL|
2|Columbus Crew Stadium|Columbus|OH|
3|RFK Stadium|Washington|DC|
4|CommunityAmerica Ballpark|Kansas City|KS|
5|Gillette Stadium|Foxborough|MA|
6|New York Giants Stadium|East Rutherford|NJ|
7|BMO Field|Toronto|ON|
8|The Home Depot Center|Carson|CA|
9|Dick's Sporting Goods Park|Commerce City|CO|
10|Pizza Hut Park|Frisco|TX|
```

The following COPY statement will successfully load the table from the file and apply the DEFAULT value ('1000') to the omitted column:

```
copy venue_new(venueid, venuename, venuecity, venuestate)
from 's3://mybucket/data/venue_noseats.txt'
credentials 'aws_access_key_id=<your-access-key-id>;
aws_secret_access_key=<your-secret-access-key>'
delimiter '|';
```

Now view the loaded table:

```
select * from venue_new order by venueid;
venueid |        venuename          |    venuecity    | venuestate | venueseats
---------+---------------------------+-----------------+------------+--------
----
1 | Toyota Park               | Bridgeview      | IL         |       1000
2 | Columbus Crew Stadium     | Columbus        | OH         |       1000
3 | RFK Stadium               | Washington      | DC         |       1000
4 | CommunityAmerica Ballpark | Kansas City     | KS         |       1000
5 | Gillette Stadium          | Foxborough      | MA         |       1000
6 | New York Giants Stadium   | East Rutherford | NJ         |       1000
7 | BMO Field                 | Toronto         | ON         |       1000
8 | The Home Depot Center     | Carson          | CA         |       1000
9 | Dick's Sporting Goods Park | Commerce City   | CO         |       1000
10 | Pizza Hut Park            | Frisco          | TX         |       1000
(10 rows)
```

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:

```
1||Bridgeview|IL|
2||Columbus|OH|
3||Washington|DC|
4||Kansas City|KS|
5||Foxborough|MA|
6||East Rutherford|NJ|
7||Toronto|ON|
```

```
8||Carson|CA|
9||Commerce City|CO|
10||Frisco|TX|
```

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:

```
copy venue(venueid, venuecity, venuestate)
from 's3://mybucket/data/venue_pipe.txt'
credentials 'aws_access_key_id=<your-access-key-id>;
aws_secret_access_key=<your-secret-access-key>'
delimiter '|';
```

Now consider a variation of the VENUE table that uses an IDENTITY column:

```
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:

```
copy venue(venueid, venuename, venuecity, venuestate)
from 's3://mybucket/data/venue_pipe.txt'
credentials 'aws_access_key_id=<your-access-key-id>;
aws_secret_access_key=<your-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:

```
copy venue(venuename, venuecity, venuestate)
from 's3://mybucket/data/venue_pipe.txt'
credentials 'aws_access_key_id=<your-access-key-id>;
aws_secret_access_key=<your-secret-access-key>'
delimiter '|' explicit_ids;
```

This statement fails because it does not include an EXPLICIT_IDS option:

```
copy venue(venueid, venuename, venuecity, venuestate)
from 's3://mybucket/data/venue_pipe.txt'
credentials 'aws_access_key_id=<your-access-key-id>;
aws_secret_access_key=<your-secret-access-key>'
delimiter '|';
```

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=<your-access-key-id>;
aws_secret_access_key=<your-secret-access-key>'
delimiter '|' escape;

select * from redshiftinfo order by 1;
infoid |      tableinfo
-------+--------------------
1      | public|event|dwuser
2      | public|sales|dwuser
(2 rows)
```

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 (`' '` 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>
<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 copying 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=<your-access-key-id>;aws_secret_access_key=<your-
secret-access-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
```

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.

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 64 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.

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:

```
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.

> **Note**
> The list of schemas in the search_path (p. 527) 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:

```
create schema us_sales authorization dwuser;
```

To view the new schema, query the PG_NAMESPACE catalog table.

```
select nspname as schema, usename as owner
from pg_namespace, pg_user
where pg_namespace.nspowner = pg_user.usesysid
and pg_user.usename ='dwuser';

   name   |  owner
----------+----------
 us_sales | dwuser
(1 row)
```

CREATE TABLE

Topics

Creates a new table in the current database. The owner of this table is the user that issues the CREATE TABLE command.

Synopsis

```
CREATE [ [LOCAL ] { TEMPORARY | TEMP } ]
TABLE table_name
(
{column_name data_type
[ DEFAULT default_expr ]
[ IDENTITY ( seed, step) ]
[ column_constraint ]
[ ENCODE encoding ]
[ DISTKEY ]
[ SORTKEY ]
| table_constraint
| LIKE parent_table
[ { INCLUDING | EXCLUDING } DEFAULTS ] } [, ... ]
)
[ DISTSTYLE { EVEN | KEY } ]
[ DISTKEY ( column_name ) ]
[ SORTKEY ( column_name [, ...] ) ]

where column_constraint is:

[ CONSTRAINT constraint_name ]
```

```
{ NOT NULL |
NULL |
UNIQUE |
PRIMARY KEY |
REFERENCES reftable
[ ( refcolumn ) ]}

and table_constraint is:

[ CONSTRAINT constraint_name ]
{ UNIQUE ( column_name [, ... ] )  |
PRIMARY KEY ( column_name [, ... ] ) |
FOREIGN KEY (column_name [, ... ] )
REFERENCES  reftable
[ ( refcolumn ) ]}
```

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 only visible within the current session. The table is automatically dropped at the end of the session in which it is created. Amazon Redshift allows a user to create a temporary table with 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. See search_path (p. 527) for more information about schemas and precedence.

> **Note**
> By default, users are granted privilege 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 privilege from the PUBLIC group and then explicitly grant the privilege to create temporary tables 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:

```
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 to be created in the new table. 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.

> **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. See CREATE TABLE usage notes (p. 204).

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. 119) 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. 36) are supported:

- BYTEDICT
- DELTA
- DELTA32K

- MOSTLY8
- MOSTLY16
- MOSTLY32
- RAW (no compression, the default setting)
- RUNLENGTH
- TEXT255
- TEXT32K

DISTSTYLE

Attribute that defines the type of data distribution for the whole table:

- KEY: The data is distributed by the values in the DISTKEY column. This approach ensures that repeating values in the DISTKEY column are stored together in the same slice on the same node. This type of distribution is efficient for distributed table joins. If you specify DISTSTYLE KEY, you must name a DISTKEY column, either here for the table or as part of the column definition.
- 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 is distributed to each node.

DISTKEY

Only one column in a table can be its distribution key. If you do not specify a DISTKEY column or a DISTSTYLE option, DISTSTYLE EVEN distribution is used by default. If you declare a column as the DISTKEY column, DISTSTYLE must be set to KEY or not set at all. 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. See Choosing a data distribution method (p. 45).

SORTKEY (*column_name* [, ...])

When data is loaded into the table, the data is sorted by one or more columns 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.

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 only copied if INCLUDING DEFAULTS is specified. The default is to exclude default expressions, resulting in all columns of the new table having null defaults.

Tables created with the LIKE option do not inherit primary and foreign key constraints. Distribution and sort properties and NULL properties are inherited by LIKE tables but cannot be explicitly set in the CREATE TABLE statement.

CONSTRAINT *constraint_name*

Name for a column or table constraint.

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 and do not accept NULLs.

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 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 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*)]
These clauses specify 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.

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 64K (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 any NOT NULL columns, the starting position of the last column cannot be greater than 64K. For example, the following table *can* be loaded and queried:

```
create table t (c1 varchar(30000) not null,
c2 char(30000) not null, c3 varchar (10000) not null);

insert into t values(1,1,1);

select trim(c1), trim(c2), trim(c3) from t;
btrim | btrim | btrim
------+-------+-------
1     | 1     | 1
(1 row)
```

  However, the following table, in which the starting position of the third column is greater than 64K, returns an error if you attempt to insert a row:

```
create table t (c1 varchar(35000) not null,
c2 char(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

Create the 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 multi-column sort key. Primary key and foreign key constraints are also defined for the table.

```
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

```
select "column", type, encoding, distkey, sortkey
from pg_table_def where tablename = 'sales';

   column   |            type             | encoding | distkey | sortkey
------------+-----------------------------+----------+---------+---------
 salesid    | integer                     | none     | f       |       0
 listid     | integer                     | none     | t       |       1
 sellerid   | integer                     | none     | f       |       2
 buyerid    | integer                     | none     | f       |       0
 eventid    | integer                     | mostly16 | f       |       0
 dateid     | smallint                    | none     | f       |       0
 qtysold    | smallint                    | mostly8  | f       |       0
 pricepaid  | numeric(8,2)                | delta32k | f       |       0
 commission | numeric(8,2)                | delta32k | f       |       0
 saletime   | timestamp without time zone | none     | f       |       0
(10 rows)
```

Create a table with default even distribution

Create a table called MYEVENT with three columns:

```
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.

```
select "column", type, encoding, distkey, sortkey
from pg_table_def where tablename = 'myevent';

  column   |          type          | encoding | distkey | sortkey
-----------+------------------------+----------+---------+---------
 eventid   | integer                | none     | f       |       0
 eventname | character varying(200) | none     | f       |       0
 eventcity | character varying(30)  | none     | f       |       0
(3 rows)
```

Create a temporary table that is LIKE another table

Create a temporary table called TEMPEVENT, which inherits its columns from the EVENT table.

```
create temp table tempevent(like event);
```

This table also inherits the DISTKEY and SORTKEY attributes of its parent table:

```
select "column", type, encoding, distkey, sortkey
 from pg_table_def where tablename = 'tempevent';

  column   |            type             | encoding | distkey | sortkey
-----------+-----------------------------+----------+---------+---------
 eventid   | integer                     | none     | t       |       1
 venueid   | smallint                    | none     | f       |       0
 catid     | smallint                    | none     | f       |       0
 dateid    | smallint                    | none     | f       |       0
 eventname | character varying(200)      | none     | f       |       0
 starttime | timestamp without time zone | none     | f       |       0
(6 rows)
```

Create a table with an IDENTITY column

Create 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

Create 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 set to KEY or not set. By default, the table has no sort key and 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 set to KEY or not set.

```
create table t2(col1 int distkey sortkey, col2 int);
```

Result

```
select "column", type, encoding, distkey, sortkey
from pg_table_def where tablename = 't2';

column |  type   | encoding | distkey | sortkey
-------+---------+----------+---------+---------
col1   | integer | none     | t       | 1
col2   | integer | none     | f       | 0
```

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 EVEN:

```
create table t3(col1 int, col2 int sortkey) diststyle even;
```

Result

```
select "column", type, encoding, distkey, sortkey
from pg_table_def where tablename = 't3';

Column |  Type   | Encoding | DistKey | SortKey
-------+---------+----------+---------+--------
col1   | integer | none     | f       | 0
col2   | integer | none     | f       | 1
```

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.

```
create table t4(col1 int, col2 int) diststyle even;
```

Result

```
select "column", type, encoding, distkey, sortkey
from pg_table_def where tablename = 't4';

column |  type   | encoding | distkey | sortkey
--------+---------+----------+---------+--------
col1   | integer | none     | f       | 0
col2   | integer | none     | f       | 0
```

CREATE TABLE AS

Topics

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 { KEY | EVEN }
| 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

Attribute that defines the type of data distribution for the whole table:

- KEY: The data is distributed by the values in the DISTKEY column (which must be specified if DISTSTYLE KEY is specified). This approach ensures that repeating values in the DISTKEY column are stored together in the same slice on the same node. This type of distribution is efficient for both distributed table joins and collocated joins in which the DISTKEY column is the joining column for both tables.
- EVEN: The data in the table is spread evenly across the nodes and slices in a round-robin distribution. Roughly the same number of rows is distributed to each node. If the DISTSTYLE option is set to EVEN, no default sort key is defined.

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';

column   |          type          | encoding | distkey | sortkey
---------+------------------------+----------+---------+---------
eventid  | integer                | none     | t       | 1
venueid  | smallint               | none     | f       | 0
...
```

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';

column   |          type          | encoding | distkey | sortkey
---------+------------------------+----------+---------+-------
eventid   | integer                | none     | t       | 1
venueid   | smallint               | none     | f       | 0
dateid    | smallint               | none     | f       | 2
eventname | character varying(200)| none     | f       | 0
```

You could create the exact 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
diststyle even
as
select eventid, venueid, dateid, eventname
from event;
```

The table cannot 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';

column    |          type          | encoding | distkey | sortkey
----------+------------------------+----------+---------+---------
eventid   | integer                | none     | f       | 0
venueid   | smallint               | none     | f       | 0
dateid    | smallint               | none     | f       | 0
eventname | character varying(200) | none     | f       | 0
```

The following statement applies even distribution and defines a sort key:

```
create table eventdistevensort diststyle even sortkey (venueid)
as select eventid, venueid, dateid, eventname from event;
```

The resulting table has a sort key but no distribution key.

```
select "column", type, encoding, distkey, sortkey
from pg_table_def where tablename = 'eventdistevensort';

column    |          type          | encoding | distkey | sortkey
----------+------------------------+----------+---------+-------
eventid   | integer                | none     | f       | 0
venueid   | smallint               | none     | f       | 1
dateid    | smallint               | none     | f       | 0
eventname | character varying(200) | none     | f       | 0
```

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     | t       | 0
catid     | smallint                    | none     | f       | 0
dateid    | smallint                    | none     | f       | 0
eventname | character varying(200)      | none     | f       | 0
starttime | timestamp without time zone | none     | f       | 0
```

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.

CREATEDB | NOCREATEDB
The CREATEDB option allows the new user account to create databases. NOCREATEDB is the default.

CREATEUSER | NOCREATEUSER
The CREATEUSER option gives the new user the privilege to create accounts. NOCREATEUSER is the default.

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. If this option is not specified, 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, 2010:

```
create user danny with password 'abcD1234' valid until '2010-06-10';
```

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.

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.

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. 241).

See Also

EXECUTE (p. 227), PREPARE (p. 241)

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. 217).

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. 231), CLOSE (p. 176).

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 to minimize round trips.

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. 217).

Amazon Redshift cursors are supported with the following limitations:

- Cursors are only supported on multi-node clusters. Cursors are not supported on single-node clusters.
- Only one cursor at time can be open per session.
- Cursors must be used within a transaction (BEGIN … END).
- The number of concurrent cursors per cluster and the maximum result set per cursor are constrained based on the node type.

  For more information, see Cursor constraints (p. 217).

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 the following constraints on the number of concurrent cursors per cluster and the size of a cursor result set, based on the cluster's node type:

Node type	Concurrent cursors	Maximum result set
XL	5	500 GB
8XL	15	1.3 TB

To view the active cursor configuration for a cluster, query the STV_CURSOR_CONFIGURATION (p. 486) system table as a superuser. To view the state of active cursors, query the STV_ACTIVE_CURSORS (p. 482) 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 (p. 282) 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. 84)Unloading Data.
- 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. 102).

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:

```
-- 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 |   92.00000000 |       3
 Lollapalooza | 2008-11-15 15:00:00 |  222.00000000 |       2
 Lollapalooza | 2008-04-17 15:00:00 |  239.00000000 |       3
 Lollapalooza | 2008-04-17 15:00:00 |  239.00000000 |       4
 Lollapalooza | 2008-04-17 15:00:00 |  239.00000000 |       1
(5 rows)

-- Fetch the next row:

fetch next from lollapalooza;

  eventname   |      starttime      | costperticket | qtysold
--------------+---------------------+---------------+---------
 Lollapalooza | 2008-10-06 14:00:00 |  114.00000000 |       2

-- Close the cursor and end the transaction:

close lollapalooza;
commit;
```

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.

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. 281).

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:

```
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:

```
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:

```
delete from category;
```

Delete rows with CATID values between 0 and 9 from the CATEGORY table:

```
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:

```
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:

```
delete from category
using event
where event.catid=category.catid and category.catid=9;
```

```
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. This command is not reversible.

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:

```
 tbloid | 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.

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

To drop a user who currently owns a database, first drop the database or change its ownership to another user before executing this command.

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. 178) 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. 227) command.

For examples and more information on the creation and usage of prepared statements, see PREPARE (p. 241).

See Also

DEALLOCATE (p. 216), PREPARE (p. 241)

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.

EXPLAIN Operators	Query Execution Steps	Description
SCAN:		
Sequential Scan	scan	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.
JOINS: Amazon Redshift uses different join operators based on the physical design of the tables being joined, the location of the data required for the join, and specific attributes of the query itself. Subquery Scan -- Subquery scan and append are used to run UNION queries.		
Nested Loop	nloop	Least optimal join; mainly used for cross-joins (Cartesian products; without a join condition) and some inequality joins.
Hash Join	hjoin	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.)
Merge Join	mjoin	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.

EXPLAIN Operators	Query Execution Steps	Description
AGGREGATION: Operators and steps used for queries that involve aggregate functions and GROUP BY operations.		
Aggregate	aggr	Operator/step for scalar aggregate functions.
HashAggregate	aggr	Operator/step for grouped aggregate functions. Can operate from disk by virtue of hash table spilling to disk.
GroupAggregate	aggr	Operator sometimes chosen for grouped aggregate queries if the Amazon Redshift configuration setting for force_hash_grouping setting is off.
SORT: Operators and steps used when queries have to sort or merge result sets.		
Sort	sort	Sort performs the sorting specified by the ORDER BY clause as well as other operations such as UNIONs and joins. Can operate from disk.
Merge	merge	Produces final sorted results of a query based on intermediate sorted results derived from operations performed in parallel.
EXCEPT, INTERCEPT, and UNION operations:		
SetOp Except [Distinct]	hjoin	Used for EXCEPT queries. Can operate from disk based on virtue of fact that input hash can be disk-based.
Hash Intersect [Distinct]	hjoin	Used for INTERSECT queries. Can operate from disk based on virtue of fact that input hash can be disk-based.
Append [All |Distinct]	save	Append used with Subquery Scan to implement UNION and UNION ALL queries. Can operate from disk based on virtue of "save".
Miscellaneous/Other:		
Hash	hash	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.)
Limit	limit	Evaluates the LIMIT clause.
Materialize	save	Materialize rows for input to nested loop joins and some merge joins. Can operate from disk.
--	parse	Used to parse textual input data during a load.
--	project	Used to rearrange columns and compute expressions, that is, project data.
Result	--	Run scalar functions that do not involve any table access.

EXPLAIN Operators	Query Execution Steps	Description
--	return	Return rows to the leader or client.
Subplan	--	Used for certain subqueries.
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:

```
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:

```
explain verbose
select eventid, eventname, event.venueid, venuename
from event, venue
where event.venueid = venue.venueid;
```

```
                                QUERY PLAN
------------------------------------------------------------------------------
{HASHJOIN
:startup_cost 2.52
:total_cost 58653620.93
:plan_rows 8712
:plan_width 43
:best_pathkeys <>
:dist_info DS_DIST_OUTER
:dist_info.dist_keys (
TARGETENTRY
{
VAR
:varno 2
:varattno 1
...

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)
(519 rows)
```

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;

QUERY PLAN
------------------------------------------------------------
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. 216).

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. 216), CLOSE (p. 176).

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.)

FORWARD [*count* | ALL]
Fetches the next *count* rows, or all remaining rows. `FORWARD 0` fetches the current row.

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:

```
-- 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 |   92.00000000 |       3
 Lollapalooza | 2008-11-15 15:00:00 |  222.00000000 |       2
 Lollapalooza | 2008-04-17 15:00:00 |  239.00000000 |       3
 Lollapalooza | 2008-04-17 15:00:00 |  239.00000000 |       4
 Lollapalooza | 2008-04-17 15:00:00 |  239.00000000 |       1
(5 rows)

-- Fetch the next row:

fetch next from lollapalooza;
```

```
  eventname   |      starttime      | costperticket | qtysold
--------------+---------------------+---------------+---------
 Lollapalooza | 2008-10-06 14:00:00 |  114.00000000 |       2

-- Close the cursor and end the transaction:

close lollapalooza;
commit;
```

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 database object, such as a table, view, function, or schema. To revoke privileges from a database object, use the REVOKE (p. 243) 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 privileges on objects within a specific schema, which makes these objects accessible to users. Specific actions on these objects must be granted separately (such as EXECUTE privilege on functions). 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 access to the underlying tables. 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 `bobr`:

```
grant select on table sales to bobr;
```

The following example grants all schema privileges on the schema QA_TICKIT to the user group QA_USERS:

```
grant all on schema qa_tickit to group qa_users;
```

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 account has been granted all privileges on VIEW_DATE.

```
select current_user;
current_user
--------------
dwuser
(1 row)

create user view_user;

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
--------------
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

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. 179) 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. 209) to improve performance. For more information about using the COPY command to load tables, see Loading Data (p. 54).

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 may 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.

Note
You cannot use a multiple-row INSERT VALUES statement on a table with an IDENTITY column.

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:

```
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. 179) 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. 209) to improve performance. For more information about using the COPY command to load tables, see Loading Data (p. 54).

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:

```
 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
(11 rows)
```

Create a CATEGORY_STAGE table with a similar schema to the CATEGORY table but define default values for the columns:

```
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.

```
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:

```
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:

```
insert into category_stage values
(13, 'Concerts', 'Other', default);
```

Run the following query to return the inserted rows:

```
select * from category_stage
where catid in(12,13) order by 1;

catid | catgroup | catname |             catdesc
-------+----------+---------+---------------------------------
12 | Concerts | Comedy  | All stand-up comedy performances
13 | Concerts | Other   | General
(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.

```
insert into category_stage values
(14, default, default, default),
(15, default, default, default);

select * from category_stage where catid in(14,15) order by 1;
```

```
catid | catgroup | catname | catdesc
-------+----------+---------+---------
14 | General  | General | General
15 | General  | General | General
(2 rows)
```

The next example inserts three rows with various combinations of specific and default values:

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

select * from category_stage where catid in(0,20,21) order by 1;
catid | catgroup | catname | catdesc
-------+----------+---------+---------
0 | General  | General | General
20 | General  | Country | General
21 | Concerts | Rock    | General
(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:

```
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.

You can insert only one row at a time into a table with an IDENTITY column. If you try to insert multiple rows into the last three columns of the table with an INSERT VALUES statement, Amazon Redshift returns an error:

```
insert into category_ident(catgroup,catname,catdesc) values
(default,default,default),
(default,default,default);
ERROR:  multi-row VALUES not supported for a table with IDENTITY column

insert into category_ident(catgroup,catname,catdesc) values
(default,default,default);
```

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. 79).

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. 471) 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. 227) 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. 227) command.

For more information on query planning and the statistics collected by Amazon Redshift for query optimization, see the ANALYZE (p. 171) 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. 216), EXECUTE (p. 227)

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. 276) command. To display the current value of a parameter, use the SHOW (p. 280) command.

Synopsis

```
RESET { parameter_name | ALL }
```

Parameters

parameter_name
Name of the parameter to reset. See Modifying the server configuration (p. 525) 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. 233) 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

```
REVOKE [ GRANT OPTION FOR ]
{ { SELECT | INSERT | UPDATE | DELETE | RULE | 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. 161) 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:

```
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:

```
select * from movie_gross;
```

The command output shows that both rows successfully inserted:

```
name           |  gross
-----------------------+----------
Raiders of the Lost Ark | 23400000
Star Wars               | 10000000
(2 rows)
```

This command now rolls back the data changes to where the transaction began:

```
rollback;
```

Selecting data from the table now shows an empty table:

```
select * from movie_gross;

name | gross
------+-------
(0 rows)
```

SELECT

Topics

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. 253).

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. 246).

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:

```
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.

```
with venuecopy as (select * from venue)
select * from venuecopy order by 1 limit 10;
```

```
 venueid |        venuename         |    venuecity    | venuestate | venueseats
---------+--------------------------+-----------------+------------+--------
----
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
6 | New York Giants Stadium   | East Rutherford | NJ         |      80242
7 | BMO Field                 | Toronto         | ON         |          0
8 | The Home Depot Center     | Carson          | CA         |          0
9 | Dick's Sporting Goods Park | Commerce City   | CO         |          0
v     10 | Pizza Hut Park            | Frisco          | TX         |
```

```
  0
(10 rows)
```

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.

```
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;
```

```
        venuename       |   venuecity   | venuestate | venue_qty | venue_sales
------------------------+---------------+------------+-----------+------------
-
August Wilson Theatre   | New York City | NY         |      3187 |  1032156.00
Biltmore Theatre        | New York City | NY         |      2629 |   828981.00
Charles Playhouse       | Boston        | MA         |      2502 |   857031.00
Ethel Barrymore Theatre | New York City | NY         |      2828 |   891172.00
Eugene O'Neill Theatre  | New York City | NY         |      2488 |   828950.00
Greek Theatre           | Los Angeles   | CA         |      2445 |   838918.00
Helen Hayes Theatre     | New York City | NY         |      2948 |   978765.00
Hilton Theatre          | New York City | NY         |      2999 |   885686.00
Imperial Theatre        | New York City | NY         |      2702 |   877993.00
Lunt-Fontanne Theatre   | New York City | NY         |      3326 |  1115182.00
Majestic Theatre        | New York City | NY         |      2549 |   894275.00
Nederlander Theatre     | New York City | NY         |      2934 |   936312.00
Pasadena Playhouse      | Pasadena      | CA         |      2739 |   820435.00
Winter Garden Theatre   | New York City | NY         |      2838 |   939257.00
(14 rows)
```

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:

```
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;

caldate   | daysales | dec25sales
----------+----------+------------
2008-12-25 | 70402.00 |   70402.00
2008-12-31 | 12678.00 |   70402.00
(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.

```
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

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:

```
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:

```
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;
q |  sum
---+--------
8 |    368
...
```

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;

catgroup
----------
Concerts
Shows
Sports
(3 rows)
```

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. 247).

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:

```
select * from sales s, listing l
where s.listid=l.listid
```

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:

```
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:

```
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. 247).

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

```
[ 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:

```
sales.listid=listing.listid
sales.listid<>listing.listid
```

The following examples are valid conditions on columns in tables:

```
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.

```
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.

```
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;

eventname    |      starttime      |   costperticket   | qtysold
---------------+---------------------+-------------------+---------
Hannah Montana | 2008-06-07 14:00:00 |     1706.00000000 |       2
Hannah Montana | 2008-05-01 19:00:00 |     1658.00000000 |       2
Hannah Montana | 2008-06-07 14:00:00 |     1479.00000000 |       1
Hannah Montana | 2008-06-07 14:00:00 |     1479.00000000 |       3
Hannah Montana | 2008-06-07 14:00:00 |     1163.00000000 |       1
Hannah Montana | 2008-06-07 14:00:00 |     1163.00000000 |       2
Hannah Montana | 2008-06-07 14:00:00 |     1163.00000000 |       4
Hannah Montana | 2008-05-01 19:00:00 |      497.00000000 |       1
Hannah Montana | 2008-05-01 19:00:00 |      497.00000000 |       2
Hannah Montana | 2008-05-01 19:00:00 |      497.00000000 |       4
(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:

```
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:

  ```
  event.eventid*10(+)=category.catid
  ```

  However, the following join condition is valid:

  ```
  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:

  ```
  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:

```
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.

```
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:

```
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:

```
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.

```
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;

listid | price  |  comm
--------+--------+--------
```

```
1 | 728.00 | 109.20
2 |        |
3 |        |
4 |  76.00 |  11.40
5 | 525.00 |  78.75
(5 rows)
```

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:

```
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:

```
select catname, catgroup, eventid
from category, event
where category.catid=event.catid(+) and eventid(+)=796;

catname | catgroup | eventid
-----------+----------+---------
Classical | Concerts |
Jazz | Concerts |
MLB | Sports   |
MLS | Sports   |
Musicals | Shows    | 796
NBA | Sports   |
NFL | Sports   |
NHL | Sports   |
Opera | Shows    |
Plays | Shows    |
Pop | Concerts |
(11 rows)
```

The equivalent query using FROM clause syntax is as follows:

```
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
106590 |      76 |   20.00 |      1
124683 |     393 |   20.00 |      1
103037 |     403 |   20.00 |      1
```

```
147685 |     429 |   20.00 |      1
(5 rows)
```

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
Legally Blonde   |  804583.00
(6 rows)
```

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: `sum(qtysold)`. Events that did not sell more than 2,000 tickets are eliminated from the final result.

```
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

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

```
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. 137).

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.

```
select listid, sellerid, eventid from listing
union select listid, sellerid, eventid from sales
order by listid, sellerid, eventid desc limit 5;

listid | sellerid | eventid
--------+----------+---------
1 |    36861 |    7872
2 |    16002 |    4806
3 |    21461 |    4256
4 |     8117 |    4337
5 |     1616 |    8647
(5 rows)
```

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.

```
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;

listid | lastname | firstname | username |   price   | buyorsell
--------+----------+-----------+----------+-----------+-----------
209658 | Lamb     | Colette   | VOR15LYI |  10000.00 | B
209658 | West     | Kato      | ELU81XAA |  10000.00 | S
212395 | Greer    | Harlan    | GXO71KOC |  12624.00 | S
212395 | Perry    | Cora      | YWR73YNZ |  12624.00 | B
215156 | Banks    | Patrick   | ZNQ69CLT |  10000.00 | S
215156 | Hayden   | Malachi   | BBG56AKU |  10000.00 | B
(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').

```
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;

eventid | listid | salesrow
---------+--------+----------
7787 |    500 | No
7787 |    500 | Yes
7787 |    500 | Yes
7787 |    500 | Yes
6473 |    501 | No
5108 |    502 | No
(6 rows)
```

If you run the same query without the ALL keyword, the result retains only one of the sales transactions.

```
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;

eventid | listid | salesrow
---------+--------+----------
7787 |    500 | No
7787 |    500 | Yes
6473 |    501 | No
5108 |    502 | No
(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').

```
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;

eventid | listid | salesrow
---------+--------+----------
7787 |    500 | No
7787 |    500 | Yes
7787 |    500 | Yes
7787 |    500 | Yes
6473 |    501 | No
5108 |    502 | No
(6 rows)
```

If you run the same query without the ALL keyword, the result retains only one of the sales transactions.

```
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;

eventid | listid | salesrow
---------+--------+----------
7787 |    500 | No
7787 |    500 | Yes
6473 |    501 | No
5108 |    502 | No
(4 rows)
```

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:

```
235494 |    23875 |    8771
```

This is the only row in the limited result of 5 rows that was found in both tables.

```
select listid, sellerid, eventid from listing
intersect
select listid, sellerid, eventid from sales
order by listid desc, sellerid, eventid
```

```
limit 5;

listid | sellerid | eventid
--------+----------+---------
235494 |    23875 |    8771
235482 |     1067 |    2667
235479 |     1589 |    7303
235476 |    15550 |     793
235475 |    22306 |    7848
(5 rows)
```

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.

```
select distinct eventname from event, sales, venue
where event.eventid=sales.eventid and event.venueid=venue.venueid
and date_part(month,starttime)=3 and venuecity='Los Angeles'
intersect
select distinct eventname from event, sales, venue
where event.eventid=sales.eventid and event.venueid=venue.venueid
and date_part(month,starttime)=3 and venuecity='New York City'
order by eventname asc;

eventname
----------------------------
A Streetcar Named Desire
Dirty Dancing
Electra
Hairspray
Mary Poppins
November
Oliver!
Return To Forever
Rhinoceros
South Pacific
The 39 Steps
The Bacchae
The Caucasian Chalk Circle
The Country Girl
Wicked
Woyzeck
(16 rows)
```

Example EXCEPT query

The CATEGORY table in the TICKIT database contains the following 11 rows:

```
 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
(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
12 | Concerts | Comedy    | All stand up comedy performances
(12 rows)
```

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
(1 row)
```

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
(1 row)
```

If you reverse the order of the SELECT expressions, the query returns no rows.

ORDER BY clause

Topics

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 2 billion (`2000000000`).

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. LIMIT ALL returns all rows so the behavior is equivalent to not including a LIMIT clause in the query.

OFFSET *start*

Option that controls the number of rows in the result set by specifying a starting row number. The rows before the starting number are skipped. When used with the LIMIT option, OFFSET returns the number of rows defined by the LIMIT number, but skips the first *n* rows that the query would normally return in that set.

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;

catid | catgroup |  catname  |                 catdesc
-------+----------+-----------+--------------------------------------
10 | Concerts | Jazz      | All jazz singers and bands
9 | Concerts | Pop       | All rock and pop music concerts
11 | Concerts | Classical | All symphony, concerto, and choir conce
6 | Shows    | Musicals  | Musical theatre
7 | Shows    | Plays     | All non-musical theatre
8 | Shows    | Opera     | All opera and light opera
5 | Sports   | MLS       | Major League Soccer
1 | Sports   | MLB       | Major League Baseball
2 | Sports   | NHL       | National Hockey League
3 | Sports   | NFL       | National Football League
4 | Sports   | NBA       | National Basketball Association
(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;

salesid | qtysold | pricepaid | commission |      saletime
---------+---------+-----------+------------+---------------------
15401 |       8 |    272.00 |      40.80 | 2008-03-18 06:54:56
61683 |       8 |    296.00 |      44.40 | 2008-11-26 04:00:23
90528 |       8 |    328.00 |      49.20 | 2008-06-11 02:38:09
74549 |       8 |    336.00 |      50.40 | 2008-01-19 12:01:21
130232 |       8 |    352.00 |      52.80 | 2008-05-02 05:52:31
55243 |       8 |    384.00 |      57.60 | 2008-07-12 02:19:53
16004 |       8 |    440.00 |      66.00 | 2008-11-04 07:22:31
489 |       8 |    496.00 |      74.40 | 2008-08-03 05:48:55
4197 |       8 |    512.00 |      76.80 | 2008-03-23 11:35:33
16929 |       8 |    568.00 |      85.20 | 2008-12-19 02:59:33
(10 rows)
```

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 c.catid = 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 c.catid = e.catid and e.eventid = s.eventid
and s.listid = l.listid
group by catgroup) as b(catgroup2, unsold)

on a.catgroup1 = b.catgroup2
order by 1;

catgroup1 |  sold  | unsold
-----------+--------+--------
Concerts  | 195444 |1067199
Shows     | 149905 | 817736
(2 rows)
```

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. 272) 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.

```
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;

qtr  |  qtrsales   |   q1sales
-------+-------------+-------------
2     | 30560050.00 | 24742065.00
3     | 31170237.00 | 24742065.00
(2 rows)
```

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.

```
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;

firstname | lastname  |      city      | maxsold
-----------+-----------+----------------+---------
Noah       | Guerrero | Worcester      |       8
Isadora    | Moss     | Winooski       |       8
Kieran     | Harrison | Westminster    |       8
Heidi      | Davis    | Warwick        |       8
Sara       | Anthony  | Waco           |       8
Bree       | Buck     | Valdez         |       8
Evangeline | Sampson  | Trenton        |       8
Kendall    | Keith    | Stillwater     |       8
```

```
Bertha     | Bishop   | Stevens Point  |       8
Patricia   | Anderson | South Portland |       8
(10 rows)
```

WITH clause subqueries

See WITH clause (p. 247).

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 `where s.listid=l.listid`. For each row that the outer query produces, the subquery is executed to qualify or disqualify the row.

```
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;

salesid | listid |   sum
---------+--------+----------
27 |     28 | 111.00
81 |    103 | 181.00
142 |    149 | 240.00
146 |    152 | 231.00
194 |    210 | 144.00
(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

```
[ 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. 246).

Examples

Select all of the rows from the EVENT table and create a NEWEVENT table:

```
select * into newevent from event;
```

Select the result of an aggregate query into a temporary table called PROFITS:

```
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. 242) command to return a parameter to its default value. See Modifying the server configuration (p. 525) for more information about parameters.

Synopsis

```
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. 387) 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. 525) 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:

```
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.

```
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. 102)

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
------
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)
```

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

Sets the transaction characteristics for all of the transactions in a session.

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.

Synopsis

```
SHOW { parameter_name | ALL }
```

Parameters

parameter_name
Displays the current value of the specified parameter.

ALL
Displays the current values of all of the parameters.

Examples

The following example displays the value for the query_group parameter:

```
show query_group;

query_group

unset
(1 row)
```

The following example displays a list of all parameters and their values:

```
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. 174).

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

```
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 a set of files on Amazon S3.

Synopsis

```
UNLOAD ('select_statement')
TO 's3_path'
[ WITH ] CREDENTIALS [AS] 'aws_access_credentials'
[ option [ ... ] ]

where option is

{ DELIMITER [ AS ] 'delimiter_char'
| FIXEDWIDTH [ AS ] 'fixedwidth_spec' }
| ENCRYPTED
| GZIP
| ADDQUOTES
| NULL [ AS ] 'null_string'
| ESCAPE
| ALLOWOVERWRITE
```

Parameters

('*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:

```
('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:

```
('select * from venue where venuestate=\'NV\'')
```

TO '*s3_path*'
The path prefix on Amazon S3 where Amazon Redshift will write the output files. UNLOAD writes one or more files per slice. File names are written in the format `path_prefix<slice-number>_part_<file-number>`.

Important
The Amazon S3 bucket where Amazon Redshift will write the output files must be created in the same region as your cluster.

WITH
This keyword is optional.

CREDENTIALS [AS] '*aws_access_credentials*'
The AWS account access credentials for the Amazon S3 bucket. If these credentials correspond to an IAM user, that IAM user must have 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. 188) in Usage notes for the COPY command.

Note
These examples contain line breaks for readability. Do not include line breaks or spaces in your *aws_access_credentials* string.

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=<your-access-key-id>;
aws_secret_access_key=<your-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=<your-access-key-id>;
aws_secret_access_key=<your-secret-access-key>'master_symmetric_key=<mas
ter_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.

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. 86). To unload to encrypted gzip-compressed files, add the GZIP option.

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.

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. The default string is `\N` (backslash-N). 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: `\`
- 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.

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')
to 's3://mybucket/venue_pipe_'
credentials 'aws_access_key_id=<your-access-key-id>;
aws_secret_access_key=<your-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)')
to 's3://mybucket/venue_pipe_'
credentials 'aws_access_key_id=<your-access-key-id>;
aws_secret_access_key=<your-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.

This example unloads the VENUE table and writes the data to `s3://mybucket/`:

```
unload ('select * from venue')
to 's3://mybucket/'
credentials 'aws_access_key_id=<your-access-key-id>;
aws_secret_access_key=<your-secret-access-key>';
```

Assuming a two-node cluster with two slices per node, the previous example creates these files in `mybucket`:

```
0000_part_00
0001_part_00
0002_part_00
0003_part_00
```

To better differentiate the output files, you can include a prefix in the location. This 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=<your-access-key-id>;
aws_secret_access_key=<your-secret-access-key>';
```

The result is these four files, again assuming four slices.

```
venue_pipe_0000_part_00
venue_pipe_0001_part_00
venue_pipe_0002_part_00
venue_pipe_0003_part_00
```

Load VENUE from unload files

To load tables 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=<your-access-key-id>;
aws_secret_access_key=<your-secret-access-key>';
```

Unload VENUE to encrypted files

The following example unloads the VENUE table to a set of encrypted files. For more information, see Unloading encrypted data files (p. 86)

```
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=EXAMPLEMAS
TERKEYtkbjk/OpCwtYSx/M4/t7DMCDIK722' 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.

```
create table loadvenue (like venue);
copy loadvenue from 's3://mybucket/venue_pipe_'
credentials 'aws_access_key_id=<your-access-key-id>;
aws_secret_access_key=<your-secret-access-key>master_symmetric_key=EXAMPLEMAS
TERKEYtkbjk/OpCwtYSx/M4/t7DMCDIK722' encrypted;
```

Unload VENUE data to a tab-delimited file

```
unload ('select venueid, venuename, venueseats from venue')
to 's3://mybucket/venue_tab_'
credentials 'aws_access_key_id=<your-access-key-id>;
aws_secret_access_key=<your-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. 188) in the usage notes for the COPY command.

The following example unloads the LISTING table using temporary credentials:

```
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 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.

Unload VENUE to a fixed-width data file

```
unload ('select * from venue')
to 's3://mybucket/venue_fw_'
credentials 'aws_access_key_id=<your-access-key-id>;
aws_secret_access_key=<your-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 BallparkKansas 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=<your-access-key-id>;
aws_secret_access_key=<your-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=<your-access-key-id>;
aws_secret_access_key=<your-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.

```
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=<your-access-key-id>;
aws_secret_access_key=<your-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
...
```

NULL AS example

The NULL AS option in the following example replaces actual null values in the VENUESEATS column with a `null` character string.

```
unload ('select * from venue')
's3://mybucket/allvenues/'
credentials 'aws_access_key_id=<your-access-key-id>;
aws_secret_access_key=<your-secret-access-key>'
null as 'null';
```

For example, the output rows look like this:

```
2|Columbus Crew Stadium|Columbus|OH|0
22|Quicken Loans Arena|Cleveland|OH|0
75|Lucas Oil Stadium|Indianapolis|IN|63000
82|Lincoln Financial Field|Philadelphia|PA|68532
262|Mandalay Bay Hotel|Las Vegas|NV|null
255|Venetian Hotel|Las Vegas|NV|null
...
```

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, specify the ALLOWOVERWRITE option.

```
unload ('select * from venue')
to 's3://mybucket/venue_pipe_'
credentials 'aws_access_key_id=<your-access-key-id>;
```

```
aws_secret_access_key=<your-secret-access-key>'
allowoverwrite;
```

UPDATE

Topics

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.

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:

```
 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
(11 rows)
```

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;

catid | catgroup |  catname  |                 catdesc
-------+----------+-----------+-----------------------------------------
6 | Theatre  | Musicals  | Musical theatre
7 | Theatre  | Plays     | All non-musical theatre
8 | Theatre  | Opera     | All opera and light opera
(3 rows)
```

Updating a table based on a current value

Update the CATNAME and CATDESC columns based on their current CATGROUP value:

```
update category
set catdesc=default, catname='Shows'
where catgroup='Theatre';
```

```
select * from category
where catname='Shows';

catid | catgroup |  catname  |                  catdesc
-------+----------+-----------+--------------------------------------------
6 | Theatre  | Shows     |
7 | Theatre  | Shows     |
8 | Theatre  | Shows     |
(3 rows)
```

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:

```
truncate category;

copy category from
's3://mybucket/data/category_pipe.txt'
credentials 'aws_access_key_id=<your-access-key-id>;aws_secret_access_key=<your-
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:

```
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:

```
select * from category order by 1;

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  | Broadway Musical
```

```
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
(11 rows)
```

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 * from category order by 1;
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
10 | Concerts | Jazz      | All jazz singers and bands
11 | Concerts | Classical | All symphony, concerto, and choir concerts
100 | Shows    | Opera     | All opera and light opera
100 | Shows    | Musicals  | Musical theatre
100 | Concerts | Pop       | All rock and pop music concerts
100 | Shows    | Plays     | All non-musical theatre
(11 rows)
```

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
(4 rows)
```

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.

See Vacuuming tables (p. 78).

Synopsis

```
VACUUM [ SORT ONLY | DELETE ONLY ] [ table_name ]
```

Parameters

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. 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. 78).

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):

```
vacuum;
```

Reclaim space and resort rows in the SALES table:

```
vacuum sales;
```

Resort rows in the SALES table:

```
vacuum sort only sales;
```

Reclaim space in the SALES table:

```
vacuum delete only sales;
```

SQL Functions Reference

Topics

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. 111).

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.

The following SQL functions are leader-node only functions and are not supported on the compute nodes

Date functions

- AGE function
- CURRENT_TIME and CURRENT_TIMESTAMP functions
- ISFINITE function
- NOW function

String functions

- ASCII function
- GET_BIT function
- GET_BYTE function
- OCTET_LENGTH function
- SET_BIT function
- SET_BYTE function
- TO_ASCII function
- TO_HEX function

System information functions

- CURRENT_SCHEMA
- CURRENT_SCHEMAS
- HAS_DATABASE_PRIVILEGE
- HAS_SCHEMA_PRIVILEGE
- HAS_TABLE_PRIVILEGE

Aggregate functions

Topics

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:

```
select avg(qtysold)from sales;

avg
```

```
-----
2
(1 row)
```

Find the average total price listed for all listings:

```
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:

```
select avg(pricepaid) as avg_price, month
from sales, date
where sales.dateid = date.dateid
group by month
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

```
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.

Data Types

The COUNT function supports all argument data types.

The return type supported by the COUNT function is BIGINT.

Examples

Count all of the users from the state of Florida:

```
select count (*) from users where state='FL';

count
-------
510
(1 row)
```

Count all of the unique venue IDs from the EVENT table:

```
select count (distinct venueid) as venues from event;

venues
--------
204
(1 row)
```

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;

count | sellerid
-------+----------
12 |    17304
11 |    25428
11 |    48950
11 |    49585
...
(16840 rows)
```

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:

```
select max(pricepaid) from sales;

max
----------
12624.00
(1 row)
```

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;

max
-------
20.00
(1 row)
```

Find the lowest price paid per ticket from all sales:

```
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 is 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.

```
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
-------+------------+-----------
17503 |   27847.76 |  27773.20
(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;

stddev
----------------
130.3912659086
(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;

stddev
--------
130
(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;

stddevsamp   |  sqrtvarsamp
----------------+---------------
130.3912659086 | 130.3912659086
(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.

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;

avg | varsamp | varpop
-----+---------+--------
10 |      54 |     54
(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;

avg | varsamp | varpop
-----+---------+---------
10 | 53.6291 | 53.6288
(1 row)
```

Bit-wise aggregate functions

Topics

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 the 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:

Integer	Binary value
3	11
7	111
10	1010
22	10110

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 `00000010`, 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 `00011111`; 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:

```
select sum(venueseats), bit_and(venueseats) from venue
where venueseats is null;

sum  | bit_and
------+---------
```

```
null |    null
(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. 308).

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. 308).

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. 308).

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. 308).

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;

userid | username | lastname  |     city     | state | likesports | liketheatre
--------+----------+-----------+--------------+-------+------------+----------
---
1 | JSG99FHE | Taylor    | Kent         | WA    | t          | t
9 | MSD36KVR | Watkins   | Port Orford  | MD    | t          | f
```

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:

	SPORTS	THEATRE	JAZZ	OPERA	ROCK	VEGAS	BROADWAY	CLASSICAL
User 1	1	1	1	1	1	1	1	1
User 2	0	0	0	0	0	0	0	0

	SPORTS	THEATRE	JAZZ	OPERA	ROCK	VEGAS	BROADWAY	CLASSICAL
User 3	1	0	1	0	0	0	0	0

In the database table, these binary values could be stored in a single LIKES column as integers:

User	Binary value	Stored value (integer)
User 1	11111111	255
User 2	00000000	0
User 3	10100000	160

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;
city      | bit_and
--------------+---------
Los Angeles   |       0
Sacramento    |       0
San Francisco |       0
San Jose      |      64
Santa Barbara |     192
(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:

```
select city, bit_or(likes) from userlikes group by city
order by city;
city      | bit_or
--------------+--------
Los Angeles   |    127
Sacramento    |    255
San Francisco |    255
San Jose      |    255
Santa Barbara |    255
(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.

```
select state, bool_or(likesports), bool_and(likesports) from users
group by state order by state limit 5;

state | bool_or | bool_and
-------+---------------------
AB    | t       | f
AK    | t       | f
AL    | t       | f
AZ    | t       | f
BC    | t       | f
(5 rows)
```

Window functions

Topics

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
- STDDEV_SAMP (synonym for STDDEV)
- SUM
- VAR_POP
- VAR_SAMP (synonym for VARIANCE)

These are the supported ranking functions:

- DENSE_RANK
- NTILE
- RANK

Aggregate window functions may have different PARTITION/ORDER BY clauses in the same query, but ranking functions may 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. For PARTITION BY, an integer constant can be used to represent the position of a field in the target list (such as PARTITION BY 3). However, a constant expression is not allowed (such as PARTITION BY 1*1+3).

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. 340).

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:

```
{UNBOUNDED PRECEDING | unsigned_value PRECEDING | CURRENT ROW}
```

or it can be a set of rows between two boundaries:

```
BETWEEN
{UNBOUNDED PRECEDING | unsigned_value { PRECEDING | FOLLOWING }
| CURRENT ROW}
AND
{UNBOUNDED FOLLOWING | unsigned_value { PRECEDING | FOLLOWING }
| 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:

```
between 5 following and 5 preceding
between current row and 2 preceding
between 3 following and current row
```

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

```
AVG ( [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 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. 312).

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 LAG window function examples (p. 332).

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. 312).

Data Types

The COUNT function supports all argument data types.

The return type supported by the COUNT function is BIGINT.

Examples

See LEAD window function examples (p. 332).

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 MAX window function examples (p. 333).

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. 312).

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 MIN window function examples (p. 334).

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. 344).

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 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 NTH_VALUE window function examples (p. 335).

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. 344).

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. 332).

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. 312).

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. 333).

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

```
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. 312).

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. 334).

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

```
NTH_VALUE (expr, offset)
[ IGNORE NULLS | RESPECT NULLS ]
OVER
( [ PARTITION BY window_partition ]
[ ORDER BY window_ordering
                          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. 312).

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. 335).

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

```
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. 335).

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

```
RANK () OVER
(
[ PARTITION BY expr_list ]
ORDER BY 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. 336).

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. 302).

STDDEV_SAMP and STDDEV are synonyms for the same function.

Synopsis

```
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. 312).

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.

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. 312).

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. 338).

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. 305).

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. 312).

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

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:

SALESID	DATEID	SELLERID	BUYERID	QTY	QTY_SHIPPED
30001	8/2/2003	3	B	10	10
10001	12/24/2003	1	C	10	10
10005	12/24/2003	1	A	30	
40001	1/9/2004	4	A	40	
10006	1/18/2004	1	C	10	
20001	2/12/2004	2	B	20	20
40005	2/12/2004	4	A	10	10
20002	2/16/2004	2	C	20	20
30003	4/18/2004	3	B	15	
30004	4/18/2004	3	B	20	
30007	9/7/2004	3	C	30	

AVG window function examples

Compute a rolling average of quantities sold by date; order the results by date ID and sales ID:

```
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)
```

COUNT window function examples

Show the sales ID, quantity, and count of all rows from the beginning of the data window:

```
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
30004 |  20 |   8
30007 |  30 |   9
40001 |  40 |   10
40005 |  10 |   11
(11 rows)
```

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.)

```
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.

```
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
30007 |  30 |     2 |   2
40001 |  40 |     1 |   1
(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.

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.

```
select salesid, sellerid, qty,
dense_rank() over(partition by sellerid order by qty desc) as d_rnk
from winsales
order by 2,3,1;

salesid | sellerid | qty | d_rnk
---------+----------+-----+-------
```

```
10001 |        1 |  10 |     2
10006 |        1 |  10 |     2
10005 |        1 |  30 |     1
20001 |        2 |  20 |     1
20002 |        2 |  20 |     1
30001 |        3 |  10 |     4
30003 |        3 |  15 |     3
30004 |        3 |  20 |     2
30007 |        3 |  30 |     1
40005 |        4 |  10 |     2
40001 |        4 |  40 |     1
(11 rows)
```

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.

```
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;

venuestate | venueseats |           venuename            |         first_value
------------+------------+--------------------------------+-------------------
-----------
CA         |      70561 | Qualcomm Stadium               | Qualcomm Stadium
CA         |      69843 | Monster Park                   | Qualcomm Stadium
CA         |      63026 | McAfee Coliseum                | Qualcomm Stadium
CA         |      56000 | Dodger Stadium                 | Qualcomm Stadium
CA         |      45050 | Angel Stadium of Anaheim       | Qualcomm Stadium
CA         |      42445 | PETCO Park                     | Qualcomm Stadium
CA         |      41503 | AT&T Park                      | Qualcomm Stadium
CA         |      22000 | Shoreline Amphitheatre         | Qualcomm Stadium
CO         |      76125 | INVESCO Field                  | INVESCO Field
CO         |      50445 | Coors Field                    | INVESCO Field
DC         |      41888 | Nationals Park                 | Nationals Park
FL         |      74916 | Dolphin Stadium                | Dolphin Stadium
FL         |      73800 | Jacksonville Municipal Stadium | Dolphin Stadium
FL         |      65647 | Raymond James Stadium          | Dolphin Stadium
FL         |      36048 | Tropicana Field                | Dolphin Stadium
...
```

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`).

```
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;

venuestate | venueseats |           venuename            |          last_value
------------+------------+--------------------------------+-------------------
-----------
CA         |      70561 | Qualcomm Stadium               | Shoreline Amphitheatre
CA         |      69843 | Monster Park                   | Shoreline Amphitheatre
CA         |      63026 | McAfee Coliseum                | Shoreline Amphitheatre
CA         |      56000 | Dodger Stadium                 | Shoreline Amphitheatre
CA         |      45050 | Angel Stadium of Anaheim       | Shoreline Amphitheatre
CA         |      42445 | PETCO Park                     | Shoreline Amphitheatre
CA         |      41503 | AT&T Park                      | Shoreline Amphitheatre
CA         |      22000 | Shoreline Amphitheatre         | Shoreline Amphitheatre
CO         |      76125 | INVESCO Field                  | Coors Field
CO         |      50445 | Coors Field                    | Coors Field
DC         |      41888 | Nationals Park                 | Nationals Park
FL         |      74916 | Dolphin Stadium                | Tropicana Field
FL         |      73800 | Jacksonville Municipal Stadium | Tropicana Field
FL         |      65647 | Raymond James Stadium          | Tropicana Field
FL         |      36048 | Tropicana Field                | Tropicana Field
...
```

The following example shows the use of the IGNORE NULLS option and relies on the addition of a new row to the VENUE table:

```
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:

```
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;

venuestate | venueseats |         venuename          | first_value
------------+------------+----------------------------+-------------
CA         |      90000 |                            |
CA         |      70561 | Qualcomm Stadium           |
CA         |      69843 | Monster Park               |
...
```

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:

```
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;

venuestate | venueseats |         venuename          |   first_value
-----------+------------+----------------------------+------------------
CA         |      90000 |                            | Qualcomm Stadium
CA         |      70561 | Qualcomm Stadium           | Qualcomm Stadium
CA         |      69843 | Monster Park               | Qualcomm Stadium
...
```

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:

```
select buyerid, saletime, qtysold,
lag(qtysold,1) over (order by buyerid, saletime) as prev_qtysold
from sales where buyerid = 3 order by buyerid, saletime;

buyerid |      saletime       | qtysold | prev_qtysold
---------+---------------------+---------+--------------
3 | 2008-01-16 01:06:09 |       1 |
3 | 2008-01-28 02:10:01 |       1 |            1
3 | 2008-03-12 10:39:53 |       1 |            1
3 | 2008-03-13 02:56:07 |       1 |            1
3 | 2008-03-29 08:21:39 |       2 |            1
3 | 2008-04-27 02:39:01 |       1 |            2
3 | 2008-08-16 07:04:37 |       2 |            1
3 | 2008-08-22 11:45:26 |       2 |            2
3 | 2008-09-12 09:11:25 |       1 |            2
3 | 2008-10-01 06:22:37 |       1 |            1
3 | 2008-10-20 01:55:51 |       2 |            1
3 | 2008-10-28 01:30:40 |       1 |            2
(12 rows)
```

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.

```
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;

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 |      70.80 | 2008-01-01 06:06:57 |     50.55
1749 |      50.55 | 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 | 2008-01-02 01:11:16 |     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 03:12:50 |     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:

```
select salesid, qty,
max(qty) over (order by salesid rows unbounded preceding) as max
from winsales
order by salesid;

salesid | qty | max
---------+-----+-----
10001 |  10 |  10
10005 |  30 |  30
10006 |  10 |  30
20001 |  20 |  30
20002 |  20 |  30
30001 |  10 |  30
30003 |  15 |  30
30004 |  20 |  30
30007 |  30 |  30
40001 |  40 |  40
40005 |  10 |  40
(11 rows)
```

Show the salesid, quantity, and maximum quantity in a restricted frame:

```
select salesid, qty,
max(qty) over (order by salesid rows between 2 preceding and 1 preceding) as
max
from winsales
order by salesid;

salesid | qty | max
---------+-----+-----
10001 |  10 |
10005 |  30 |  10
```

```
10006 |  10 |  30
20001 |  20 |  30
20002 |  20 |  20
30001 |  10 |  20
30003 |  15 |  20
30004 |  20 |  15
30007 |  30 |  20
40001 |  40 |  30
40005 |  10 |  40
(11 rows)
```

MIN window function examples

Show the sales ID, quantity, and minimum quantity from the beginning of the data window:

```
select salesid, qty,
min(qty) over
(order by salesid rows unbounded preceding)
from winsales
order by salesid;

salesid | qty | min
---------+-----+-----
10001 |  10 |  10
10005 |  30 |  10
10006 |  10 |  10
20001 |  20 |  10
20002 |  20 |  10
30001 |  10 |  10
30003 |  15 |  10
30004 |  20 |  10
30007 |  30 |  10
40001 |  40 |  10
40005 |  10 |  10
(11 rows)
```

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
FL         | Raymond James Stadium          |      65647 |            65647
FL         | Tropicana Field                |      36048 |            65647
NY         | Ralph Wilson Stadium           |      73967 |            20000
NY         | Yankee Stadium                 |      52325 |            20000
NY         | Madison Square Garden          |      20000 |            20000
(15 rows)
```

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:

```
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='Ham
let'
and caldate='2008-08-26'
order by 4;

eventname |  caldate   | pricepaid | ntile
----------+------------+-----------+-------
Hamlet    | 2008-08-26 |   1883.00 |     1
Hamlet    | 2008-08-26 |   1065.00 |     1
Hamlet    | 2008-08-26 |    589.00 |     1
Hamlet    | 2008-08-26 |    530.00 |     1
```

```
Hamlet    | 2008-08-26 |    472.00 |     1
Hamlet    | 2008-08-26 |    460.00 |     2
Hamlet    | 2008-08-26 |    355.00 |     2
Hamlet    | 2008-08-26 |    334.00 |     2
Hamlet    | 2008-08-26 |    296.00 |     2
Hamlet    | 2008-08-26 |    230.00 |     3
Hamlet    | 2008-08-26 |    216.00 |     3
Hamlet    | 2008-08-26 |    212.00 |     3
Hamlet    | 2008-08-26 |    106.00 |     3
Hamlet    | 2008-08-26 |    100.00 |     4
Hamlet    | 2008-08-26 |     94.00 |     4
Hamlet    | 2008-08-26 |     53.00 |     4
Hamlet    | 2008-08-26 |     25.00 |     4
(17 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;

salesid | qty | rnk
--------+-----+-----
10001 |  10 |  1
10006 |  10 |  1
30001 |  10 |  1
40005 |  10 |  1
30003 |  15 |  5
20001 |  20 |  6
20002 |  20 |  6
30004 |  20 |  6
10005 |  30 |  9
30007 |  30 |  9
40001 |  40 |  11
(11 rows)
```

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.

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.

```
select salesid, qty,
rank() over (order by qty desc) as rnk
from winsales
order by 2,1;
```

```
salesid | qty | rnk
---------+-----+-----
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)
```

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.

```
select salesid, sellerid, qty, rank() over
(partition by sellerid
order by qty desc) as rnk
from winsales
order by 2,3,1;

salesid | sellerid | qty | rnk
--------+----------+-----+-----
10001 |        1 |  10 |  2
10006 |        1 |  10 |  2
10005 |        1 |  30 |  1
20001 |        2 |  20 |  1
20002 |        2 |  20 |  1
30001 |        3 |  10 |  4
30003 |        3 |  15 |  3
30004 |        3 |  20 |  2
30007 |        3 |  30 |  1
40005 |        4 |  10 |  2
40001 |        4 |  40 |  1
(11 rows)
```

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.

```
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;

salesid | dateid | pricepaid | stddevpop | varpop
---------+--------+-----------+-----------+---------
```

```
33095 |   1827 |    234.00 |         0 |       0
65082 |   1827 |    472.00 |       119 |   14161
88268 |   1827 |    836.00 |       248 |   61283
97197 |   1827 |    708.00 |       230 |   53019
110328 |   1827 |    347.00 |       223 |   49845
110917 |   1827 |    337.00 |       215 |   46159
150314 |   1827 |    688.00 |       211 |   44414
157751 |   1827 |   1730.00 |       447 |  199679
165890 |   1827 |   4192.00 |      1185 | 1403323
...
```

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;

salesid |   dateid   | sellerid | qty | sum
---------+------------+----------+-----+-----
30001 | 2003-08-02 |        3 |  10 |  10
10001 | 2003-12-24 |        1 |  10 |  20
10005 | 2003-12-24 |        1 |  30 |  50
40001 | 2004-01-09 |        4 |  40 |  90
10006 | 2004-01-18 |        1 |  10 | 100
20001 | 2004-02-12 |        2 |  20 | 120
40005 | 2004-02-12 |        4 |  10 | 130
20002 | 2004-02-16 |        2 |  20 | 150
30003 | 2004-04-18 |        3 |  15 | 165
30004 | 2004-04-18 |        3 |  20 | 185
30007 | 2004-09-07 |        3 |  30 | 215
(11 rows)
```

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;

salesid |   dateid   | sellerid | qty | sum
---------+------------+----------+-----+-----
30001 | 2003-08-02 |        3 |  10 |  10
10001 | 2003-12-24 |        1 |  10 |  10
10005 | 2003-12-24 |        1 |  30 |  40
40001 | 2004-01-09 |        4 |  40 |  40
10006 | 2004-01-18 |        1 |  10 |  50
20001 | 2004-02-12 |        2 |  20 |  20
```

```
40005 | 2004-02-12 |        4 |  10 |  50
20002 | 2004-02-16 |        2 |  20 |  40
30003 | 2004-04-18 |        3 |  15 |  25
30004 | 2004-04-18 |        3 |  20 |  45
30007 | 2004-09-07 |        3 |  30 |  75
(11 rows)
```

Number rows sequentially

Number all of the rows in the result set, ordered by the SELLERID and SALESID columns:

```
select salesid, sellerid, qty,
sum(1) over (order by sellerid, salesid rows unbounded preceding) as rownum
from winsales
order by 2,1;

salesid | sellerid |  qty | rownum
--------+----------+------+--------
10001 |        1 |   10 |     1
10005 |        1 |   30 |     2
10006 |        1 |   10 |     3
20001 |        2 |   20 |     4
20002 |        2 |   20 |     5
30001 |        3 |   10 |     6
30003 |        3 |   15 |     7
30004 |        3 |   20 |     8
30007 |        3 |   30 |     9
40001 |        4 |   40 |    10
40005 |        4 |   10 |    11
(11 rows)
```

Number all rows in the result set, partition the results by SELLERID, and order the results by SELLERID and SALESID within the partition:

```
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;

salesid | sellerid | qty | rownum
---------+----------+-----+--------
10001 |        1 |  10 |      1
10005 |        1 |  30 |      2
10006 |        1 |  10 |      3
20001 |        2 |  20 |      1
20002 |        2 |  20 |      2
30001 |        3 |  10 |      1
30003 |        3 |  15 |      2
30004 |        3 |  20 |      3
30007 |        3 |  30 |      4
40001 |        4 |  40 |      1
40005 |        4 |  10 |      2
(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.

```
select dateid, pricepaid,
sum(pricepaid) over(order by dateid rows unbounded preceding) as sumpaid
from sales
group by dateid, pricepaid;

dateid | pricepaid |   sumpaid
--------+-----------+-------------
1827 |   1730.00 |     1730.00
1827 |    708.00 |     2438.00
1827 |    234.00 |     2672.00
...

select dateid, pricepaid,
sum(pricepaid) over(order by dateid rows unbounded preceding) as sumpaid
from sales
group by dateid, pricepaid;

dateid | pricepaid |   sumpaid
--------+-----------+-------------
1827 |    234.00 |      234.00
1827 |    472.00 |      706.00
1827 |    347.00 |     1053.00
...
```

In this case, adding a second ORDER BY column to the window function may solve the problem:

```
select dateid, pricepaid,
sum(pricepaid) over(order by dateid, pricepaid rows unbounded preceding) as
sumpaid
from sales
group by dateid, pricepaid;

dateid | pricepaid | sumpaid
--------+-----------+---------
1827 |    234.00 |  234.00
1827 |    337.00 |  571.00
1827 |    347.00 |  918.00
...
```

Conditional expressions

Topics

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 is used to replace `New York City` with `Big Apple` in a query against the VENUE table. Replace all other city names with `other`.

```
select venuecity,
case venuecity
when 'New York City'
then 'Big Apple' else 'other'
end from venue
order by venueid desc;

venuecity        |   case
-----------------+-----------
Los Angeles      | other
New York City    | Big Apple
San Francisco    | other
Baltimore        | other
...
(202 rows)
```

Use a searched CASE expression to assign group numbers based on the PRICEPAID value for individual ticket sales:

```
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;

pricepaid |  case
----------+---------
12624.00 | group 2
10000.00 | group 3
10000.00 | group 3
9996.00 | group 1
9988.00 | group 1
...
(172456 rows)
```

COALESCE

Synonym of the NVL expression.

See NVL expression (p. 344).

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;

case
----------------
June 1st, 2008

...
(30 rows)
```

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`.

```
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;

catid  | case
-------+---------------------------------
1  | Major League Baseball
2  | National Hockey League
3  | National Football League
4  | National Basketball Association
5  | Major League Soccer
6  | Unknown
7  | Unknown
8  | Unknown
9  | Unknown
10 | Unknown
11 | Unknown
(11 rows)
```

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.

```
select venuename, venuestate, decode(venueseats,null,0,1)
from venue
where venuestate in('NV','CO')
order by 2,3,1;

venuename               | venuestate | case
------------------------+------------+------
Coors Field                | CO   | 1
Dick's Sporting Goods Park | CO   | 1
Ellie Caulkins Opera House | CO   | 1
INVESCO Field     | CO   | 1
Pepsi Center      | CO   | 1
Ballys Hotel      | NV   | 0
Bellagio Hotel    | NV   | 0
Caesars Palace    | NV   | 0
Harrahs Hotel     | NV   | 0
Hilton Hotel      | NV   | 0
...
(20 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.

```
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;

coalesce
------------
2008-06-01
2008-12-31
2008-12-31
```

The default column name for an NVL expression is COALESCE. The following query would return the same results:

```
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`:

```
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.

```
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';

catgroup
----------
null
(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;
c1
```

```
------
null
null
(2 rows)
```

Date functions

Topics

This section contains the date and time scalar functions that Amazon Redshift supports.

ADD_MONTHS (Oracle compatibility function)

The ADD_MONTHS function adds the specified number of months to a datetime value or expression. The DATEADD function provides similar functionality.

Synopsis

```
ADD_MONTHS ( date, num_months )
```

Arguments

date
A datetime expression or any value that implicitly converts to a timestamp. 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.

```
select distinct trunc(add_months(caldate, 12)) as calplus12,
trunc(caldate) as cal
from date
order by 1 asc;

calplus12  |    cal
-----------+-----------
2009-01-01 | 2008-01-01
2009-01-02 | 2008-01-02
2009-01-03 | 2008-01-03
...
(365 rows)
```

The following examples demonstrate the behavior when the ADD_MONTHS function operates on dates with months that have different numbers of days.

```
select add_months('2008-03-31',1);

add_months
---------------------
2008-04-30 00:00:00
(1 row)

select add_months('2008-04-30',1);

add_months
---------------------
2008-05-31 00:00:00
(1 row)
```

AGE function

The AGE function returns the interval of time from a timestamp to midnight on the current date or the interval of time elapsed between two timestamps.

Syntax

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.

```
AGE(timestamp [, timestamp ])
```

Arguments

timestamp
A single timestamp if you want to return the interval of time from the timestamp to midnight on the current date, or the first timestamp if you want to return the time elapsed between two timestamps.

timestamp
The second timestamp if you want to return the time elapsed between two timestamps.

Return type

The AGE function returns an INTERVAL.

Examples

The following example returns the interval of time between a timestamp and midnight on the current date:

```
select age(timestamp '1956-07-10 12:42:05');
age
---------------------------------
55 years 4 mons 19 days 11:17:55
(1 row)
```

The following example returns the interval of time between two timestamps:

```
select age(timestamp '1998-07-28 04:05:11', timestamp '2011-09-22 12:33:33');
age
-------------------------------------
-13 years -1 mons -25 days -08:28:22
(1 row)
```

CONVERT_TIMEZONE function

CONVERT_TIMEZONE converts a timestamp from one timezone to another.

Syntax

```
CONVERT_TIMEZONE ( ['source_zone',] 'target_zone', 'timestamp')
```

Arguments

source_zone
(Optional) The time zone of the current timestamp. The default is UTC.

target_zone
The time zone for the new timestamp.

timestamp
The timestamp value to be converted.

Return type

CONVERT_TIMEZONE returns a TIMESTAMP.

Examples

The following example converts the timestamp value in the LISTTIME column from the default UTC timezone to PST:

```
select listtime, convert_timezone('PST', listtime) from listing;

     listtime       |   convert_timezone
--------------------+---------------------
2008-03-05 12:25:29 2008-03-05 04:25:29
```

The following example converts a timestamp string from EST to PST:

```
select convert_timezone('EST', 'PST', '20080305 12:25:29');

 convert_timezone
-------------------
2008-03-05 09:25:29
```

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 date, time, and weekday.

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 Oct 09 09:04:45.237819 2008 PDT
(1 row)
```

CURRENT_TIME and CURRENT_TIMESTAMP functions

The CURRENT_TIME and CURRENT_TIMESTAMP functions return the start time of the current transaction.

Syntax

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_TIME [ (precision) ]
```

```
CURRENT_TIMESTAMP [ (precision) ]
```

You can add an optional precision parameter to the CURRENT_TIME and CURRENT_TIMESTAMP functions, which rounds the results in the seconds field to the number of digits that you specify.

Return type

The CURRENT_TIME function returns the current time with timezone in the default format. The CURRENT_TIMESTAMP function returns the current date and time with timezone in the default format.

Examples

The following examples return the current time with and without precision:

```
select current_time;
timetz
------------
15:43:41.229380-07
(1 row)
```

```
select current_time(2);
timetz
------------
15:43:41.23-07
(1 row)
```

The following examples return the current timestamp with and without precision:

```
select current_timestamp(3);
timestamptz
------------
```

```
2008-10-01 15:48:42.301-07
(1 row)
```

```
select current_timestamp;
timestamptz
------------
2008-10-01 15:48:42.301004-07
(1 row)
```

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:

```
select caldate, '2008-01-04',
date_cmp(caldate,'2008-01-04')
from date
order by dateid
limit 10;

caldate   |  ?column?  | date_cmp
----------+------------+----------
2008-01-01 | 2008-01-04 |       -1
2008-01-02 | 2008-01-04 |       -1
2008-01-03 | 2008-01-04 |       -1
2008-01-04 | 2008-01-04 |        0
2008-01-05 | 2008-01-04 |        1
2008-01-06 | 2008-01-04 |        1
2008-01-07 | 2008-01-04 |        1
2008-01-08 | 2008-01-04 |        1
2008-01-09 | 2008-01-04 |        1
2008-01-10 | 2008-01-04 |        1
(10 rows)
```

DATE_CMP_TIMESTAMP function

Compares the value of a date to a specified timestamp and returns an integer. If the date is "greater" alphabetically, returns 1. If the timestamp is "greater", returns -1. If the date and timestamp are identical, returns a 0.

Synopsis

```
TIMESTAMP_CMP_DATE(date, timestamp)
```

Arguments

date1
The first input parameter is a DATE.

date2
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 | 2008-11-09 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:

```
select caldate, date_part_year(caldate)
from date
order by
dateid limit 10;

caldate   | date_part_year
------------+---------------
2008-01-01 |           2008
2008-01-02 |           2008
2008-01-03 |           2008
2008-01-04 |           2008
2008-01-05 |           2008
2008-01-06 |           2008
2008-01-07 |           2008
2008-01-08 |           2008
2008-01-09 |           2008
2008-01-10 |           2008
(10 rows)
```

DATEADD function

Given a datepart and an expression, this function increments datetime values by a specified interval.

Synopsis

```
DATEADD ( datepart, interval, expression )
```

This function returns a timestamp 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 datetime functions (p. 367).

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 datetime 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:

```
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:

```
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 timestamp for a date value is `00:00:00`.

Add 30 minutes to a date value that does not specify a timestamp:

```
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:

```
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:

```
select dateadd(month,1,'2008-04-30');

date_add
---------------------
2008-05-30 00:00:00
(1 row)
```

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 datetime functions (p. 367). 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 timestamps, `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 datetime expressions and they must both contain the specified datepart.

Return type

DATEDIFF returns an integer.

Examples

Find the difference, in number of weeks, between two literal date values:

```
select datediff(week,'2009-01-01','2009-12-31') as numweeks;

numweeks
----------
```

```
52
(1 row)
```

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);

date_diff
-----------
40
(1 row)
```

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;

priceperticket | wait
----------------+------
96.00 |   15
123.00 |   11
131.00 |    9
123.00 |    6
129.00 |    4
96.00 |    4
96.00 |    0
(7 rows)
```

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;

avgwait
---------
465
(1 row)
```

DATE_PART function

This function extracts datepart values from an expression. Synonym of the PGDATE_PART function.

Synopsis

```
DATE_PART ( datepart, expression )
```

Arguments

datepart
The specific part of the date value (year, month, or day, for example) that the function operates on. See Dateparts for datetime functions (p. 367).

expression
The target column or expression that the function operates on. The expression must be a datetime 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:

```
select date_part(w, listtime) as weeks, listtime
from listing where listid=10;

weeks |      listtime
-------+---------------------
25 | 2008-06-17 09:44:54
(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):

```
select date_part(dow, starttime) as dow,
starttime from event
where date_part(dow, starttime)=6
order by 2,1;

dow |      starttime
-----+---------------------
6 | 2008-01-05 14:00:00
6 | 2008-01-05 14:00:00
6 | 2008-01-05 14:00:00
6 | 2008-01-05 14:00:00
...
(1147 rows)
```

Apply the DATE_PART function to a literal date value:

```
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 timestamp expression or literal based on the time interval that you specify, such as hour, week, or month. Truncates means that the timestamp is reduced to the first of the specified year for year, first of the specified month for month, and so on.

Synopsis

```
DATE_TRUNC('field', source)
```

Arguments

field

The first parameter designates the precision to which to truncate the timestamp value. See Dateparts for datetime functions (p. 367) for valid formats.

source

The second parameter is a timestamp value or expression.

Return type

The DATE_TRUNC function returns a timestamp.

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 timestamp value or expression.

Synopsis

```
EXTRACT ( datepart FROM
{ TIMESTAMP 'literal' | timestamp }
)
```

Arguments

datepart
See Dateparts for datetime functions (p. 367).

literal
A timestamp value, enclosed in quotes and preceded by the TIMESTAMP keyword.

timestamp
A timestamp 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.

```
select salesid, extract(week from saletime) as weeknum
from sales where pricepaid > 9999 order by 2;

salesid | weeknum
---------+---------
159073 |       6
160318 |       8
161723 |      26
(3 rows)
```

Return the minute value from a literal timestamp value:

```
select extract(minute from timestamp '2009-09-09 12:08:43');
date_part
-----------
8
(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

```
GETDATE()
```

The parentheses are required.

Return type

GETDATE returns a TIMESTAMP.

Examples

The following example uses the GETDATE() function to return the full timestamp for the current date:

```
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:

```
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

```
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":

```
select interval_cmp('3 days','1 year');

interval_cmp
--------------
-1
```

This example compares the value "7 days" to "1 week":

```
select interval_cmp('7 days','1 week');

interval_cmp
--------------
0
(1 row)
```

ISFINITE function

The ISFINITE function determines if a date, timestamp, or interval is a finite number.

Syntax

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.

```
ISFINITE([ date date | timestamp timestamp | interval interval ])
```

Arguments

date
A date if you want to determine if a date is finite.

timestamp
A timestamp if you want to determine if a timestamp is finite.

interval
An interval if you want to determine if an interval is finite.

Return type

The ISFINITE function returns a BOOLEAN.

Examples

The following example queries whether the date January 1, 2012 is a finite number:

```
select isfinite(date '2012-01-01');
isfinite
----------
t
(1 row)
```

The following example queries whether a timestamp is a finite number:

```
select isfinite(timestamp '2008-07-06 12:45:08');
isfinite
----------
t
(1 row)
```

The following example queries whether an interval is a finite number:

```
select isfinite(interval '12673 hours');
isfinite
----------
t
(1 row)
```

LOCALTIME and LOCALTIMESTAMP functions

Synopsis

The LOCALTIME and LOCALTIMESTAMP functions return the current local time, without time zones.

```
LOCALTIME [ (precision) ]
```

```
LOCALTIMESTAMP [ (precision) ]
```

Arguments

***precision* (optional)**
Integer specifying the number of digits of precision used to round the returned timestamp.

Return type

LOCALTIME and LOCALTIMESTAMP return a TIMESTAMP WITHOUT TIMEZONE.

Examples

Show the local time with and without precision:

```
select localtime;
time
-----------------
13:38:39.036675
(1 row)
```

```
select localtime(2);
time
-------------
13:38:39.04
(1 row)
```

Show the local timestamp with and without precision:

```
select localtimestamp;
timestamp
------------
2008-10-08 13:41:34.359875
(1 row)
```

```
select localtimestamp(3);
timestamp
------------
2008-10-08 13:41:34.359
(1 row)
```

NOW function

The NOW function returns the start time of the current statement. NOW takes no parameters.

Syntax

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.

```
NOW()
```

Return type

The NOW function returns a TIMESTAMP with timezone.

Examples

The following example returns the start time of the statement:

```
select now();
now
-------------------------------
2011-11-30 15:55:12.214919-08
(1 row)
```

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 timestamp 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
------------
2008-12-04
(1 row)
```

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:

```
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;

salesid | pricepaid |  saletime  |    now
---------+-----------+------------+------------
91535 |    670.00 | 2008-08-07 | 2008-12-05
91635 |    365.00 | 2008-08-07 | 2008-12-05
91901 |   1002.00 | 2008-08-07 | 2008-12-05
...
```

TIMESTAMP_CMP function

Compares the value of two timestamps and returns an integer. If the timestamps are identical, returns 0. If the first timestamp is "greater", returns 1. If the second timestamp is "greater", returns -1.

Synopsis

```
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 timestamp for the sale is after the timestamp for the listing:

```
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;

listid |      listtime       |      saletime       | timestamp_cmp
--------+---------------------+---------------------+---------------
1 | 2008-01-24 06:43:29 | 2008-02-18 02:36:48 |            -1
4 | 2008-05-24 01:18:37 | 2008-06-06 05:00:16 |            -1
5 | 2008-05-17 02:29:11 | 2008-06-06 08:26:17 |            -1
5 | 2008-05-17 02:29:11 | 2008-06-09 08:38:52 |            -1
6 | 2008-08-15 02:08:13 | 2008-08-31 09:17:02 |            -1
10 | 2008-06-17 09:44:54 | 2008-06-26 12:56:06 |            -1
10 | 2008-06-17 09:44:54 | 2008-07-10 02:12:36 |            -1
10 | 2008-06-17 09:44:54 | 2008-07-16 11:59:24 |            -1
10 | 2008-06-17 09:44:54 | 2008-07-22 02:23:17 |            -1
12 | 2008-07-25 01:45:49 | 2008-08-04 03:06:36 |            -1
(10 rows)
```

This example shows that TIMESTAMP_CMP returns a 0 for identical timestamps:

```
select listid, timestamp_cmp(listtime, listtime)
from listing
order by 1 , 2
limit 10;


listid | timestamp_cmp
--------+---------------
1 |             0
2 |             0
3 |             0
4 |             0
5 |             0
6 |             0
7 |             0
8 |             0
9 |             0
10 |             0
(10 rows)
```

TIMESTAMP_CMP_DATE function

Compares the value of a timestamp to a specified date and returns an integer. If the first timestamp is "greater" alphabetically, returns 1. If the second timestamp is "greater", returns -1. If the timestamp and date are identical, returns a 0.

Syntax

```
TIMESTAMP_CMP_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_CMP_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;


listid |      listtime       | timestamp_cmp_date
--------+---------------------+--------------------
1 | 2008-01-24 06:43:29 |                  -1
2 | 2008-03-05 12:25:29 |                  -1
3 | 2008-11-01 07:35:33 |                   1
4 | 2008-05-24 01:18:37 |                  -1
5 | 2008-05-17 02:29:11 |                  -1
6 | 2008-08-15 02:08:13 |                   1
7 | 2008-11-15 09:38:15 |                   1
8 | 2008-11-09 05:07:30 |                   1
9 | 2008-09-09 08:03:36 |                   1
10 | 2008-06-17 09:44:54 |                  -1
(10 rows)
```

TRUNC(timestamp)

See TRUNC function (p. 392).

Dateparts for datetime 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

Datepart or timepart	Abbreviations
millennium, millennia	mil, mils
century, centuries	c, cent, cents
decade, decades	dec, decs
epoch	epoch (supported by the DATEPART, DATE_PART, and EXTRACT functions)
year, years	y, yr, yrs
quarter, quarters	qtr, qtrs
month, months	mon, mons
week, weeks	w
day of week	dayofweek, dow, dw, weekday (supported by the DATEPART, DATE_PART, and EXTRACT functions)
day of year	dayofyear, doy, dy, yearday (supported by the DATEPART, DATE_PART, and EXTRACT functions)
day, days	d
hour, hours	h, hr, hrs
minute, minutes	m, min, mins
second, seconds	s, sec, secs
millisecond, milliseconds	ms, msec, msecs, msecond, mseconds, millisec, millisecs, millisecon
microsecond, microseconds	microsec, microsecs, microsecon, usecond, useconds, us, usec, usecs

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. I f 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 timestamp, regardless of the specified datepart, returning either a decimal value or an integer as required.

For example, compare the results of the following queries:

```
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;
pgdate_part
-------------
3.189717
(1 row)
```

CENTURY, DECADE, and MIL notes

CENTURY or CENTURIES

Amazon Redshift interprets a CENTURY to start with year *###1* and end with year ###0:

```
select extract (century from timestamp '2000-12-16 12:21:13');
date_part
-----------
20
(1 row)

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:

```
select extract(decade from timestamp '1999-02-16 20:38:40');
date_part
-----------
200
(1 row)

select extract(decade from timestamp '2000-02-16 20:38:40');
date_part
-----------
201
(1 row)
```

```
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`:

```
select extract (mil from timestamp '2000-12-16 12:21:13');
date_part
-----------
2
(1 row)

select extract (mil from timestamp '2001-12-16 12:21:13');
date_part
-----------
3
(1 row)
```

Math functions

Topics

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

Operator	Description	Example	Result
+	addition	2 + 3	5
-	subtraction	2 - 3	-1
*	multiplication	2 * 3	6
/	division	4 / 2	2
%	modulo	5 % 4	1
^	exponentiation	2.0 ^ 3.0	8
|/	square root	| / 25.0	5
||/	cube root	|| / 27.0	3
@	absolute value	@ -5.0	5
<<	bitwise shift left	1 << 4	16
>>	bitwise shift right	8 >> 2	2

Examples

Calculate the commission paid plus a $2.00 handling for a given transaction:

```
select commission, (commission + 2.00) as comm
from sales where salesid=10000;

commission | comm
-----------+-------
28.05 | 30.05
(1 row)
```

Calculate 20% of the sales price for a given transaction:

```
select pricepaid, (pricepaid * .20) as twentypct
from sales where salesid=10000;

pricepaid | twentypct
-----------+-----------
187.00 |    37.400
(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;

sum_price | dateid | sum_comm |   value
-----------+--------+----------+-----------
364445.00 |   2044 | 54666.75 | 309778.25
349344.00 |   2112 | 52401.60 | 296942.40
343756.00 |   2124 | 51563.40 | 292192.60
378595.00 |   2116 | 56789.25 | 321805.75
328725.00 |   2080 | 49308.75 | 279416.25
349554.00 |   2028 | 52433.10 | 297120.90
249207.00 |   2164 | 37381.05 | 211825.95
285202.00 |   2064 | 42780.30 | 242421.70
320945.00 |   2012 | 48141.75 | 272803.25
321096.00 |   2016 | 48164.40 | 272931.60
(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:

```
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:

```
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:

```
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. 382) 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.

```
select (select sum(qtysold) from sales, date
where sales.dateid=date.dateid
and year=2008) * dexp((7::float/100)*10) qty2010;

qty2010
------------------
695447.483772222
(1 row)
```

DLOG1 function

The DLOG1 function returns the natural logarithm of the input parameter. Synonym for the LN function.

Synonym of LN function (p. 382).

DLOG10 function

The DLOG10 returns the base 10 logarithm of the input parameter. Synonym of the LOG function.

Synonym of LOG function (p. 384).

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);

dlog10
--------
2
(1 row)
```

EXP function

The EXP function returns the exponential value in scientific notation for a numeric expression.

Synopsis

```
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;

qty2010
```

```
------------------
695447.483772222
(1 row)
```

FLOOR function

The FLOOR function rounds a number down to the next whole number.

Synopsis

```
FLOOR (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. 380).

Synopsis

```
LN(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 |  1.09861228866811
XDZ38RDD |  1.38629436111989
AEB55QTM |   1.6094379124341
NDQ15VBM |  1.79175946922805
OWY35QYB |  1.94591014905531
AZG78YIP |  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. 380).

Synopsis

```
LOG(number)
```

Argument

number
The input parameter is a double precision number.

Return type

The LOG function returns a double precision number.

Example

The following example returns the base 10 logarithm of the number 100:

```
select log(100);
dlog10
--------
2
(1 row)
```

MOD function

The MOD function returns a numeric result that is the remainder of two numeric parameters. The first parameter is divided by the second parameter.

Synopsis

```
MOD(number1, number2)
```

Arguments

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 or BIGINT, but one parameter cannot be a SMALLINT if the other is a BIGINT.

number2
The second parameter is an INTEGER, SMALLINT, BIGINT, or DECIMAL number. The same data type rules apply to *number2* as to *number1*.

Return type

Valid return types are DECIMAL, INT, SMALLINT, and BIGINT. The return type of the 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 CATEGORY table:

```
select catid, catname
from category
where mod(catid,2)=1
order by 1,2;

catid |  catname
-------+-----------
1 | MLB
3 | NFL
5 | MLS
7 | Plays
9 | Pop
11 | Classical
(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.

```
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):

```
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 RADIANS function returns a double precision number.

Examples

The following example returns the radian equivalent of 180 degrees:

```
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. 276) 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:

```
select cast (random() * 100 as int);
int4
------
24
(1 row)
```

This example uses the SET (p. 276) 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:

```
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;

commission | round
-----------+-------
28.05 |    28
(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;

commission | round
-----------+-------
28.05 |  28.1
(1 row)
```

For the same query, extend the precision in the opposite direction.

```
select commission, round(commission, -1)
from sales where salesid=10000;

commission | round
-----------+-------
28.05 |    30
(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());

sin
-----------------------
-1.22464679914735e-16
(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;

commission | sign
-----------+------
28.05 |    1
(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.

```
select sqrt(commission)
from sales where salesid <10 order by salesid;

sqrt
------------------
10.4498803820905
3.37638860322683
7.24568837309472
5.1234753829798
...
```

The following query returns the rounded square root for the same set of COMMISSION values.

```
select salesid, commission, round(sqrt(commission))
from sales where salesid <10 order by salesid;
```

```
salesid | commission | round
---------+------------+-------
1 |     109.20 |    10
2 |      11.40 |     3
3 |      52.50 |     7
4 |      26.25 |     5
...
```

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);
tan
-----
0
(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 timestamp 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 timestamp. (To return a timestamp value with `00:00:00` as the time, cast the function result to a timestamp.)

Return type

TRUNC returns the same numeric data type as the first input argument. For timestamps, TRUNC returns a date.

Examples

Truncate the commission paid for a given sales transaction.

```
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.

```
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`.

```
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 timestamp):

```
select sysdate;

timestamp
---------------------------
```

```
2011-07-21 10:32:38.248109
(1 row)

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;

trunc
------------
2008-01-25
(1 row)
```

String functions

Topics

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.

|| (Concatenation) operator

Concatenates two strings on either side of the || symbol and returns the concatenated string.

Similar to CONCAT (Oracle compatibility function) (p. 401).

> **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:

```
select firstname || ' ' || lastname
from users
order by 1
limit 10;

?column?
-----------------
Aaron Banks
Aaron Booth
Aaron Browning
Aaron Burnett
Aaron Casey
Aaron Cash
Aaron Castro
Aaron Dickerson
Aaron Dixon
Aaron Dotson
(10 rows)
```

ASCII function

The ASCII function returns the ASCII code point value for the first character in the argument.

Syntax

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.

```
ASCII(string)
```

Arguments

string
The input parameter is a character such as a symbol or letter.

Return type

The ASCII function returns an INTEGER.

Examples

The following example returns the ASCII code point value for an exclamation mark (!):

```
select ascii('!');
ascii
-------
33
(1 row)
```

This example provides the ASCII code for the letter a:

```
select ascii('a');
ascii
-------
97
(1 row)
```

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. 399).

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:

```
select userid, firstname, lastname,
bpcharcmp(firstname, lastname)
from users
order by 1, 2, 3, 4
limit 10;
```

This example returns the following sample output:

```
 userid | firstname | lastname  | bpcharcmp
--------+-----------+-----------+-----------
1 | Rafael    | Taylor    |        -1
2 | Vladimir  | Humphrey  |         1
3 | Lars      | Ratliff   |        -1
4 | Barry     | Roy       |        -1
5 | Reagan    | Hodge     |         1
6 | Victor    | Hernandez |         1
7 | Tamekah   | Juarez    |         1
8 | Colton    | Roy       |        -1
```

```
9 | Mufutau   | Watkins   |        -1
10 | Naida     | Calderon  |         1
(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:

```
select userid, firstname, lastname,
bpcharcmp(firstname, lastname)
from users where bpcharcmp(firstname, lastname)=0
order by 1, 2, 3, 4;


userid | firstname | lastname | bpcharcmp
--------+-----------+----------+-----------
62 | Chase     | Chase    |         0
4008 | Whitney   | Whitney  |         0
12516 | Graham    | Graham   |         0
13570 | Harper    | Harper   |         0
16712 | Cooper    | Cooper   |         0
18359 | Chase     | Chase    |         0
27530 | Bradley   | Bradley  |         0
31204 | Harding   | Harding  |         0
(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 '`:

```
select '     abc    ' as untrim, btrim('     abc    ') as trim;

untrim    | trim
--------------+------
abc     | abc
(1 row)
```

The following example removes the leading and trailing `'xyz'` strings from the string `'xyzaxyzbxyzcxyz'`

```
select 'xyzaxyzbxyzcxyz' as untrim,
btrim('xyzaxyzbxyzcxyz', 'xyz') as trim;

untrim      |   trim
-----------------+-----------
xyzaxyzbxyzcxyz | axyzbxyzc
(1 row)
```

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. 397) for details.

CHAR_LENGTH function

Synonym of the LEN function.

See LEN function (p. 407)

CHARACTER_LENGTH function

Synonym of the LEN function.

See LEN function (p. 407)

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('.', commis
sion)
order by 1,2;

charindex | count
----------+-------
5 | 629
(1 row)
```

See STRPOS function (p. 420) for details.

CHR function

The CHR function returns the character that matches the ASCII code point value specified by of 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);

eventname
---------------------------
Adriana Lecouvreur
A Man For All Seasons
A Bronx Tale
A Christmas Carol
Allman Brothers Band
...
```

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');

concat
-------------------
December 25, 2008
(1 row)
```

The following query, using the || operator instead of CONCAT, produces the same result:

```
select 'December 25, '||'2008';

?column?
-------------------
December 25, 2008
(1 row)
```

The following example uses two CONCAT functions to concatenate three character strings:

```
select concat('Thursday, ', concat('December 25, ', '2008'));

concat
----------------------------
Thursday, December 25, 2008
(1 row)
```

The following query concatenates CITY and STATE values from the VENUE table:

```
select concat(venuecity, venuestate)
from venue
where venueseats > 75000
order by venueseats;

concat
-------------------
DenverCO
Kansas CityMO
East RutherfordNJ
LandoverMD
(4 rows)
```

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:

```
select concat(concat(venuecity,', '),venuestate)
from venue
where venueseats > 75000
order by venueseats;

concat
---------------------
Denver, CO
Kansas City, MO
East Rutherford, NJ
```

```
Landover, MD
(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

```
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.

Example

The following example shows the 32-bit value for the string 'Amazon Redshift':

```
select crc32('Amazon Redshift');
crc32
----------------------------------
56cbb480
(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

The GET_BIT function extracts a bit from a string.

Syntax

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.

```
GET_BIT(string, offset)
```

Arguments

string
The array that contains the bit.

offset
The position of the bit to return the status for.

Return type

The GET_BIT function returns an INTEGER.

Examples

The following example returns a bit for the given string:

```
select get_bit('Th\\000omas'::bytea, 4);
get_bit
---------
1
(1 row)
```

GET_BYTE function

The GET_BYTE function extracts a byte from a string.

Syntax

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.

```
GET_BYTE(string, offset)
```

Arguments

string
The array that contains the byte

offset
The position of the byte to return the status for.

Return type

The GET_BYTE function returns an INTEGER.

Examples

The following example returns a byte for the given string:

```
select get_byte('Th\\000omas'::bytea, 6);
get_byte
----------
115
(1 row)
```

INITCAP function

Capitalizes the first letter of each word in a specified string. INITCAP has no effect on multi-byte characters.

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, newlines, 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:

```
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.

```
select initcap(catname)
from category
order by catname;

initcap
-----------
Classical
Jazz
Mlb
Mls
Musicals
Nba
Nfl
Nhl
Opera
Plays
Pop
(11 rows)
```

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:

```
select email, initcap(email)
from users
order by userid desc limit 5;

email                             | initcap
----------------------------------+----------------------------------
urna.Ut@egetdictumplacerat.edu  | Urna.Ut@Egetdictumplacerat.Edu
nibh.enim@egestas.ca            | Nibh.Enim@Egestas.Ca
in@Donecat.ca                   | In@Donecat.Ca
sodales@blanditviverraDonec.ca  | Sodales@Blanditviverradonec.Ca
```

```
sociis.natoque.penatibus@vitae.org | Sociis.Natoque.Penatibus@Vitae.Org
(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

```
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:

```
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;

eventid |   eventname    | left_5 | right_5
---------+----------------+--------+---------
1000 | Gypsy          | Gypsy  | Gypsy
1001 | Chicago        | Chica  | icago
1002 | The King and I | The K  | and I
1003 | Pal Joey       | Pal J  |  Joey
1004 | Grease         | Greas  | rease
1005 | Chicago        | Chica  | icago
(6 rows)
```

LEN function

Returns the length of the specified string.

Synopsis

LEN is a synonym of LENGTH function (p. 409), CHAR_LENGTH function (p. 399), CHARACTER_LENGTH function (p. 399), and TEXTLEN function (p. 423).

```
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:

```
select len('cat'), len('cat   ');
len | len
-----+-----
3 |   6
(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. 407)

LOWER function

Converts a string to lower case.

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). LOWER has no effect on multi-byte characters.

Examples

The following example converts the CATNAME field to lower case:

```
select catname, lower(catname) from category order by 1,2;

catname  |   lower
----------+----------
Classical | classical
Jazz      | jazz
MLB       | mlb
MLS       | mls
Musicals  | musicals
NBA       | nba
NFL       | nfl
NHL       | nhl
Opera     | opera
Plays     | plays
Pop       | pop
(11 rows)
```

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
La Cenerentola (Cind
(5 rows)
```

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
2 | 2008-03-05 12:25:29 | 3-05 12:25:29
3 | 2008-11-01 07:35:33 | 11-01 07:35:33
4 | 2008-05-24 01:18:37 | 5-24 01:18:37
5 | 2008-05-17 02:29:11 | 5-17 02:29:11
6 | 2008-08-15 02:08:13 | 15 02:08:13
7 | 2008-11-15 09:38:15 | 11-15 09:38:15
8 | 2008-11-09 05:07:30 | 11-09 05:07:30
9 | 2008-09-09 08:03:36 | 9-09 08:03:36
10 | 2008-06-17 09:44:54 | 6-17 09:44:54
(10 rows)
```

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
(1 row)
```

OCTET_LENGTH function

The OCTET_LENGTH function returns the number of bytes in a binary string.

Syntax

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.

```
OCTET_LENGTH(string)
```

Arguments

string
The input parameter is a binary string.

Return type

The OCTET_LENGTH function returns an INTEGER.

Examples

The following example shows the octet length for the word "Amazon Redshift" :

```
select octet_length('Amazon Redshift');
octet_length
--------------
4
(1 row)
```

POSITION function

Returns the location of the specified substring within a string.

Synonym of the STRPOS function (p. 420) function.

Synopsis

```
POSITION(substring IN string )
```

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:

```
select position('dog' in 'fish');

position
----------
0
(1 row)
```

Examples

The following example shows the position of the string `fish` within the word `dogfish`:

```
select position('fish' in 'dogfish');

position
----------
4
(1 row)
```

The following example returns the number of sales transactions with a COMMISSION over 999.00 from the SALES table:

```
select distinct position('.' in commission), count (position('.' in commission))
from sales where position('.' in commission) > 4 group by position('.' in com
mission)
order by 1,2;
```

```
position | count
----------+-------
5 | 629
(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 single 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:

```
select catid, quote_ident(catname)
from category
order by 1,2;

catid | quote_ident
-------+-------------
1 | "MLB"
2 | "NHL"
3 | "NFL"
4 | "NBA"
5 | "MLS"
6 | "Musicals"
7 | "Plays"
8 | "Opera"
9 | "Pop"
10 | "Jazz"
11 | "Classical"
(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:

```
select quote_literal(catid), catname
from category
order by 1,2;

quote_literal |  catname
--------------+-----------
'1'           | MLB
'10'          | Jazz
'11'          | Classical
'2'           | NHL
'3'           | NFL
'4'           | NBA
'5'           | MLS
'6'           | Musicals
'7'           | Plays
'8'           | Opera
'9'           | Pop
(11 rows)
```

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. 417).

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;

catid | repeat
-------+--------
1 | 111
2 | 222
3 | 333
4 | 444
5 | 555
6 | 666
7 | 777
8 | 888
9 | 999
10 | 101010
11 | 111111
(11 rows)
```

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 search

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:

```
select catid, catgroup,
replace(catgroup, 'Shows', 'Theatre')
from category
order by 1,2,3;

catid | catgroup | replace
------+----------+----------
1 | Sports   | Sports
2 | Sports   | Sports
3 | Sports   | Sports
4 | Sports   | Sports
5 | Sports   | Sports
6 | Shows    | Theatre
7 | Shows    | Theatre
8 | Shows    | Theatre
9 | Concerts | Concerts
10 | Concerts | Concerts
11 | Concerts | Concerts
(11 rows)
```

REPLICATE function

Synonym for the REPEAT function.

See REPEAT function (p. 415).

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

```
REVERSE ( expression )
```

Argument

expression

An expression with a character, datetime, 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:

```
select distinct city as cityname, reverse(cityname)
from users order by city limit 5;

cityname | reverse
----------+----------
Aberdeen | needrebA
Abilene  | enelibA
Ada      | adA
Agat     | tagA
Agawam   | mawagA
(5 rows)
```

Select five sales IDs and their corresponding reversed IDs cast as character strings:

```
select salesid, reverse(salesid)::varchar
from sales order by salesid desc limit 5;

salesid | reverse
---------+---------
172456 | 654271
172455 | 554271
172454 | 454271
172453 | 354271
172452 | 254271
(5 rows)
```

RTRIM function

The RTRIM function trims a specified set of characters from the end of a string.

Synopsis

```
RTRIM( string, 'characters' )
```

Arguments

characters
The first input parameter is a CHAR or VARCHAR string representing the characters to be trimmed from the end of the string.

string
The second parameter is the CHAR or VARCHAR string to be trimmed.

Return type

The RTRIM function returns a character string that is the same data type as the input string (CHAR or VARCHAR).

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;

venueid |         venuename          |          rtrim
---------+----------------------------+-----------------------
1 | Toyota Park               | Toyota
2 | Columbus Crew Stadium     | Columbus Crew Stadium
3 | RFK Stadium               | RFK Stadium
4 | CommunityAmerica Ballpark | CommunityAmerica Ballp
5 | Gillette Stadium          | Gillette Stadium
6 | New York Giants Stadium   | New York Giants Stadium
7 | BMO Field                 | BMO Field
8 | The Home Depot Center     | The Home Depot Cente
9 | Dick's Sporting Goods Park | Dick's Sporting Goods
10 | Pizza Hut Park            | Pizza Hut
(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

The SET_BIT function sets a bit in a string.

Syntax

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.

```
SET_BIT(string, offset, new_value)
```

Arguments

string
The input parameter is a bit expression for which to change the bit.

offset
The position of the bit to be set.

new_value
The value to set the bit to.

Return type

The SET_BIT function returns a BYTEA value.

Examples

The following example sets the bit for the given string:

```
select set_bit('123\\000\\001'::bytea, 4, 1);
set_bit
-------------
```

```
123\000\001
(1 row)
```

SET_BYTE function

The SET_BYTE function sets a byte in a string.

Syntax

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.

```
SET_BYTE(string, offset, new_value)
```

Arguments

string
The input parameter is a byte expression for which to change the byte.

offset
The position of the byte to be set.

new_value
The value to set the byte to.

Return type

The SET_BYTE function returns a BYTEA value.

Examples

The following example sets a byte for the given string:

```
select set_byte('Th\\000omas'::bytea, 4, 41);
set_byte
------------
Th\000o)as
(1 row)
```

STRPOS function

Returns the position of a substring within a specified string.

Synonym of CHARINDEX function (p. 399) and POSITION function (p. 413).

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)
```

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',5,4);
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:

```
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:

```
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
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)
```

TEXTLEN function

Synonym of LEN function.

See LEN function (p. 407).

TO_ASCII function

The TO_ASCII function converts a string from a LATIN1, LATIN2, LATIN9, or WIN1250 encoding to ASCII.

Syntax

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.

```
TO_ASCII(string [, encoding ])
```

Arguments

string
The input parameter is the string for which you want to change the encoding to ASCII.

encoding
The encoding that you want to convert from. LATIN1, LATIN2, LATIN9, and WIN1250 encodings are supported.

Return type

The TO_ASCII function returns an ASCII string.

Examples

The following example converts to ASCII a term with LATIN1 encoding:

```
select to_ascii('español', 'latin1');
to_ascii
----------
espanol
(1 row)
```

TO_HEX function

The TO_HEX function converts a number to its equivalent hexadecimal value.

Syntax

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.

```
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:

```
select to_hex(2147676847);
to_hex
----------
8002f2af
(1 row)
```

TRIM function

The TRIM function trims a specified set of characters from the beginning and end of a string.

Synopsis

```
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 convert the results, and you might need to perform an explicit conversion to avoid a data type mismatch error. See the CAST and CONVERT functions (p. 429) and CONVERT (p. 430) functions for information about explicit conversions.

Example

The following example removes the double quotes that surround the string `"dog"`:

```
select trim('"' FROM '"dog"');

btrim
-------
dog
(1 row)
```

UPPER function

Converts a string to upper case. UPPER has no effect on multi-byte characters.

Synopsis

```
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 upper case:

```
select catname, upper(catname) from category order by 1,2;

catname  |   upper
----------+----------
Classical | CLASSICAL
Jazz      | JAZZ
MLB       | MLB
MLS       | MLS
Musicals  | MUSICALS
NBA       | NBA
NFL       | NFL
NHL       | NHL
Opera     | OPERA
Plays     | PLAYS
Pop       | POP
(11 rows)
```

JSON Functions

Topics

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. 426).

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. 426).

Synopsis

```
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:

```
select json_extract_array_element_text('[111,112,113]', 2);

json_extract_array_element_text
-------------------------------
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. 426).

Synopsis

```
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.

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

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. 137).

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. 119).

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. 119).

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;
```

```
pricepaid
-----------
162
(1 row)
```

```
select pricepaid::integer
from sales where salesid=100;

pricepaid
-----------
162
(1 row)
```

The following query uses the CONVERT function to return the same result:

```
select convert(integer, pricepaid)
from sales where salesid=100;

pricepaid
-----------
162
(1 row)
```

In this example, the values in a timestamp column are cast as dates:

```
select cast(saletime as date), salesid
from sales order by salesid limit 10;

saletime  | salesid
------------+---------
2008-02-18 |       1
2008-06-06 |       2
2008-06-06 |       3
2008-06-09 |       4
2008-08-31 |       5
2008-07-16 |       6
2008-06-26 |       7
2008-07-10 |       8
2008-07-22 |       9
2008-08-06 |      10
(10 rows)
```

In this example, the values in a date column are cast as timestamps:

```
select cast(caldate as timestamp), dateid
from date order by dateid limit 10;

caldate       | dateid
---------------------+--------
2008-01-01 00:00:00 |   1827
2008-01-02 00:00:00 |   1828
2008-01-03 00:00:00 |   1829
2008-01-04 00:00:00 |   1830
2008-01-05 00:00:00 |   1831
```

```
2008-01-06 00:00:00 |   1832
2008-01-07 00:00:00 |   1833
2008-01-08 00:00:00 |   1834
2008-01-09 00:00:00 |   1835
2008-01-10 00:00:00 |   1836
(10 rows)
```

In this example, an integer is cast as a character string:

```
select cast(2008 as char(4));
bpchar
--------
2008
```

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;

salesid |           value
---------+---------------------------
1 | 72800000000000000000000.00
2 |  76000000000000000000000.00
3 | 35000000000000000000000.00
4 | 17500000000000000000000.00
5 | 15400000000000000000000.00
6 | 39400000000000000000000.00
7 | 78800000000000000000000.00
8 | 19700000000000000000000.00
9 | 59100000000000000000000.00
(9 rows)
```

TO_CHAR

TO_CHAR converts a timestamp 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 timestamp type value or a value that can implicitly be coerced to a timestamp.

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. 120).

Note
TO_CHAR does not support 128-bit DECIMAL values.

format
Format for the new value. See Datetime format strings (p. 436) and Numeric format strings (p. 437) 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:

```
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:

```
select starttime, to_char(starttime, 'MON-DD-YYYY HH12:MIPM')
from event where eventid=1;

starttime      |       to_char
---------------------+---------------------
2008-01-25 14:30:00 | JAN-25-2008 02:30PM
(1 row)
```

The following example converts a timestamp literal to a character string:

```
select to_char(timestamp '2009-12-31 23:15:59','HH24:MI:SS');
to_char
----------
23:15:59
(1 row)
```

The following example converts an integer to a character string:

```
select to_char(-125.8, '999D99S');
to_char
```

```
---------
125.80-
(1 row)
```

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:

```
select salesid, pricepaid, commission, (pricepaid - commission)
as difference, to_char(pricepaid - commission, 'rn') from sales
group by sales.pricepaid, sales.commission, salesid
order by salesid limit 10;

salesid | pricepaid | commission | difference |     to_char
---------+-----------+------------+------------+-----------------
1 |    728.00 |     109.20 |     618.80 |           dcxix
2 |     76.00 |      11.40 |      64.60 |             lxv
3 |    350.00 |      52.50 |     297.50 |        ccxcviii
4 |    175.00 |      26.25 |     148.75 |           cxlix
5 |    154.00 |      23.10 |     130.90 |           cxxxi
6 |    394.00 |      59.10 |     334.90 |         cccxxxv
7 |    788.00 |     118.20 |     669.80 |           dclxx
8 |    197.00 |      29.55 |     167.45 |          clxvii
9 |    591.00 |      88.65 |     502.35 |             dii
10 |     65.00 |       9.75 |      55.25 |              lv
(10 rows)
```

The following example adds the currency symbol to the difference values shown in the to_char column:

```
select salesid, pricepaid, commission, (pricepaid - commission)
as difference, to_char(pricepaid - commission, 'l99999D99') from sales
group by sales.pricepaid, sales.commission, salesid
order by salesid limit 10;

salesid | pricepaid | commission | difference |  to_char
---------+-----------+------------+------------+------------
1 |    728.00 |     109.20 |     618.80 | $   618.80
2 |     76.00 |      11.40 |      64.60 | $    64.60
3 |    350.00 |      52.50 |     297.50 | $   297.50
4 |    175.00 |      26.25 |     148.75 | $   148.75
5 |    154.00 |      23.10 |     130.90 | $   130.90
6 |    394.00 |      59.10 |     334.90 | $   334.90
7 |    788.00 |     118.20 |     669.80 | $   669.80
8 |    197.00 |      29.55 |     167.45 | $   167.45
9 |    591.00 |      88.65 |     502.35 | $   502.35
10 |     65.00 |       9.75 |      55.25 | $    55.25
(10 rows)
```

The following example lists the century in which each sale was made.

```
select salesid, saletime, to_char(saletime, 'cc') from sales
order by salesid limit 10;

salesid |      saletime       | to_char
---------+---------------------+---------
1 | 2008-02-18 02:36:48 | 21
```

```
2 | 2008-06-06 05:00:16 | 21
3 | 2008-06-06 08:26:17 | 21
4 | 2008-06-09 08:38:52 | 21
5 | 2008-08-31 09:17:02 | 21
6 | 2008-07-16 11:59:24 | 21
7 | 2008-06-26 12:56:06 | 21
8 | 2008-07-10 02:12:36 | 21
9 | 2008-07-22 02:23:17 | 21
10 | 2008-08-06 02:51:55 | 21
(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
Format of the string to be converted, in terms of its date parts. See Datetime format strings (p. 436) for valid formats.

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. 437).

Return type

TO_NUMBER returns a DECIMAL number.

Examples

The following example converts the string `12,454.8-` to a number:

```
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":

Datepart/timepart	Meaning
BC or B.C., AD or A.D., b.c. or bc, ad or a.d.	Upper and lowercase era indicators
CC	Two-digit century
YYYY, YYY, YY, Y	4-digit, 3-digit, 2-digit, 1-digit year
Y,YYY	4-digit year with comma
IYYY, IYY, IY, I	4-digit, 3-digit, 2-digit, 1-digit ISO year
Q	Quarter number (1 to 4)
MONTH, Month, month	Month name (uppercase, mixed-case, lowercase, blank-padded to 9 characters)
MON, Mon, mon	Abbreviated month name (uppercase, mixed-case, lowercase, blank-padded to 9 characters)
MM	Month number (01-12)
RM, rm	Month in Roman numerals (I-XII; I=January) (uppercase or lowercase)

Datepart/timepart	Meaning
W	Week of month (1-5) (The first week starts on the first day of the month.)
WW	Week number of year (1-53) (The first week starts on the first day of the year.)
IW	ISO week number of year (The first Thursday of the new year is in week 1.)
DAY, Day, day	Day name (uppercase, mixed-case, lowercase, blank-padded to 9 characters)
DY, Dy, dy	Abbreviated day name (uppercase, mixed-case, lowercase, blank-padded to 9 characters)
DDD	Day of year (001-366)
DD	Day of month as a number (01-31)
D	Day of week (1-7; Sunday is 1)
J	Julian Day (days since January 1, 4712 BC)
HH24	Hour (24-hour clock, 00-23)
HH or HH12	Hour (12-hour clock, 01-12)
MI	Minutes (00-59)
SS	Seconds (00-59)
AM or PM, A.M. or P.M., a.m. or p.m., am or pm	Upper and lowercase meridian indicators (for 12-hour clock)

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:

Format	Description
9	Numeric value with the specified number of digits.
0	Numeric value with leading zeros.
. (period), D	Decimal point.
, (comma)	Thousands separator.
CC	Century code. For example, the 21st century started on 2001-01-01 (supported for TO_CHAR only).
PR	Negative value in angle brackets.

Format	Description
S	Sign anchored to a number.
L	Currency symbol in the specified position.
G	Group separator.
MI	Minus sign in the specified position for numbers that are less than 0.
PL	Plus sign in the specified position for numbers that are greater than 0.
SG	Plus or minus sign in the specified position.
RN	Roman numeral between 1 and 3999 (supported for TO_CHAR only).
TH or th	Ordinal number suffix. Does not convert fractional numbers or values that are less than zero.

System administration functions

Topics

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 command in SQL.

Synopsis

```
current_setting('parameter')
```

Argument

parameter
Parameter value to display.

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)
```

SET_CONFIG

Sets a configuration parameter to a new setting.

This function is equivalent to the SET command in SQL.

Synopsis

```
set_config('parameter', 'new_value' , is_local)
```

Parameters

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:

```
select set_config('query_group', 'test', true);

set_config
------------
test
(1 row)
```

System information functions

Topics

Amazon Redshift supports numerous system information functions.

CURRENT_DATABASE

Returns the name of the database where you are currently connected.

Synopsis

```
current_database()
```

Return type

Returns a CHAR or VARCHAR string.

Example

The following query returns the name of the current database:

```
select current_database();

current_database
------------------
tickit
(1 row)
```

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.

```
current_schema()
```

Return type

CURRENT_SCHEMA returns a CHAR or VARCHAR string.

Examples

The following query returns the current schema:

```
select current_schema();

current_schema
----------------
public
(1 row)
```

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:

```
select has_database_privilege('guest', 'tickit', 'temp');

has_database_privilege
------------------------
t
(1 row)
```

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:

```
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.

```
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:

```
select has_table_privilege('guest', 'listing', 'select');

has_table_privilege
---------------------
f
(1 row)
```

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
3 |         3
4 |         0
5 |         1
6 |         2
7 |         3
8 |         0
9 |         1
10 |         2
(10 rows)
```

The following example returns a code (1012) to show that a query without a FROM statement executes on the leader node:

```
select slice_num();
slice_num
-----------
1012
(1 row)
```

USER

Synonym for CURRENT_USER. See CURRENT_USER (p. 442).

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. 118).

```
AES128
AES256
ALL
ALLOWOVERWRITE
ANALYSE
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
CURSOR
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
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
```

```
RIGHT
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

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).

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

Topics

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 save disk space, the STL log tables only maintain log data for approximately one week. If you want to retain the log data, you will need to copy it to other tables or unload it to Amazon S3.

STL_CONNECTION_LOG

STL_CONNECTION_LOG logs authentication attempts and connections and disconnections.

STL_CONNECTION_LOG is superuser visible.

Table columns

Column name	Data type	Description
event	character(50)	Connection or authentication event.
recordtime	timestamp without time zone	Time the event occurred.
remotehost	character(32)	Name or IP address of remote host.
remoteport	character(32)	Port number for remote host.
dbname	character(50)	Database name.

Column name	Data type	Description
username	character(50)	User name.
authmethod	character(32)	Authentication method.
duration	integer	Duration of connection in microseconds.

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;
```

```
           event      |         recordtime         |  remotehost   | username
----------------------+----------------------------+---------------+---------
authentication failure | 2012-10-25 14:41:56.96391 | 10.49.42.138 | jonh

authenticated          | 2012-10-25 14:42:10.87613 | 10.49.42.138 | john

initiating session     | 2012-10-25 14:42:10.87638 | 10.49.42.138 | john

disconnecting session  | 2012-10-25 14:42:19.95992 | 10.49.42.138 | john

(4 rows)
```

STL_DDLTEXT

Query the STL_DDLTEXT table to capture 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. 465), STL_UTILITYTEXT (p. 474), and SVL_STATEMENTTEXT (p. 515). 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.

STL_DDLTEXT is user visible.

Table columns

Column name	Data type	Description
userid	integer	ID of user who generated entry.
xid	bigint	Transaction ID associated with the statement.
pid	integer	Process ID associated with the statement.
label	character(30)	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.
starttime	timestamp without time zone	Exact time when the statement started executing, with 6 digits of precision for fractional seconds. For example: **2009-06-12 11:29:19.131358**
endtime	timestamp without time zone	Exact time when the statement finished executing, with 6 digits of precision for fractional seconds. For example: **2009-06-12 11:29:19.193640**
sequence	integer	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.
text	character(200)	SQL text, in 200-character increments.

Sample queries

The following query shows three entries in the STL_DDLTEXT table for a CREATE VIEW statement and a fourth entry for a CREATE TABLE statement. The TEXT column is truncated here for display purposes.

```
select starttime, sequence, text
from stl_ddltext order by starttime, sequence;

starttime                   | sequence |              text
----------------------------+----------+-----------------------------------------
2009-06-12 11:29:19.131358 |        0 | create view alltickitjoin...
2009-06-12 11:29:19.131358 |        1 | ue on venue.venueid = eve...
2009-06-12 11:29:19.131358 |        2 | .buyerid group by lastnam...
2009-06-16 10:36:50.625097 |        0 | create table test(c1 int)...
...
```

STL_ERROR

The STL_ERROR table 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.

STL_ERROR is user visible.

Table columns

Column name	Data type	Description
userid	integer	ID of user who generated entry.
process	character(12)	Process that threw the exception.
recordtime	timestamp without time zone	Time that the error occurred.
pid	integer	Process ID. The STL_QUERY (p. 463) table contains process IDs and unique query IDs for executed queries.
errcode	integer	Error code corresponding to the error category.
file	character(90)	Name of the source file where the error occurred.
linenum	integer	Line number in the source file where the error occurred.
context	character(100)	Cause of the error.
error	character(512)	Error message.

Sample queries

The following example retrieves the error information for a query with a process ID of 24953:

```
select process, errcode, file, linenum as line,
trim(error) as err
from stl_error
where pid=24953;
```

This query returns the following example output:

```
process  | errcode |     file     | line |  err
---------+---------+--------------+---------+---------------------
diskman2 |       9 | fdisk.cpp    | 1599 | aio_error():Bad file
                                           descriptor
diskman2 |    1001 | fdisk.cpp    |  135 | 'false' - need to handle
                                           redshift exceptions:Not implemented
diskman2 |    1001 | resman.cpp   |  128 | 'false' - need to handle
                                           redshift exceptions:Not implemented
diskman2 |    1008 | shutdown.cpp |  111 | uncaught exception from
                                           an unknown source
(4 rows)
```

STL_EXPLAIN

Use the STL_EXPLAIN table to view the EXPLAIN plan for a query that has been submitted for execution.

STL_EXPLAIN is user visible.

Table columns

Column name	Data type	Description
userid	integer	ID of user who generated entry.
query	integer	Query ID, as tracked in other system tables.
nodeid	integer	Plan node identifier, where a node maps to one or more steps in the execution of the query.
parentid	integer	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.
plannode	character(400)	The node text from the EXPLAIN output. Plan nodes that refer to execution on compute nodes are prefixed with **XN** in the EXPLAIN output.
info	character(400)	Qualifier and filter information for the plan node. For example, join conditions and WHERE clause restrictions are included in this column.

Sample queries

Consider the following EXPLAIN output for an aggregate join query:

```
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)
(6 rows)
```

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:

```
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;

query| nodeid |parentid|           substring            |    substring
-----+--------+--------+--------------------------------+-------------------
10   |      1 |      0 |XN Aggregate  (cost=6717.61..6  |
10   |      2 |      1 |  -> XN Merge Join DS_DIST_NO| Merge Cond:("outer"
10   |      3 |      2 |       -> XN Seq Scan on lis |
```

```
10   |      4 |      2 |       -> XN Seq Scan on sal |
(4 rows)
```

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;

eventid |   sum
--------+----------
    289 | 51846.00
   7895 | 51049.00
   1602 | 50301.00
    851 | 49956.00
   7315 | 49823.00
...
```

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;

query|nodeid|parentid|                          substring
-----+------+--------+-----------------------------------------------------------
15   |    8 |      7 |                               -> XN Seq Scan on eve
15   |    7 |      5 |                          -> XN Hash(cost=87.98..87.9
15   |    6 |      5 |                          -> XN Seq Scan on sales(cos
15   |    5 |      4 |                     -> XN Hash Join DS_DIST_OUTER(cos
15   |    4 |      3 |               -> XN HashAggregate(cost=862286577.07..
15   |    3 |      2 |         -> XN Sort(cost=1000862287175.47..10008622871
15   |    2 |      1 |  -> XN Network(cost=1000862287175.47..1000862287197.
15   |    1 |      0 |XN Merge(cost=1000862287175.47..1000862287197.46 rows=87
(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%';

query|                                      btrim
-----+-------------------------------------------------------------------------
26   | -> XN Window(cost=1000985348268.57..1000985351256.98 rows=170 width=33)
27   | -> XN Window(cost=1000985348268.57..1000985351256.98 rows=170 width=33)
(2 rows)
```

STL_FILE_SCAN

Use the STL_FILE_SCAN table to find out which files 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.

STL_FILE_SCAN is user visible.

Table columns

Column name	Data type	Description
userid	integer	ID of user who generated entry.
query	integer	Query ID. Can be used to join various other system tables and views.
slice	integer	Node slice number.
name	character(90)	Full path and name of the file that was loaded.
lines	bigint	Number of lines read from the file.
bytes	bigint	Number of bytes read from the file.
loadtime	bigint	Amount of time spent loading the file (in microseconds).
curtime	bigint	Timestamp representing the time that Amazon Redshift started processing the file.

Sample queries

The following query retrieves the names and load times of any files that took over 1000000 microseconds for Amazon Redshift to read:

```
select trim(name)as name, loadtime from stl_file_scan
where loadtime > 1000000;
```

This query returns the following example output:

```
           name             | loadtime
----------------------------+----------
 listings_pipe.txt          |  9458354
 allusers_pipe.txt          |  2963761
 allevents_pipe.txt         |  1409135
 tickit/listings_pipe.txt   |  7071087
 tickit/allevents_pipe.txt  |  1237364
 tickit/allusers_pipe.txt   |  2535138
 listings_pipe.txt          |  6706370
 allusers_pipe.txt          |  3579461
 allevents_pipe.txt         |  1313195
 tickit/allusers_pipe.txt   |  3236060
 tickit/listings_pipe.txt   |  4980108
(11 rows)
```

STL_LOAD_COMMITS

Use the STL_LOAD_COMMITS to track or troubleshoot a data load.

This table records the progress of each data file as it is loaded into a database table. As each file successfully loads, the **status** entry in this table changes from 0 to 1.

STL_LOAD_COMMITS is user visible.

Table columns

Column name	Data type	Description
userid	integer	ID of user who generated entry.
query	integer	Query ID associated with the COPY command.
slice	integer	Slice loaded for this entry.
name	character(256)	System-defined value.
filename	character(256)	Name of file being tracked.
byte_offset	integer	This information is for internal use only.
lines_scanned	integer	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.
errors	integer	Running total of errors encountered while loading the file.
curtime	timestamp without time zone	Time that this entry was last updated.
status	integer	**0** if the file has not been loaded yet; **1** if the file has been loaded. (The fact that a file is marked as loaded does not necessarily mean that those rows were committed to the database.)

Sample queries

The following query contains entries for a fresh load of the tables in the TICKIT database:

```
select query, trim(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
 22480 | tickit/category_pipe.txt  | 2013-02-08 20:58:27.333472 |      1
 22482 | tickit/date2008_pipe.txt  | 2013-02-08 20:58:28.608305 |      1
 22485 | tickit/allevents_pipe.txt | 2013-02-08 20:58:29.99489  |      1
 22487 | tickit/listings_pipe.txt  | 2013-02-08 20:58:37.632939 |      1
```

```
 22593 | tickit/allusers_pipe.txt  | 2013-02-08 21:04:08.400491 |      1
 22596 | tickit/venue_pipe.txt     | 2013-02-08 21:04:10.056055 |      1
 22598 | tickit/category_pipe.txt  | 2013-02-08 21:04:11.465049 |      1
 22600 | tickit/date2008_pipe.txt  | 2013-02-08 21:04:12.461502 |      1
 22603 | tickit/allevents_pipe.txt | 2013-02-08 21:04:14.785124 |      1
 22605 | tickit/listings_pipe.txt  | 2013-02-08 21:04:20.170594 |      1
(12 rows)
```

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:

```
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');

 query |           rtrim           |  xid
-------+---------------------------+-------
 22600 | tickit/date2008_pipe.txt  | 68311
 22480 | tickit/category_pipe.txt  | 68066
  7508 | allusers_pipe.txt         | 23365
  7552 | category_pipe.txt         | 23415
  7576 | allevents_pipe.txt        | 23429
  7516 | venue_pipe.txt            | 23390
  7604 | listings_pipe.txt         | 23445
 22596 | tickit/venue_pipe.txt     | 68309
 22605 | tickit/listings_pipe.txt  | 68316
 22593 | tickit/allusers_pipe.txt  | 68305
 22485 | tickit/allevents_pipe.txt | 68071
  7561 | allevents_pipe.txt        | 23429
  7541 | category_pipe.txt         | 23415
  7558 | date2008_pipe.txt         | 23428
 22478 | tickit/venue_pipe.txt     | 68065
   526 | date2008_pipe.txt         |  2572
  7466 | allusers_pipe.txt         | 23365
 22482 | tickit/date2008_pipe.txt  | 68067
 22598 | tickit/category_pipe.txt  | 68310
 22603 | tickit/allevents_pipe.txt | 68315
 22475 | tickit/allusers_pipe.txt  | 68061
   547 | date2008_pipe.txt         |  2572
 22487 | tickit/listings_pipe.txt  | 68072
  7531 | venue_pipe.txt            | 23390
  7583 | listings_pipe.txt         | 23445
(25 rows)
```

STL_LOAD_ERRORS

Use the STL_LOAD_ERRORS table to view logged records of all Amazon Redshift load errors.

STL_LOAD_ERRORS contains a history of all Amazon Redshift load errors. See Load error reference (p. 73) for a comprehensive list of possible load errors and explanations.

STL_LOAD_ERRORS is user visible.

Table columns

Column name	Data type	Description
userid	integer	ID of user who generated entry.
slice	integer	Slice where the error occurred.
tbl	integer	Table ID.
starttime	timestamp without time zone	Start time for the load.
session	integer	Session ID for the session performing the load.
query	integer	Query ID. Can be used to join various other system tables and views.
filename	character(256)	Complete path to the input file for the load.
line_number	bigint	Line number in the load file with the error.
colname	character(127)	Field with the error.
type	character(10)	Data type of the field.
col_length	character(10)	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 "3".
position	integer	Position of the error in the field.
raw_line	character(1024)	Raw load data that contains the error.
raw_field_value	char(1024)	The pre-parsing value for the field "colname" that lead to the parsing error.
err_code	integer	Error code.
err_reason	character(100)	Explanation for the error.

Sample queries

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:

```
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);
```

Next, the following query actually returns the last error that occurred while loading the EVENT table:

```
select table_name, query, line_number, colname,
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_name | query | line_number | colname | error
------+-----+-------+-------------+---------+----------------------
event | 309 |  0 |  5 | Error in Timestamp value or format [%Y-%m-%d %H:%M:%S]
(1 row)
```

STL_LOADERROR_DETAIL

Use the STL_LOADERROR_DETAIL table to view a log of data parse errors that occurred while loading tables into an Amazon Redshift database.

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.

STL_LOADERROR_DETAIL is user visible.

Table columns

Column name	Data type	Description
userid	integer	ID of user who generated entry.
slice	integer	Slice where the error occurred.
session	integer	Session ID for the session performing the load.
query	integer	Query ID. Can be used to join various other system tables and views.
filename	character(256)	Complete path to the input file for the load.
line_number	bigint	Line number in the load file with the error.
field	integer	Field with the error.
colname	character(1024)	Column name.
value	character(1024)	Parsed data value of the field. (May be truncated.)
is_null	integer	Whether or not the parsed value is null.

Column name	Data type	Description
type	character(10)	Data type of the field.
col_length	character(10)	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 "3".

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:

```
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:

```
query  | line_number | value | raw_line | err_reason
-------+-------------+-------+----------+----------------
4      |      3      |  1201 |  1201    | Invalid digit
4      |      3      |   126 |   126    | Invalid digit
4      |      3      |       |   aaa    | Invalid digit
(3 rows)
```

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_QUERY

Use the STL_QUERY table to find out 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.

STL_QUERY is user visible.

Table columns

Column name	Data type	Description
userid	integer	ID of user who generated entry.
query	integer	Query ID. Can be used to join various other system tables and views.
label	character(15)	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.
xid	bigint	Transaction ID.
pid	integer	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. 454) table.
database	character(32)	The name of the database the user was connected to when the query was issued.
querytxt	character(4000)	Actual query text for the query.
starttime	timestamp without time zone	Start time for the query.
endtime	timestamp without time zone	Time that the query completed.
aborted	integer	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.
insert_pristine	integer	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.

Sample queries

To list the five most recent queries against the database, type the following statement:

```
select query, trim(querytxt) as sqlquery
from stl_query
order by query desc limit 5;

query |                                  sqlquery
------+-------------------------------------------------
129 | select query, trim(querytxt) from stl_query order by query;
128 | select node from stv_disk_read_speeds;
127 | select system_status from stv_gui_status
126 | select * from systable_topology order by slice
```

```
125 | load global dict registry
(5 rows)
```

To query time elapsed in descending order for queries that ran on February 15, 2013, type the following statement:

```
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;

 query | date_diff |  sqlquery
-------+-----------+-------------------------------------------
 55    |       119 | redshift_fetch_sample: select count(*) from category
121    |         9 | select * from svl_query_summary;
181    |         6 | select * from svl_query_summary where query in(179,178);
172    |         5 | select * from svl_query_summary where query=148;
...
(189 rows)
```

STL_QUERYTEXT

STL_QUERY_TEXT 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 STL_DDLTEXT and STL_UTILITYTEXT tables. For a complete listing of all statements executed by Amazon Redshift, you can query the SVL_STATEMENTTEXT view.

See also STL_DDLTEXT (p. 453), STL_UTILITYTEXT (p. 474), and SVL_STATEMENTTEXT (p. 515).

STL_QUERYTEXT is user visible.

Table columns

Column name	Data type	Description
userid	integer	ID of user who generated entry.

Column name	Data type	Description
query	integer	Query ID for the statement.
xid	bigint	Transaction ID.
pid	integer	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. 454) table.
sequence	integer	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.
text	character(200)	SQL text, in 200-character increments.

Sample queries

The following query returns the list of COPY commands that were run on the system. The query returns only the first 25 characters of the COPY statement and lists the most recently run statement first (highest-to-lowest query IDs).

```
select query, substring(text,1,25)
from stl_querytext where text like 'copy%'
order by query desc;

 query |         substring
-------+---------------------------
   137 | copy sales from 's3://sam
   134 | copy listing from 's3://s
   131 | copy event from 's3://sam
   128 | copy date from 's3://samp
   125 | copy category from 's3://
...
```

STL_REPLACEMENTS

Use the STL_REPLACEMENTS table to view a log recording when invalid UTF-8 characters were replaced by the COPY (p. 179) 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.

STL_REPLACEMENTS is user visible.

Table columns

Column name	Data type	Description
userid	integer	ID of the user who ran the COPY command.
query	integer	Query ID. Can be used to join various other system tables and views.

Column name	Data type	Description
slice	integer	Node slice number where the replacement occurred.
tbl	integer	Table ID.
starttime	timestamp without time zone	Start time for the COPY command.
session	integer	Session ID for the session performing the COPY command.
filename	character(256)	Complete path to the input file for the COPY command.
line_number	bigint	Line number in the input data file that contained an invalid UTF-8 character.
colname	character(127)	First field that contained an invalid UTF-8 character.
raw_line	character(1024)	Raw load data that contained an invalid UTF-8 character.

STL_S3CLIENT

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.

STL_S3CLIENT is user visible.

Table columns

Column name	Data type	Description
userid	integer	ID of the user who ran the COPY command.
query	integer	Query ID. Can be used to join various other system tables and views.
slice	integer	Node slice number.
recordtime	timestamp without time zone	Time the record is logged.
pid	integer	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.
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.

Column name	Data type	Description
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.

Sample query

The following query returns the time taken to load files using a COPY command.

```
select slice, key, transfer_time
from stl_s3client
where query=7 and http_method='GET';
```

Result

```
slice |  key                         | transfer_time
------+------------------------------+---------------
    0 | simple-parallel/customer5  |       122578
    1 | simple-parallel/customer2  |       125926
    4 | simple-parallel/customer4  |       132787
    5 | simple-parallel/customer1  |       142985
    2 | simple-parallel/customer3  |       187540
    3 | simple-parallel/customer0  |       149612
(6 rows)
```

STL_S3CLIENT_ERROR

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.

STL_S3CLIENT_ERROR is user visible.

Table columns

Column name	Data type	Description
userid	integer	ID of user who generated entry.
query	integer	Query ID. Can be used to join various other system tables and views.
slice	integer	Node slice number.
recordtime	timestamp without time zone	Time the record is logged.
pid	integer	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.
http_method	character(64)	HTTP method name corresponding to the Amazon S3 request.
bucket	character(64)	Amazon S3 bucket name.
key	character(256)	Key corresponding to the Amazon S3 object.
error	character(256)	Error message.

Sample query

The following query returns the last 10 errors from COPY commands.

```
select query, sliceid, substring(key from 1 for 20) as file,
substring(error from 1 for 35) as error
from stl_s3client_error
order by recordtime desc limit 10;
```

Result

```
 query  | sliceid |     file           |             error

--------+---------+--------------------+-----------------------------------
 362228 |      12 | part.tbl.25.159.gz | transfer closed with 1947655 bytes
 362228 |      24 | part.tbl.15.577.gz | transfer closed with 1881910 bytes
 362228 |       7 | part.tbl.22.600.gz | transfer closed with 700143 bytes r
 362228 |      22 | part.tbl.3.34.gz   | transfer closed with 2334528 bytes
 362228 |      11 | part.tbl.30.274.gz | transfer closed with 699031 bytes r
 362228 |      30 | part.tbl.5.509.gz  | Unknown SSL protocol error in conne
 361999 |      10 | part.tbl.23.305.gz | transfer closed with 698959 bytes r
 361999 |      19 | part.tbl.26.582.gz | transfer closed with 1881458 bytes
 361999 |       4 | part.tbl.15.629.gz | transfer closed with 2275907 bytes
 361999 |      20 | part.tbl.6.456.gz  | transfer closed with 692162 bytes r
(10 rows)
```

STL_SESSIONS

Use the STL_SESSIONS table to find out information about user session history for Amazon Redshift.

STL_SESSIONS differs from STV_SESSIONS in that STL_SESSIONS contains session history, whereas STV_SESSIONS contains the current active sessions.

STL_SESSIONS is user visible.

Table columns

Column name	Data type	Description
userid	integer	ID of user who generated entry.
starttime	timestamp without time zone	Time that the session started.
endtime	timestamp without time zone	Time that the session ended.
process	integer	Process ID for the session.
user_name	character(50)	User name associated with the session.
db_name	character(50)	Name of the database associated with the session.

Sample queries

To view session history for the TICKIT database, type the following query:

```
select starttime, process, user_name
from stl_sessions
where db_name='tickit' order by starttime;
```

This query returns the following sample output:

```
    starttime              | process |    user_name
---------------------------+---------+----------------
2008-09-15 09:54:06.746705 |   32358 | dwuser
2008-09-15 09:56:34.30275  |   32744 | dwuser
2008-09-15 11:20:34.694837 |   14906 | dwuser
2008-09-15 11:22:16.749818 |   15148 | dwuser
2008-09-15 14:32:44.66112  |   14031 | dwuser
2008-09-15 14:56:30.22161  |   18380 | dwuser
2008-09-15 15:28:32.509354 |   24344 | dwuser
2008-09-15 16:01:00.557326 |   30153 | dwuser
2008-09-15 17:28:21.419858 |   12805 | dwuser
2008-09-15 20:58:37.601937 |   14951 | dwuser
2008-09-16 11:12:30.960564 |   27437 | dwuser
2008-09-16 14:11:37.639092 |   23790 | dwuser
2008-09-16 15:13:46.02195  |    1355 | dwuser
2008-09-16 15:22:36.515106 |    2878 | dwuser
2008-09-16 15:44:39.194579 |    6470 | dwuser
2008-09-16 16:50:27.02138  |   17254 | dwuser
```

```
2008-09-17 12:05:02.157208 |    8439 | dwuser
(17 rows)
```

STL_STREAM_SEGS

Use the STL_STREAM_SEGS table to list the relationship between streams and concurrent segments.

STL_STREAM_SEGS is user visible.

Table columns

Column name	Data type	Description
userid	integer	ID of user who generated entry.
query	integer	Query ID. Can be used to join various other system tables and views.
stream	integer	The set of concurrent segments of a query.
segment	integer	Query segment that executed.

Sample queries

To view the relationship between streams and concurrent segments for query 10, type the following query:

```
select *
from stl_stream_segs
where query = 10;
```

This query returns the following sample output:

```
 query | stream | segment
-------+--------+---------
    10 |      1 |       2
    10 |      0 |       0
    10 |      2 |       4
    10 |      1 |       3
    10 |      0 |       1
(5 rows)
```

STL_TR_CONFLICT

Use the STL_TR_CONFLICT table 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.

STL_TR_CONFLICT is superuser visible.

Table columns

Column name	Data type	Description
xact_id	bigint	Transaction ID for the rolled back transaction.
process_id	bigint	Process associated with the lock.
xact_start_ts	timestamp without time zone	Timestamp for the transaction start.
abort_time	timestamp without time zone	Time that the transaction was aborted.
table_id	bigint	Table ID for the table where the conflict occurred.

Sample query

For examples of lock conflicts, see Managing concurrent write operations (p. 79). To return information about conflicts that involved a particular table, run a query that specifies the table ID:

```
select * from stl_tr_conflict where table_id=100234
order by xact_start_ts;

xact_id|process_|      xact_start_ts       |        abort_time        |table_
       |id      |                          |                          |id
-------+--------+--------------------------+--------------------------+------
  1876 |  8551  |2010-03-30 09:19:15.852326|2010-03-30 09:20:17.582499|100234
  1928 | 15034  |2010-03-30 13:20:00.636045|2010-03-30 13:20:47.766817|100234
  1991 | 23753  |2010-04-01 13:05:01.220059|2010-04-01 13:06:06.94098 |100234
  2002 | 23679  |2010-04-01 13:17:05.173473|2010-04-01 13:18:27.898655|100234
(4 rows)
```

You can get the table ID from the DETAIL section of the error message for serializability violations (error 1023).

STL_UNDONE

Use the STL_UNDONE table to view information about transactions that have been undone.

STL_UNDONE is user visible.

Table columns

Column name	Data type	Description
userid	integer	ID of user who generated entry.
xact_id	bigint	ID for the undo transaction.
xact_id_undone	bigint	ID for the transaction that was undone.
undo_start_ts	timestamp without time zone	Start time for the undo transaction.

Column name	Data type	Description
undo_end_ts	timestamp without time zone	End time for the undo transaction.
table_id	bigint	ID for the table that was affected by the undo transaction.

Sample query

To view a concise log of all undone transactions, type the following command:

```
select xact_id, xact_id_undone, table_id from stl_undone;
```

This command returns the following sample output:

```
 xact_id | xact_id_undone | table_id
---------+----------------+----------
1344 |           1344 |   100192
1326 |           1326 |   100192
1551 |           1551 |   100192
(3 rows)
```

STL_UNLOAD_LOG

STL_UNLOAD_LOG records the details for a 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.

STL_UNLOAD_LOG is user visible.

Table columns

Column name	Data type	Description
userid	integer	ID of user who generated entry.
query	integer	ID for the undo transaction.
slice	integer	ID for the transaction that was undone.
pid	integer	Start time for the undo transaction.
path	character(1280)	The complete Amazon S3 object path for the file.
start_time	timestamp without time zone	End time for the undo transaction.
end_time	timestamp without time zone	ID for the table that was affected by the undo transaction.
line_count	bigint	Number of lines (rows) unloaded to the file.

Column name	Data type	Description
transfer_size	bigint	Number of bytes transferred.

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:

```
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:

```
 query |             path
-------+--------------------------------------
  2320 | s3://my-bucket/venue0000_part_00
  2320 | s3://my-bucket/venue0001_part_00
  2320 | s3://my-bucket/venue0002_part_00
  2320 | s3://my-bucket/venue0003_part_00
(4 rows)
```

STL_UTILITYTEXT

The STL_UTILITYTEXT table 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. 453), STL_QUERYTEXT (p. 465), and SVL_STATEMENTTEXT (p. 515).

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.

STL_UTILITYTEXT is user visible.

Table columns

Column name	Data type	Description
userid	integer	ID of user who generated entry.
xid	bigint	Transaction ID.
pid	integer	Process ID associated with the query statement.
label	character(30)	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.
starttime	timestamp without time zone	Exact time when the statement started executing, with 6 digits of precision for fractional seconds. For example: **2009-06-12 11:29:19.131358**
endtime	timestamp without time zone	Exact time when the statement finished executing, with 6 digits of precision for fractional seconds. For example: **2009-06-12 11:29:19.193640**
sequence	integer	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.
text	character(200)	SQL text, in 200-character increments.

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:

```
select starttime, sequence, rtrim(text)
from stl_utilitytext
where starttime like '2012-01-26%'
order by starttime, sequence;

starttime          | sequence |              rtrim
---------------------------+-----+---------------------------------
2012-01-26 13:05:52.529235 |   0 | show all;
2012-01-26 13:20:31.660255 |   0 | SET query_group to ''
2012-01-26 13:20:54.956131 |   0 | SET query_group to 'soldunsold.sql'
...
```

STL_VACUUM

Use the STL_VACUUM table to view 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. 294) command description for information about the requirements for running this command.

STL_VACUUM is superuser visible.

Table columns

Column name	Data type	Description
userid	integer	ID of user who generated entry.
xid	bigint	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.
table_id	integer	Table ID.
status	character(30)	**`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.
rows	bigint	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.
sortedrows	integer	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.
blocks	integer	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.
max_merge_partitions	integer	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.)
eventtime	timestamp without time zone	When the vacuum operation started or finished.

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;

xid |table_id| status |  rows  |sortedrows|blocks|   eventtime
----+--------+--------+--------+----------+------+----------------------
1922| 100236 |Started |12319812| 12319812 | 2476 | 2010-05-26 14:08:59...
1922| 100236 |Finished| 9387396|  9387396 | 2120 | 2010-05-26 14:09:10...
1927| 100236 |Started | 9534074|  9387396 | 2120 | 2010-05-26 14:18:25...
1927| 100236 |Finished| 9534074|  9534074 | 2120 | 2010-05-26 14:18:26...
(4 rows)
```

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;

xid |table_id|     status      | rows  |sortedrows|blocks|     eventtime
----+--------+-----------------+-------+----------+------+--------------------
...
2925| 110116 |Started Sort Only|1379648|   172456 |  132 | 2011-02-24 16:25:21...
2925| 110116 |Finished         |1379648|  1379648 |  132 | 2011-02-24 16:26:28...
```

STL_WARNING

Use the STL_WARNING table to view a log of any unexpected occurrences for Amazon Redshift that were not severe enough to cause an error.

STL_WARNING is user visible.

Table columns

Column name	Data type	Description
userid	integer	ID of user who generated entry.
process	character(12)	Process that triggered the warning.
pid	integer	Process ID.
recordtime	timestamp without time zone	Time that the warning occurred.
file	character(20)	Name of the source file where the warning occurred.
linenum	integer	Line number in the source file where the warning occurred.
bug_desc	character(512)	Warning message.

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_WLM_ERROR

The STL_WLM_ERROR table records all WLM-related errors as they occur.

STL_WLM_ERROR is user visible.

Table columns

Column name	Data type	Description
userid	integer	ID of user who generated entry.
recordtime	timestamp without time zone	Time that the error occurred.
pid	integer	ID for the process that generated the error.
error_string	character(256)	Error description.

STL_WLM_QUERY

Contains a record of each attempted execution of a query in a service class handled by WLM.

STL_WLM_QUERY is user visible.

Table columns

Column name	Data type	Description
userid	integer	ID of user who generated entry.
xid	integer	Transaction ID of the query or subquery.
task	integer	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.
query	integer	Query ID. If a query is restarted, the query is assigned a new query ID but not a new task ID.
service_class	integer	ID for the service class. Service classes are defined in the WLM configuration file.
slot_count	integer	Number of WLM query slots.
service_class_start_time	timestamp without time zone	Time that the query was assigned to the service class.
queue_start_time	timestamp without time zone	Time that the query entered the queue for the service class.
queue_end_time	timestamp without time zone	Time when the query left the queue for the service class.
total_queue_time	bigint	Total number of microseconds that the query spent in the queue.
exec_start_time	timestamp without time zone	Time that the query began executing in the service class.
exec_end_time	timestamp without time zone	Time that the query completed execution in the service class.
total_exec_time	bigint	Number of microseconds that the query spent executing.
service_class_end_time	timestamp without time zone	Time that the query left the service class.
final_state	character(16)	End status of the query. Valid values are Completed, Evicted, or Rejected.

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(*),
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:

```
 svc_class | count | avg_queue_time | avg_exec_time
-----------+-------+----------------+---------------
         1 | 20103 |              0 |         80415
         2 |  3421 |          34015 |        234015
         4 |    42 |              0 |        944266
         6 |   196 |           6439 |       1364399
(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(*),
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:

```
 svc_class | count | max_queue_time | max_exec_time
-----------+-------+----------------+---------------
         1 | 20125 |              0 |       6342943
         2 |  3421 |          22143 |      10221439
         4 |    42 |              0 |       3775896
         6 |   197 |          37947 |      16379473
(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:

```
 xid | task | query | svc_class | exec_time |   final_state
-----+------+-------+-----------+-----------+-----------------
1504 |  122 |   122 |         2 | 2001845   | Evicted
1504 |  122 |   123 |         3 | 7738786   | Completed
1509 |  123 |   124 |         2 | 2002352   | Evicted
1509 |  123 |   125 |         3 | 788426    | Completed
1520 |  126 |   128 |         2 | 2001024   | Evicted
1520 |  126 |   129 |         3 | 809977    | Completed
1525 |  127 |   130 |         2 | 2001726   | Evicted
1525 |  127 |   131 |         3 | 5624033   | Completed
...
(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:

```
  xid  | task  | query | svc_class |final_state| total_exec_time
-------+-------+-------+-----------+-----------+-----------------
 69611 | 23039 | 23039 |         2 | Completed |            2481
 69605 | 23037 | 23037 |         2 | Completed |            2503
 69602 | 23036 | 23036 |         2 | Completed |            1646
 69599 | 23035 | 23035 |         2 | Completed |            1225
 68794 | 22766 | 22766 |         2 | Completed |            1306
 67956 | 22439 | 22439 |         2 | Completed |            1555
 67785 | 22382 | 22382 |         2 | Completed |            1033
 67782 | 22381 | 22381 |         2 | Completed |            1733
 67692 | 22351 | 22351 |         2 | Completed |            1788
 67239 | 22200 | 22200 |         2 | Completed |            1161
(10 rows)
```

STV tables for snapshot data

Topics

STV tables are actually virtual system tables that contain snapshots of the current system data.

STV_ACTIVE_CURSORS

STV_ACTIVE_CURSORS displays details for currently open cursors. For more information, see DECLARE (p. 216).

STV_ACTIVE_CURSORS is user visible. A user can only view cursors opened by that user. A superuser can view all cursors.

Table columns

Column name	Data type	Description
userid	integer	ID of user who generated entry.
name	character(256)	Cursor name.
xid	bigint	Transaction context.
pid	integer	Leader process running the query.
starttime	timestamp without time zone	Time when the cursor was declared.

Column name	Data type	Description
row_count	bigint	Number of rows in the cursor result set.
byte_count	bigint	Number of bytes in the cursor result set.
fetched_rows	bigint	Number of rows currently fetched from the cursor result set.

STV_BLOCKLIST

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. 490) to view summary information about disk utilization.

STV_BLOCKLIST is superuser visible.

Table columns

Column name	Data type	Description
slice	integer	Data slice.
col	integer	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.
tbl	integer	Table ID for the database table.
blocknum	integer	ID for the data block.
num_values	integer	Number of values contained on the block.
extended_limits	integer	For internal use.
minvalue	bigint	Minimum data value of the block. Stores first eight characters as 64-bit integer for non-numeric data. Used for disk scanning.
maxvalue	bigint	Maximum data value of the block. Stores first eight characters as 64-bit integer for non-numeric data. Used for disk scanning.
sb_pos	integer	Internal Amazon Redshift identifier for super block position on the disk.
pinned	integer	Whether or not the block is pinned into memory as part of pre-load. 0 = false; 1 = true. Default is false.
on_disk	integer	Whether or not the block is automatically stored on disk. 0 = false; 1 = true. Default is false.
modified	integer	Whether or not the block has been modified. 0 = false; 1 = true. Default is false.

Column name	Data type	Description
hdr_modified	integer	Whether or not the block header has been modified. 0 = false; 1 = true. Default is false.
unsorted	integer	Whether or not a block is unsorted. 0 = false; 1 = true. Default is true.
tombstone	integer	Whether or not a block is tombstoned. 0 = false; 1 = true. Default is false.
preferred_diskno	integer	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.
temporary	integer	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.
newblock	integer	Indicates whether or not a block is new (true) or was never committed to disk (false). 0 = false; 1 = true.
num_readers	integer	Number of references on each block.
flags	integer	Internal Amazon Redshift flags for the block header.

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. 505) 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:

```
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:

```
 col | count
-----+-------
   0 |  4
   1 |  4
   2 |  4
   3 |  4
   4 |  4
   5 |  4
   7 |  4
   8 |  4
(8 rows)
```

The following query shows whether or not table data is actually distributed over all slices:

```
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 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   | slice | rows
----------+-------+-------
listing  |    13 | 10527
listing  |    14 | 10526
listing  |     8 | 10526
listing  |     9 | 10526
listing  |     7 | 10525
listing  |     4 | 10525
listing  |    17 | 10525
listing  |    11 | 10525
listing  |     5 | 10525
listing  |    18 | 10525
listing  |    12 | 10525
listing  |     3 | 10525
listing  |    10 | 10525
listing  |     2 | 10524
listing  |    15 | 10524
listing  |    16 | 10524
listing  |     6 | 10524
listing  |    19 | 10524
listing  |     1 | 10523
listing  |     0 | 10521
...
(180 rows)
```

The following query determines whether any tombstoned blocks were committed to disk:

```
select slice, col, tbl, blocknum, newblock
from stv_blocklist
where  tombstone > 0;

slice | col |   tbl  | blocknum | newblock
-------+-----+--------+----------+----------
4     |  0  | 101285 |    0     |   1
4     |  2  | 101285 |    0     |   1
4     |  4  | 101285 |    1     |   1
5     |  2  | 101285 |    0     |   1
5     |  0  | 101285 |    0     |   1
5     |  1  | 101285 |    0     |   1
5     |  4  | 101285 |    1     |   1
```

```
...
(24 rows)
```

STV_CURSOR_CONFIGURATION

STV_CURSOR_CONFIGURATION displays cursor configuration constraints. For more information, see DECLARE (p. 216).

STV_CURSOR_CONFIGURATION is superuser visible.

Table columns

Column name	Data type	Description
maximum_cursor_count	integer	Maximum number of cursors allowed concurrently.
maximum_cursor_size	integer	Maximum size of an individual cursor result set, in megabytes.

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

Column name	Data type	Description
userid	integer	ID of user who generated entry.
query	integer	Query ID. Can be used to join various other system tables and views.
slice	integer	Data slice where the step executed.
segment	integer	Segment of the query that executed. A query segment is a series of steps.
step	integer	Step of the query segment that executed. A step is the smallest unit of query execution.
starttime	timestamp without time zone	Time that the step executed.
currenttime	timestamp without time zone	Current time.
tasknum	integer	Query task process that is assigned to the execute the step.

Column name	Data type	Description
rows	bigint	Number of rows processed.
bytes	bigint	Number of bytes processed.
label	character	Step name.
is_diskbased	char(1)	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.
workmem	bigint	Number of bytes of working memory assigned to the step.
num_parts	integer	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.
is_rrscan	char(1)	If true (**t**), indicates that range-restricted scan was used on the step. Default is false (**f**).
is_delayed_scan	char(1)	If true (**t**), indicates that delayed scan was used on the step. Default is false (**f**).

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. 513) or SVV_QUERY_STATE (p. 511) 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

Column name	Data type	Description
userid	integer	ID of user who generated entry.
slice	integer	Slice where the query is running.
query	integer	Query ID. Can be used to join various other system tables and views.

Column name	Data type	Description
label	character(30)	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.
xid	bigint	Transaction ID.
pid	integer	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. 454) table.
starttime	timestamp without time zone	Time that the query started.
text	character(100)	Query text, truncated to 100 characters if the statement exceeds that limit.
suspended	integer	Whether the query is suspended or not. 0 = false; 1 = true.
insert_pristine	integer	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.

Sample queries

To view all active queries currently running on the database, type the following query:

```
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`:

```
select slice, query, trim(label) querylabel, pid,
starttime, substring(text,1,20) querytext
from stv_inflight;

slice|query|querylabel | pid |        starttime         |      querytext
-----+-----+-----------+-----+--------------------------+--------------------
1011 |  21 |           | 646 |2012-01-26 13:23:15.645503|select slice, query,
1011 |  20 |avgwait.sql| 499 |2012-01-26 13:23:14.159912|select avg(datediff(
(2 rows)
```

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

Column name	Data type	Description
userid	integer	ID of user who generated entry.
session	integer	Session PID of process doing the load.
query	integer	Query ID. Can be used to join various other system tables and views.
slice	integer	Node slice number.
pid	integer	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.
recordtime	timestamp without time zone	Time the record is logged.
bytes_to_load	bigint	Total number of bytes to be loaded by this slice. This is 0 if the data being loaded is compressed
bytes_loaded	bigint	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.
bytes_to_load_compressed	bigint	Total number of bytes of compressed data to be loaded by this slice. This is 0 if the data being loaded is not compressed.
bytes_loaded_compressed	bigint	Number of bytes of compressed data loaded by this slice. This is 0 if the data being loaded is not compressed.
lines	integer	Number of lines loaded by this slice.
num_files	integer	Number of files to be loaded by this slice.
num_files_complete	integer	Number of files loaded by this slice.
current_file	character(256)	Name of the file being loaded by this slice.
pct_complete	integer	Percentage of data load completed by this slice.

Sample query

To view the progress of each slice for a COPY command, type the follow query. In this example, the query ID of the COPY command is 210.

```
select slice , bytes_loaded, bytes_to_load , pct_complete from stv_load_state
where query = 210;

 slice | bytes_loaded | bytes_to_load | pct_complete
-------+--------------+---------------+--------------
     2 |            0 |             0 |            0
     3 |     12840898 |      39104640 |           32
(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. 471) table to see a log of lock conflicts.

STV_LOCKS is superuser visible.

Table columns

Column name	Data type	Description
table_id	bigint	Table ID for the table acquiring the lock.
last_commit	timestamp without time zone	Timestamp for the last commit in the table.
last_update	timestamp without time zone	Timestamp for the last update for the table.
lock_owner	bigint	Transaction ID associated with the lock.
lock_owner_pid	bigint	Process ID associated with the lock.
lock_owner_start_ts	timestamp without time zone	Timestamp for the transaction start time.
lock_owner_end_ts	timestamp without time zone	Timestamp for the transaction end time.
lock_status	character (22)	Status of the process either waiting for or holding a lock.

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

Row name	Data type	Description
owner	integer	Disk node that owns the partition.
host	integer	Node that is physically attached to the partition.
diskno	integer	Disk containing the partition.
part_begin	bigint	Offset of the partition. Raw devices are logically partitioned to open space for mirror blocks.
part_end	bigint	End of the partition.
used	integer	Number of 1 MB disk blocks currently in use on the partition.
tossed	integer	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.
capacity	integer	Total capacity of the partition in 1 MB disk blocks.
reads	bigint	Number of reads that have occurred since the last Amazon Redshift xstart.
writes	bigint	Number of writes that have occurred since the last Amazon Redshift xstart.
seek_forward	integer	Number of times that a request is not for the subsequent address given the previous request address.
seek_back	integer	Number of times that a request is not for the previous address given the subsequent address.
is_san	integer	Whether the partition belongs to a SAN. Valid values are **0** (false) or **1** (true).
failed	integer	Whether the device has been marked as failed. Valid values are **0** (false) or **1** (true).
mbps	integer	Disk speed in megabytes per second.
mount	character(256)	Directory path to the device.

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.

```
select owner, host, diskno, used, capacity,
(used-tossed)/capacity::numeric *100 as pctused
from stv_partitions order by owner;

 owner | host | diskno |  used  | capacity | pctused
-------+------+--------+--------+----------+---------
   0   |  0   |    0   | 236480 |  949954  | 24.9
   0   |  0   |    1   | 236420 |  949954  | 24.9
   0   |  0   |    2   | 236440 |  949954  | 24.9
   0   |  1   |    2   | 235150 |  949954  | 24.8
   0   |  1   |    1   | 237100 |  949954  | 25.0
   0   |  1   |    0   | 237090 |  949954  | 25.0
   1   |  1   |    0   | 236310 |  949954  | 24.9
   1   |  1   |    1   | 236300 |  949954  | 24.9
   1   |  1   |    2   | 236320 |  949954  | 24.9
   1   |  0   |    2   | 237910 |  949954  | 25.0
   1   |  0   |    1   | 235640 |  949954  | 24.8
   1   |  0   |    0   | 235380 |  949954  | 24.8

(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

Column name	Data type	Description
userid	integer	ID of user who generated entry.
status	character(20)	Query status. Valid values are **`Running`**, **`Done`**.
starttime	timestamp without time zone	Time that the query started.
duration	integer	Number of microseconds since the session started.
user_name	character(50)	User name who ran the process.
db_name	character(50)	Name of the database.

Column name	Data type	Description
query	character(600)	Query text, up to 600 characters. Any additional characters are truncated.
pid	integer	Process ID for the session associated with the query, which is always -1 for queries that have completed.

Sample queries

To determine what queries are currently running against the database, type the following query:

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

The sample output below shows a single query running on the TICKIT database:

```
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:

```
select * from stv_recents where status<>'Done';

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:

```
select * from stv_recents where status<>'Done'
and pid not in (select pid from stv_inflight);
...
```

STV_SLICES

Use the STV_SLICES table to view the current mapping of a slice to a node.

The information in STV_SLICES is used mainly for investigation purposes.

STV_SLICES is superuser visible.

Table columns

Column name	Data type	Description
node	integer	Cluster node where the slice is located.
slice	integer	Data slice.

Sample query

To view which cluster nodes are managing which slices, type the following query:

```
select * from stv_slices;
```

This query returns the following sample output:

```
 node | slice
------+-------
    0 |     2
    0 |     3
    0 |     1
    0 |     0
(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. 470) table instead of STV_SESSIONS.

STV_SESSIONS is user visible.

Table columns

Column name	Data type	Description
starttime	timestamp without time zone	Time that the session started.
process	integer	Process ID for the session.
user_name	character(50)	User associated with the session.
db_name	character(50)	Name of the database associated with the session.

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:

```
select *
from stv_sessions;
```

The sample query output below shows three active sessions currently running on Amazon Redshift:

```
starttime       | process |user_name | db_name
-----------------------+---------+----------+----------------
2008-08-06 08:54:20.50 |   19829 | dwuser | dev
2008-08-06 08:56:34.50 |   20279 | dwuser | dev
2008-08-06 08:55:00.50 |   19996 | dwuser | tickit
(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. 497), which contains information about transient database tables that the system creates during query processing.

STV_TBL_PERM is superuser visible.

Table columns

Column name	Data type	Description
slice	integer	Data slice allocated to the table.
id	integer	Table ID.
name	character(72)	Table name.
rows	bigint	Number of data rows in the slice.
sorted_rows	bigint	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.
temp	integer	Whether or not the table is a temporary table. 0 = false; 1 = true.
db_id	integer	ID of the database where the table was created.

Sample queries

The following query returns a list of distinct table IDs and names:

```
select distinct id, name
from stv_tbl_perm order by name;

id   |          name
--------+------------------------
100571 | category
100575 | date
100580 | event
100596 | listing
100003 | redshift_config_harvest
100612 | sales
...
```

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:

```
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:

```
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;

tablename | sum
-----------+-----
venue     | 202
(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;

tablename | sum
-----------+-----
venue     | 200
(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. 495) in that STV_TBL_PERM contains information about permanent database tables.

STV_TBL_TRANS is superuser visible.

Table columns

Column name	Data type	Description
slice	integer	Data slice allocated to the table.
id	integer	Table ID.
rows	bigint	Number of data rows in the table.
size	bigint	Number of bytes allocated to the table.
query_id	bigint	Query ID.
ref_cnt	integer	Number of references.
from_suspended	integer	Whether or not this table was created during a query that is now suspended.
prep_swap	integer	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.)

Sample queries

To view transient table information for a query with a query ID of 90, type the following command:

```
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:

```
slice | id | rows | size | query_ | ref_ | from_     | prep_
      |    |      |      | id     | cnt  | suspended | swap
------+----+------+------+--------+------+-----------+-------
 1013 | 95 |    0 |    0 |     90 |    4 |         0 |   0
    7 | 96 |    0 |    0 |     90 |    4 |         0 |   0
   10 | 96 |    0 |    0 |     90 |    4 |         0 |   0
   17 | 96 |    0 |    0 |     90 |    4 |         0 |   0
   14 | 96 |    0 |    0 |     90 |    4 |         0 |   0
    3 | 96 |    0 |    0 |     90 |    4 |         0 |   0
 1013 | 99 |    0 |    0 |     90 |    4 |         0 |   0
    9 | 96 |    0 |    0 |     90 |    4 |         0 |   0
    5 | 96 |    0 |    0 |     90 |    4 |         0 |   0
   19 | 96 |    0 |    0 |     90 |    4 |         0 |   0
    2 | 96 |    0 |    0 |     90 |    4 |         0 |   0
 1013 | 98 |    0 |    0 |     90 |    4 |         0 |   0
   13 | 96 |    0 |    0 |     90 |    4 |         0 |   0
    1 | 96 |    0 |    0 |     90 |    4 |         0 |   0
 1013 | 96 |    0 |    0 |     90 |    4 |         0 |   0
    6 | 96 |    0 |    0 |     90 |    4 |         0 |   0
   11 | 96 |    0 |    0 |     90 |    4 |         0 |   0
   15 | 96 |    0 |    0 |     90 |    4 |         0 |   0
   18 | 96 |    0 |    0 |     90 |    4 |         0 |   0
```

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

Column name	Data type	Description
id	integer	Service class ID.
condition	character(128)	Query type. Valid query types are read, write, system, or any.
action_seq	integer	Zero-based index specifying the action position in the order of actions.

Column name	Data type	Description
action	character(64)	Specifies whether the condition assigns a service class or rejects (evicts) the query.
action_service_class	integer	The service class where the action takes place. The query is either assigned to this service class or evicted from the class.

Sample query

The following query returns the current classification rules.

```
select id, trim(condition), action_seq,
trim(action), action_service_class
from stv_wlm_classification_config
order by 1;

 id |        btrim        | action_seq | btrim  | action_service_class
----+---------------------+------------+--------+----------------------
  1 | (querytype: system) |          0 | assign |                    1
  2 | (querytype: read)   |          0 | assign |                    2
  2 | (querytype: read)   |          1 | assign |                    3
  3 | (querytype: write)  |          0 | assign |                    3
(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

Column name	Data type	Description
service_class	integer	ID for the service class. Service classes are defined in the WLM configuration.
position	integer	Position of the query in the queue. The query with the smallest **`position`** value runs next.
task	integer	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.
query	integer	Query ID. If a query is restarted, the query is assigned a new query ID but not a new task ID.
slot_count	integer	Number of WLM query slots.
start_time	timestamp without time zone	Time that the query entered the queue.
queue_time	bigint	Number of microseconds that the query has been in the queue.

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:

```
 service_class | position | task | query | start_time    | queue_time
---------------+----------+------+-------+---------------+------------
3 |        0 |  455 |   476 | 2010-10-06 13:18:24.065838 |   20937257
3 |        1 |  456 |   478 | 2010-10-06 13:18:26.652906 |   18350191
(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

Column name	Data type	Description
xid	integer	Transaction ID of the query or subquery.
task	integer	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.
query	integer	Query ID. If a query is restarted, the query is assigned a new query ID but not a new task ID.
service_class	integer	ID for the service class. Service classes are defined in the WLM configuration.
slot_count	integer	Number of WLM query slots.
wlm_start_time	timestamp without time zone	Time that the query entered the system table queue or short query queue.
state	character(16)	Current state of the query or subquery.

Column name	Data type	Description
queue_time	bigint	Number of microseconds that the query has spent in the queue.
exec_time	bigint	Number of microseconds that the query has been executing.

Sample query

The following query returns records for all currently executing queries:

```
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

Column name	Data type	Description
service_class	integer	ID for the service class. Service classes are defined in the WLM configuration.
task	integer	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.
query	integer	Query ID. If a query is restarted, the query is assigned a new query ID but not a new task ID.
slot_count	integer	Number of WLM query slots.
start_time	timestamp without time zone	Time that the query began executing.
exec_time	bigint	Number of microseconds that the query has been executing.

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:

```
select * from stv_wlm_query_task_state
where service_class=3;
```

This query returns the following sample output:

```
service_class | task | query |         start_time         | exec_time
--------------+------+-------+----------------------------+-----------
    3         |  466 |   491 | 2010-10-06 13:29:23.063787 | 357618748
(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

Column name	Data type	Description
service_class	integer	ID for the service class. Service classes are defined in the WLM configuration.
queueing_strategy	character(32)	Queueing strategy used by the service class. Valid value is **`FIFO Queue Policy`** (first in, first out).
num_query_tasks	integer	Number of concurrent tasks that can run in the service class.
evictable	character(8)	Indicates whether or not the queries in the service class are evictable. Valid values are **`true`** or **`false`**.
eviction_threshold	bigint	Number of microseconds that the query can execute before being evicted and reassigned to the next service class.
query_working_mem	integer	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
min_step_mem	integer	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.

Column name	Data type	Description
name	character(64)	Comment about the state of the service class.
max_execution_time	bigint	Number of microseconds that the query can execute before being terminated.
user_group_wild_card	Boolean	If TRUE, the WLM queue treats an asterisk (*) as a wildcard character in user group strings in the WLM configuration. The default is FALSE.
query_group_wild_card	Boolean	If TRUE, the WLM queue treats an asterisk (*) as a wildcard character in query group strings in the WLM configuration. The default is FALSE.

Sample query

The following query displays the current service class configuration:

```
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:

```
service_|num_query_|evictable|eviction_| name
class   |tasks     |         |threshold|
--------+----------+---------+---------+--------------------------------
      1 |        1 | false   |        0| System table query service class
      2 |        3 | true    |  2000000| short read service class
      3 |        1 | false   |        0| write  long read service class
(3 rows)
```

STV_WLM_SERVICE_CLASS_STATE

Contains the current state of the service classes.

STV_WLM_SERVICE_CLASS_STATE is superuser visible.

Table columns

Column name	Data type	Description
service_class	integer	ID for the service class. Service classes are defined in the WLM configuration.
num_queued_queries	integer	Number of queries currently in the queue.
num_executing_queries	integer	Number of queries currently executing.
num_serviced_queries	integer	Number of queries that have ever been in the service class.

Column name	Data type	Description
num_executed_queries	integer	Number of queries to have executed since Amazon Redshift was initialized.
num_restarted_queries	integer	Number of queries that have restarted since Amazon Redshift was initialized.

Sample query

The following query shows the number of queries that are either executing or have finished executing by service class.

```
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:

```
 service_class | num_executing_queries | num_executed_queries
---------------+-----------------------+----------------------
             1 |                     1 |                  222
             2 |                     0 |                  135
             3 |                     1 |                   39
(3 rows)
```

System views

Topics

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 contain 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

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_PARTITIONS (p. 490) to view summary information about disk utilization.

SVV_DISKUSAGE is superuser visible.

Table rows

Row name	Data type	Description
db_id	integer	Database ID.
name	character(72)	Table name.
slice	integer	Data slice allocated to the table.
col	integer	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.
tbl	integer	Table ID.
blocknum	integer	ID for the data block.
num_values	integer	Number of values contained on the block.
minvalue	bigint	Minimum value contained on the block.
maxvalue	bigint	Maximum value contained on the block.
sb_pos	integer	Internal identifier for the position of the super block on disk.
pinned	integer	Whether or not the block is pinned into memory as part of pre-load. 0 = false; 1 = true. Default is false.
on_disk	integer	Whether or not the block is automatically stored on disk. 0 = false; 1 = true. Default is false.
modified	integer	Whether or not the block has been modified. 0 = false; 1 = true. Default is false.
hdr_modified	integer	Whether or not the block header has been modified. 0 = false; 1 = true. Default is false.
unsorted	integer	Whether or not a block is unsorted. 0 = false; 1 = true. Default is true.
tombstone	integer	Whether or not a block is tombstoned. 0 = false; 1 = true. Default is false.

Row name	Data type	Description
preferred_diskno	integer	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.
temporary	integer	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.
newblock	integer	Indicates whether or not a block is new (true) or was never committed to disk (false). 0 = false; 1 = true.

Sample queries

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):

```
select db_id, trim(name) as tablename, max(blocknum)
from svv_diskusage
where name='users' and col=6
group by db_id, name;

db_id  | tablename | max
--------+-----------+-----
175857 | users     |   2
(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.)

```
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;

db_id  | tablename  | col |  tbl   | max
--------+------------+-----+--------+-----
175857 | salesnew   |   0 | 187605 | 154
175857 | salesnew   |   1 | 187605 | 154
175857 | salesnew   |   2 | 187605 | 154
175857 | salesnew   |   3 | 187605 | 154
175857 | salesnew   |   4 | 187605 | 154
175857 | salesnew   |   5 | 187605 |  79
175857 | salesnew   |   6 | 187605 |  79
175857 | salesnew   |   7 | 187605 | 302
175857 | salesnew   |   8 | 187605 | 302
175857 | salesnew   |   9 | 187605 | 302
175857 | salesnew   |  10 | 187605 |   3
175857 | salesnew   |  11 | 187605 |   2
175857 | salesnew   |  12 | 187605 | 296
(13 rows)
```

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. 463) 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

Column name	Data type	Description
userid	integer	ID of user who generated entry.
query	integer	Query ID. Can be used to join various other system tables and views.
xid	bigint	Transaction ID.
pid	integer	Process ID associated with the query.
starttime	timestamp without time zone	Exact time when the statement started executing, with 6 digits of precision for fractional seconds. For example: **`2009-06-12 11:29:19.131358`**
endtime	timestamp without time zone	Exact time when the statement finished executing, with 6 digits of precision for fractional seconds. For example: **`2009-06-12 11:29:19.193640`**
elapsed	bigint	Length of time that it took the query to execute (in microseconds).
aborted	integer	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.
label	character(30)	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.
substring	character(60)	Truncated query text.

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`.

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

 query  |  pid  | elapsed  |           substring
--------+-------+----------+-----------------------------------------------
 187752 | 18921 | 18465685 | select query, elapsed, substring from svl_...
 204168 |  5117 |    59603 | insert into testtable values (100);
 187561 | 17046 |  1003052 | select * from pg_table_def where tablename...
 187549 | 17046 |  1108584 | select * from STV_WLM_SERVICE_CLASS_CONFIG
 187468 | 17046 |  5670661 | select * from pg_table_def where schemaname...
(5 rows)
```

The following example returns the SQL script name (LABEL column) and elapsed time for a query that was cancelled (**`aborted=1`**):

```
select query, elapsed, label
from svl_qlog where aborted=1;

query | elapsed |             label
------+---------+--------------------------------
16 | 6935292 | alltickittablesjoin.sql
(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. 487) to STL_QUERYTEXT (p. 465).

SVV_QUERY_INFLIGHT is user visible.

Table columns

Column name	Data type	Description
userid	integer	ID of user who generated entry.
slice	integer	Slice where the query is running.
query	integer	Query ID. Can be used to join various other system tables and views.
pid	integer	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. 454) table.
starttime	timestamp without time zone	Time that the query started.
suspended	integer	Whether the query is suspended: **`0`** = false; **`1`** = true.
text	character(200)	Query text, in 200-character increments.

Column name	Data type	Description
sequence	integer	Sequence number for segments of query statements.

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.)

```
select slice, query, pid, starttime, suspended, trim(text) as statement, sequence
from svv_query_inflight
order by query, sequence;

slice|query| pid  |      starttime       |suspended| statement | sequence
-----+-----+------+----------------------+---------+-----------+---------
1012 | 428 | 1658 | 2012-04-10 13:53:... |       0 | select ...|    0
1012 | 428 | 1658 | 2012-04-10 13:53:... |       0 | enueid ...|    1
1012 | 428 | 1658 | 2012-04-10 13:53:... |       0 | atname,...|    2
1012 | 429 | 1608 | 2012-04-10 13:53:... |       0 | select ...|    0
(4 rows)
```

SVL_QUERY_REPORT

Amazon Redshift creates the SVL_QUERY_REPORT view from a UNION of a number of internal STL 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.

Table rows

Row name	Data type	Description
userid	integer	ID of user who generated entry.
query	integer	Query ID. Can be used to join various other system tables and views.
slice	integer	Data slice where the step executed.
segment	integer	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.
step	integer	Query step that executed.
start_time	timestamp without time zone	Exact time when the started executing, with 6 digits of precision for fractional seconds. For example: **`2012-12-12 11:29:19.131358`**

Row name	Data type	Description
end_time	timestamp without time zone	Exact time when the started executing, with 6 digits of precision for fractional seconds. For example: **2012-12-12 11:29:19.131467**
elapsed_time	bigint	Time (in microseconds) that it took the step to execute.
rows	bigint	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.
bytes	bigint	Number of bytes produced by the step (per slice).
label	text	Step label, which consists of a query step name and in many cases a table ID (for example, `scan tbl=100448`).
is_diskbased	character(1)	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.
workmem	bigint	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
is_rrscan	character(1)	If true (**t**), indicates that range-restricted scan was used on the step. Default is false (**f**).
is_delayed_scan	character(1)	If true (**t**), indicates that delayed scan was used on the step. Default is false (**f**).

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:

```
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:

```
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

Column name	Data type	Description
userid	integer	ID of user who generated entry.
query	integer	Query ID. Can be used to join various other system tables and views.
seg	integer	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.
step	integer	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.
maxtime	interval	Maximum amount of time (in microseconds) for this step to execute.
avgtime	interval	Average time (in microseconds) for this step to execute.
rows	bigint	Number of rows produced by the step that is executing.
bytes	bigint	Number of bytes produced by the step that is executing.
cpu	bigint	The percentage of CPU used by the query processes in the current stream. (Time is measured in microseconds.)
memory	bigint	The current amount of shared memory used by the current stream (in bytes).

Column name	Data type	Description
rate_row	double precision	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.
rate_byte	double precision	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.
label	character(25)	Query label: a name for the step, such as `scan` or `sort`.
is_diskbased	character(1)	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.
workmem	bigint	Amount of working memory (in bytes) assigned to the query step.
num_parts	integer	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.
is_rrscan	character(1)	If true (**t**), indicates that range-restricted scan was used on the step. Default is false (**f**).
is_delayed_scan	character(1)	If true (**t**), indicates that delayed scan was used on the step. Default is false (**f**).

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:

```
select query, seg, step, maxtime, avgtime, rows, label
from svv_query_state
where query = 279
order by query, seg, step;
```

This query retrieves the processing information about query 279, as shown in the following sample output:

```
query |   seg   | step | maxtime | avgtime |  rows   | label
------+---------+------+---------+---------+---------+-------------------
279 |       3 |    0 | 1658054 | 1645711 | 1405360 | scan
279 |       3 |    1 | 1658072 | 1645809 |       0 | project
279 |       3 |    2 | 1658074 | 1645812 | 1405434 | insert
279 |       3 |    3 | 1658080 | 1645816 | 1405437 | distribute
279 |       4 |    0 | 1677443 | 1666189 | 1268431 | scan
279 |       4 |    1 | 1677446 | 1666192 | 1268434 | insert
```

```
279 |       4 |    2 | 1677451 | 1666195 |       0 | aggr
(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:

```
select query, label, is_diskbased from svv_query_state
where is_diskbased = 't';
```

This sample output shows any active queries currently running on disk:

```
 query | label        | is_diskbased
-------+--------------+--------------
1025   | hash tbl=142 |      t
(1 row)
```

SVL_QUERY_SUMMARY

Use the SVL_QUERY_SUMMARY view to find out 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

Column name	Data type	Description
userid	integer	ID of user who generated entry.
query	integer	Query ID. Can be used to join various other system tables and views.
stm	integer	Stream: A set of concurrent segments in a query. A query has one or more streams.
seg	integer	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.
step	integer	Query step that executed.
maxtime	bigint	Maximum amount of time for the step to execute (in microseconds).
avgtime	bigint	Average time for the step to execute (in microseconds).
rows	bigint	Number of data rows involved in the query step.

Column name	Data type	Description
bytes	bigint	Number of data bytes involved in the query step.
rate_row	double precision	Query execution rate per row.
rate_byte	double precision	Query execution rate per byte.
label	text	Step label, which consists of a query step name and in many cases a table ID (for example, `scan tbl=100448`). 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
is_diskbased	character(1)	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.
workmem	bigint	Amount of working memory (in bytes) assigned to the query step.
is_rrscan	character(1)	If true (**t**), indicates that range-restricted scan was used on the step. Default is false (**f**).
is_delayed_scan	character(1)	If true (**t**), indicates that delayed scan was used on the step. Default is false (**f**).

Sample queries

Viewing processing information for a query step

The following query shows basic processing information for each step of query 87:

```
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:

```
 query | stm | seg | step |  rows  |  bytes
-------+-----+-----+------+--------+---------
87     |   0 |   0 |    0 |     90 |    1890
87     |   0 |   0 |    2 |     90 |     360
87     |   0 |   1 |    0 |     90 |     360
87     |   0 |   1 |    2 |     90 |    1440
87     |   1 |   2 |    0 | 210494 | 4209880
87     |   1 |   2 |    3 |  89500 |       0
87     |   1 |   2 |    6 |      4 |      96
87     |   2 |   3 |    0 |      4 |      96
87     |   2 |   3 |    1 |      4 |      96
87     |   2 |   4 |    0 |      4 |      96
87     |   2 |   4 |    1 |      1 |      24
87     |   3 |   5 |    0 |      1 |      24
```

```
87     |   3 |   5 |    4 |      0 |       0
(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. 507) view to learn how to obtain the query ID for a query) spilled to disk or if the query ran entirely in-memory:

```
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:

```
query| step|  rows  |  workmem   |  label        | is_diskbased
-----+-----+--------+------------+---------------+--------------
1025 |  0  |16000000|  141557760 |scan tbl=9     | f
1025 |  2  |16000000|  135266304 |hash tbl=142   | t
1025 |  0  |16000000|  128974848 |scan tbl=116536| f
1025 |  2  |16000000|  122683392 |dist           | f
(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. 453), STL_QUERYTEXT (p. 465), and STL_UTILITYTEXT (p. 474) tables. This view also includes a join to the STL_QUERY table.

SVL_STATEMENTTEXT is user visible.

Table columns

Column name	Data type	Description
userid	integer	ID of user who generated entry.
xid	bigint	Transaction ID associated with the statement.
pid	integer	Process ID for the statement.
label	character(30)	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.

Column name	Data type	Description
starttime	timestamp without time zone	Exact time when the statement started executing, with 6 digits of precision for fractional seconds. For example: **2009-06-12 11:29:19.131358**
endtime	timestamp without time zone	Exact time when the statement finished executing, with 6 digits of precision for fractional seconds. For example: **2009-06-12 11:29:19.193640**
sequence	integer	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.
type	varchar(10)	Type of SQL statement: **QUERY**, **DDL**, or **UTILITY**.
text	character(200)	SQL text, in 200-character increments.

Sample query

The following query returns DDL statements that were run on June 16th, 2009:

```
select starttime, type, rtrim(text) from svl_statementtext
where starttime like '2009-06-16%' and type='DDL' order by starttime asc;

starttime          | type |              rtrim
---------------------------|------|-------------------------------
2009-06-16 10:36:50.625097 | DDL  | create table ddltest(c1 int);
2009-06-16 15:02:16.006341 | DDL  | drop view alltickitjoin;
2009-06-16 15:02:23.65285  | DDL  | drop table sales;
2009-06-16 15:02:24.548928 | DDL  | drop table listing;
2009-06-16 15:02:25.536655 | DDL  | drop table event;
...
```

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

Row name	Data type	Description
table_name	text	Name of the table currently being vacuumed, or the table that was last vacuumed if no operation is in progress.
status	text	Description of the current activity being done as part of the vacuum operation (**initialize**, **sort**, or **merge**, for example).

Row name	Data type	Description
time_remaining_estimate	text	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.

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_name    |            status              | time_remaining_estimate
--------------+--------------------------------+-------------------------
salesnew      |  Vacuum: initialize salesnew  |
(1 row)
...
select * from svv_vacuum_progress;

table_name   |         status         | time_remaining_estimate
-------------+------------------------+-------------------------
salesnew     |  Vacuum salesnew sort  | 33m 21s
(1 row)
```

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_name   |  status  | time_remaining_estimate
-------------+----------+-------------------------
  sales      | Complete |
(1 row)
```

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

Row name	Data type	Description
table_name	text	Name of the vacuumed table.
xid	bigint	Transaction ID of the VACUUM operation.
sort_partitions	bigint	Number of sorted partitions created during the sort phase of the vacuum operation.
merge_increments	bigint	Number of merge increments required to complete the merge phase of the vacuum operation.
elapsed_time	bigint	Elapsed run time of the vacuum operation (in microseconds).
row_delta	bigint	Difference in the total number of table rows before and after the vacuum.
sortedrow_delta	bigint	Difference in the number of sorted table rows before and after the vacuum.
block_delta	integer	Difference in block count for the table before and after the vacuum.
max_merge_partitions	integer	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.)

Sample query

The following query returns statistics for vacuum operations on three different tables. The SALES table was vacuumed twice.

```
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_  | xid  |parts|merges| elapsed_ | row_    | sorted_ | block_
name    |      |     |      | time     | delta   | delta   | delta
--------+------+-----+------+----------+---------+---------+--------
users   | 2985 |   1 |    1 | 61919653 |       0 |   49990 |      20
category| 3982 |   1 |    1 | 24136484 |       0 |      11 |       0
sales   | 3992 |   2 |    1 | 71736163 |       0 | 1207192 |      32
sales   | 4000 |   1 |    1 | 15363010 | -851648 | -851648 |    -140
(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. 294) command for more information about the vacuum utility.

SVL_VACUUM_PERCENTAGE is superuser visible.

Table rows

Row name	Data type	Description
xid	bigint	Transaction ID for the vacuum statement.
table_id	integer	Table ID for the vacuumed table.
percentage	bigint	Percentage of data blocks after a vacuum (relative to the number of blocks in the table before the vacuum was run).

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;

xid  | table_id | percentage
-----+----------+------------
1337 |   100238 |         60
(1 row)
```

After this vacuum operation, the table contained 60% of the original blocks.

System catalog tables

Topics

The system catalogs store schema metadata, such as information about tables and columns. System catalog tables have a PG prefix.

The standard Postgres catalog tables are accessible to Amazon Redshift users. For more information about Postgres 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. 527) parameter is set correctly to include the relevant schemas.

Table columns

Column name	Data type	Description
schemaname	name	Schema name.
tablename	name	Table name.
column	name	Column name.
type	text	Datatype of column.
encoding	character(32)	Encoding of column.
distkey	boolean	True if this column is the distribution key for the table.
sortkey	integer	If greater than 0, the column is part of the sort key. 1 = primary sort key, 2 = secondary sort key, and so on.
notnull	boolean	True if the column has a NOT NULL constraint.

Example

This example returns the information for table T2.

```
select * from pg_table_def where tablename = 't2';
schemaname|tablename|column|  type   | encoding | distkey |sortkey| notnull
----------+---------+------+---------+----------+---------+-------+---------
public    | t2      | c1   | bigint  | none     | t       |     0 | f
public    | t2      | c2   | integer | mostly16 | f       |     0 | f
public    | t2      | c3   | integer | none     | f       |     1 | t
public    | t2      | c4   | integer | none     | f       |     2 | f
(4 rows)
```

PG_TABLE_DEF will only return information for tables in schemas that are included in the search path. See search_path (p. 527).

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';

schemaname|tablename|column| type | encoding | distkey | sortkey | notnull
----------+---------+------+------+----------+---------+---------+--------
```

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

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, if you create a table named ALLREDSHIFTUSERS that lists the users of your Amazon Redshift database, and you try to join it to the PG_SHADOW table, the query returns an error:

```
select * from allredshiftusers, pg_shadow where usesysid=usid;
INFO:  Column "pg_shadow.valuntil" has unsupported type "abstime".
INFO:  Column "pg_shadow.useconfig" has unsupported type "text[]".
ERROR:  Specified types or functions (one per INFO message) not supported on
Amazon Redshift tables.
```

You will see similar errors when catalog views are defined with SQL functions that Amazon Redshift does not support. 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
- pg_cast
- pg_depend
- pg_description
- pg_locks
- pg_opclass

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_shadow
- 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.

```
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;

nspname | relname  | num_cols
--------+----------+----------
public  | category |        4
public  | date     |        8
public  | event    |        6
public  | listing  |        8
public  | sales    |       10
public  | users    |       18
public  | venue    |        5
(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;

datname | nspname | relname  |  rows
--------+---------+----------+--------
tickit  | public  | category |     11
tickit  | public  | date     |    365
tickit  | public  | event    |   8798
tickit  | public  | listing  | 192497
tickit  | public  | sales    | 172456
tickit  | public  | users    |  49990
tickit  | public  | venue    |    202
(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;

attrelid |  rtrim   |    attname     |  typname
---------+----------+----------------+-----------
  100133 | users    | likebroadway   | bool
  100133 | users    | likeclassical  | bool
  100133 | users    | likeconcerts   | bool
...
  100137 | venue    | venuestate     | bpchar
  100137 | venue    | venueid        | int2
  100137 | venue    | venueseats     | int4
  100137 | venue    | venuecity      | varchar
...
```

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.

```
select col, count(*)
from stv_blocklist s, pg_class p
where s.tbl=p.oid and relname='sales'
group by col
order by col;

col | count
----+-------
  0 |     4
  1 |     4
  2 |     4
  3 |     4
  4 |     4
  5 |     4
  6 |     4
  7 |     4
  8 |     4
  9 |     8
 10 |     4
 12 |     4
 13 |     8
(13 rows)
```

Configuration Reference

Topics

Modifying the server configuration

You can make changes to the server configuration in the following ways:

- By using a SET (p. 276) command to override a setting for the duration of the current session only.

  For example:

```
set extra_float_digits to 2;
```

- By modifying the parameter group settings for the cluster. 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 view all settings.

```
show all;
```

```
name              |setting
--------------------+---------------
datestyle         |ISO, MDY
extra_float_digits   |2
query_group       |default
```

```
search_path       |$user, public
statement_timeout |0
wlm_query_slot_count |1
(6 rows)
```

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.

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;

query | pid  |        substring                    |  elapsed  | label
------+------+-------------------------------------+-----------+--------
789   | 6084 | select * from category limit 1;     | 65468     | Monday
790   | 6084 | select query, trim(label) from ... | 1260327   | Monday
791   | 6084 | select * from svl_qlog where ..     | 2293547   | Monday
792   | 6084 | select count(*) from bigsales;      | 108235617 | Monday
...
```

search_path

Values (default in bold)

'$user, public' (the second part will be ignored if there is no schema named public), a comma-separated list of schema names (If $user is present, then the schema having the same name as the SESSION_USER is substituted when available, otherwise 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

In this example, the schema BOBR has been added to the search_path parameter:

```
show search_path;
search_path
-------------------
$user,public,bobr
(1 row)

create table bobr.t1 (c1 int);

...
```

When the table PUBLIC.T1 is created in the same database, and the user does not specify the schema name in a query, PUBLIC.T1 takes precedence over BOBR.T1:

```
create table public.t1(c1 int);

insert into bobr.t1 values(1);

select * from t1;
c1
----
(0 rows)

select * from bobr.t1;
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. 104).

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 15 (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 cluster (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. 102).

> **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

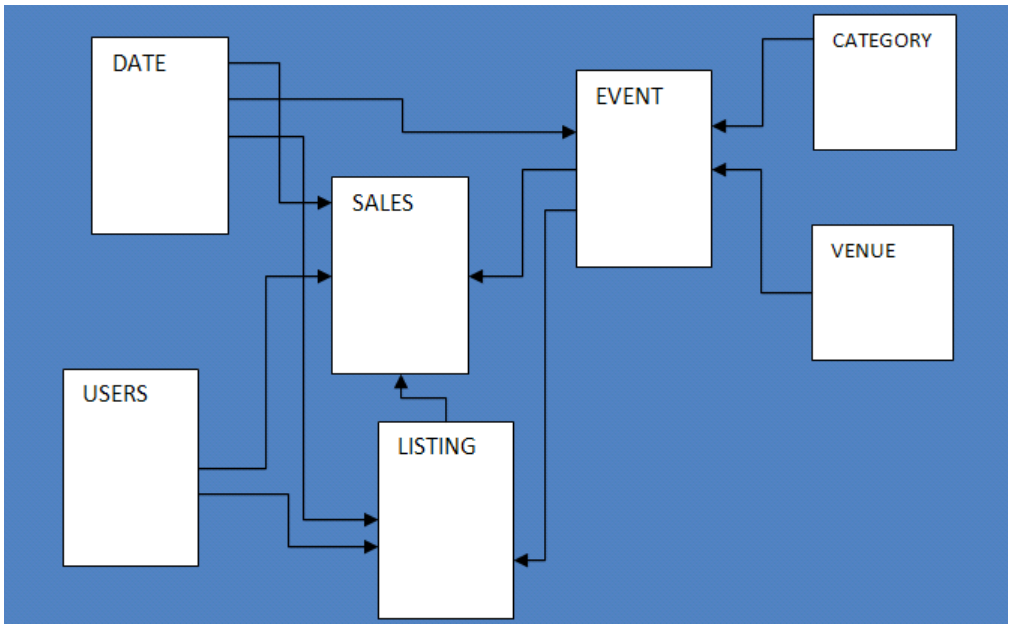
Use the SET command to set the value of wlm_query_slot_count for the duration of the current session.

```
set wlm_query_slot_count to 3;
```

Sample Database

Topics

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;

sellerid | username |       name        |   city    | sum
---------+----------+-------------------+-----------+-----
49977 | JJK84WTE | Julie Hanson      | San Diego |  22
19750 | AAS23BDR | Charity Zimmerman | San Diego |  21
29069 | SVL81MEQ | Axel Grant        | San Diego |  17
43632 | VAG08HKW | Griffin Dodson    | San Diego |  16
36712 | RXT40MKU | Hiram Turner      | San Diego |  14
(5 rows)
```

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

Column name	Data type	Description
CATID	SMALLINT	Primary key, a unique ID value for each row. Each row represents a specific type of event for which tickets are bought and sold.
CATGROUP	VARCHAR(10)	Descriptive name for a group of events, such as **`Shows`** and **`Sports`**.
CATNAME	VARCHAR(10)	Short descriptive name for a type of event within a group, such as **`Opera`** and **`Musicals`**.
CATDESC	VARCHAR(30)	Longer descriptive name for the type of event, such as **`Musical theatre`**.

DATE table

Column name	Data type	Description
DATEID	SMALLINT	Primary key, a unique ID value for each row. Each row represents a day in the calendar year.
CALDATE	DATE	Calendar date, such as **2008-06-24**.
DAY	CHAR(3)	Day of week (short form), such as **SA**.
WEEK	SMALLINT	Week number, such as **26**.
MONTH	CHAR(5)	Month name (short form), such as **JUN**.
QTR	CHAR(5)	Quarter number (**1** through **4**).
YEAR	SMALLINT	Four-digit year (**2008**).
HOLIDAY	BOOLEAN	Flag that denotes whether the day is a public holiday (U.S.).

EVENT table

Column name	Data type	Description
EVENTID	INTEGER	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.
VENUEID	SMALLINT	Foreign-key reference to the VENUE table.
CATID	SMALLINT	Foreign-key reference to the CATEGORY table.
DATEID	SMALLINT	Foreign-key reference to the DATE table.
EVENTNAME	VARCHAR(200)	Name of the event, such as **Hamlet** or **La Traviata**.
STARTTIME	TIMESTAMP	Full date and start time for the event, such as **2008-10-10 19:30:00**.

VENUE table

Column name	Data type	Description
VENUEID	SMALLINT	Primary key, a unique ID value for each row. Each row represents a specific venue where events take place.
VENUENAME	VARCHAR(100)	Exact name of the venue, such as **Cleveland Browns Stadium**.
VENUECITY	VARCHAR(30)	City name, such as **Cleveland**.
VENUESTATE	CHAR(2)	Two-letter state or province abbreviation (United States and Canada), such as **OH**.

Column name	Data type	Description
VENUESEATS	INTEGER	Maximum number of seats available at the venue, if known, such as **73200**. For demonstration purposes, this column contains some null values and zeroes.

USERS table

Column name	Data type	Description
USERID	INTEGER	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.
USERNAME	CHAR(8)	An 8-character alphanumeric username, such as **PGL08LJI**.
FIRSTNAME	VARCHAR(30)	The user's first name, such as **Victor**.
LASTNAME	VARCHAR(30)	The user's last name, such as **Hernandez**.
CITY	VARCHAR(30)	The user's home city, such as **Naperville**.
STATE	CHAR(2)	The user's home state, such as **GA**.
EMAIL	VARCHAR(100)	The user's email address; this column contains random Latin values, such as **turpis@accumsanlaoreet.org**.
PHONE	CHAR(14)	The user's 14-character phone number, such as **(818) 765-4255**.
LIKESPORTS, ...	BOOLEAN	A series of 10 different columns that identify the user's likes and dislikes with **true** and **false** values.

LISTING table

Column name	Data type	Description
LISTID	INTEGER	Primary key, a unique ID value for each row. Each row represents a listing of a batch of tickets for a specific event.
SELLERID	INTEGER	Foreign-key reference to the USERS table, identifying the user who is selling the tickets.
EVENTID	INTEGER	Foreign-key reference to the EVENT table.
DATEID	SMALLINT	Foreign-key reference to the DATE table.
NUMTICKETS	SMALLINT	The number of tickets available for sale, such as **2** or **20**.
PRICEPERTICKET	DECIMAL(8,2)	The fixed price of an individual ticket, such as **27.00** or **206.00**.
TOTALPRICE	DECIMAL(8,2)	The total price for this listing (NUMTICKETS*PRICEPERTICKET).

Column name	Data type	Description
LISTTIME	TIMESTAMP	The full date and time when the listing was posted, such as **2008-03-18 07:19:35**.

SALES table

Column name	Data type	Description
SALESID	INTEGER	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.
LISTID	INTEGER	Foreign-key reference to the LISTING table.
SELLERID	INTEGER	Foreign-key reference to the USERS table (the user who sold the tickets).
BUYERID	INTEGER	Foreign-key reference to the USERS table (the user who bought the tickets).
EVENTID	INTEGER	Foreign-key reference to the EVENT table.
DATEID	SMALLINT	Foreign-key reference to the DATE table.
QTYSOLD	SMALLINT	The number of tickets that were sold, from **1** to **8**. (A maximum of 8 tickets can be sold in a single transaction.)
PRICEPAID	DECIMAL(8,2)	The total price paid for the tickets, such as **75.00** or **488.00**. The individual price of a ticket is PRICEPAID/QTYSOLD.
COMMISSION	DECIMAL(8,2)	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.
SALETIME	TIMESTAMP	The full date and time when the sale was completed, such as **2008-05-24 06:21:47**.

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: Sept 10, 2013

Change	Description	Date Changed
JSON, Regular Expression, and Cursors.	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. 426), Pattern-matching conditions (p. 149), and DECLARE (p. 216).	Sept 10, 2013
ACCEPTINVCHAR option for COPY	You can successfully load data that contains invalid UTF-8 characters by specifying the ACCEPTINVCHAR option with the COPY (p. 179) command.	Aug 29, 2013
CSV option for COPY	The COPY (p. 179) command now supports loading from CSV formatted input files.	Aug 9, 2013
CRC32	The CRC32 function (p. 403) performs cyclic redundancy checks.	Aug 9, 2013
WLM wildcards	Workload management (WLM) supports wildcards for adding user groups and query groups to queues. For more information, see Wildcards (p. 103).	Aug 1, 2013
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. For more information, see WLM timeout (p. 104).	Aug 1, 2013
New COPY options 'auto' and 'epochsecs'	The COPY (p. 179) command performs automatic recognition of date and time formats. New time formats, 'epochsecs' and 'epochmillisecs' enable COPY to load data in epoch format.	July 25, 2013

Change	Description	Date Changed
CONVERT_TIMEZONE function	The CONVERT_TIMEZONE function (p. 349) converts a timestamp from one timezone to another.	July 25, 2013
FUNC_SHA1 function	The FUNC_SHA1 function (p. 403) converts a string using the SHA1 algorithm.	July 15, 2013
max_execution_time	To limit the amount of time queries are permitted to use, you can set the max_execution_time parameter as part of the WLM configuration. For more information, see Modifying the WLM configuration (p. 106).	July 22, 2013
Four-byte UTF-8 characters	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 Storage and ranges (p. 128).	July 18, 2013
SVL_QERROR	The SVL_QERROR system view has been deprecated.	July 12, 2013
Revised Document History	The Document History page now shows the date the documentation was updated.	July 12, 2013
STL_UNLOAD_LOG	STL_UNLOAD_LOG (p. 473) records the details for an unload operation.	July 5, 2013
JDBC fetch size parameter	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. 102).	June 27, 2013
UNLOAD encrypted files	UNLOAD (p. 282) now supports unloading table data to encrypted files on Amazon S3.	May 22, 2013
Temporary credentials	COPY (p. 179) and UNLOAD (p. 282) now support the use of temporary credentials.	April 11, 2013
Added clarifications	Clarified and expanded discussions of Designing Tables and Loading Data.	February 14, 2013
Added Best Practices	Added Best practices for designing tables (p. 32) and Best practices for loading data (p. 54).	February 14, 2013
Clarified password constraints	Clarified password constraints for CREATE USER and ALTER USER, various minor revisions.	February 14, 2013
New Guide	This is the first release of the *Amazon Redshift Developer Guide*.	February 14, 2013