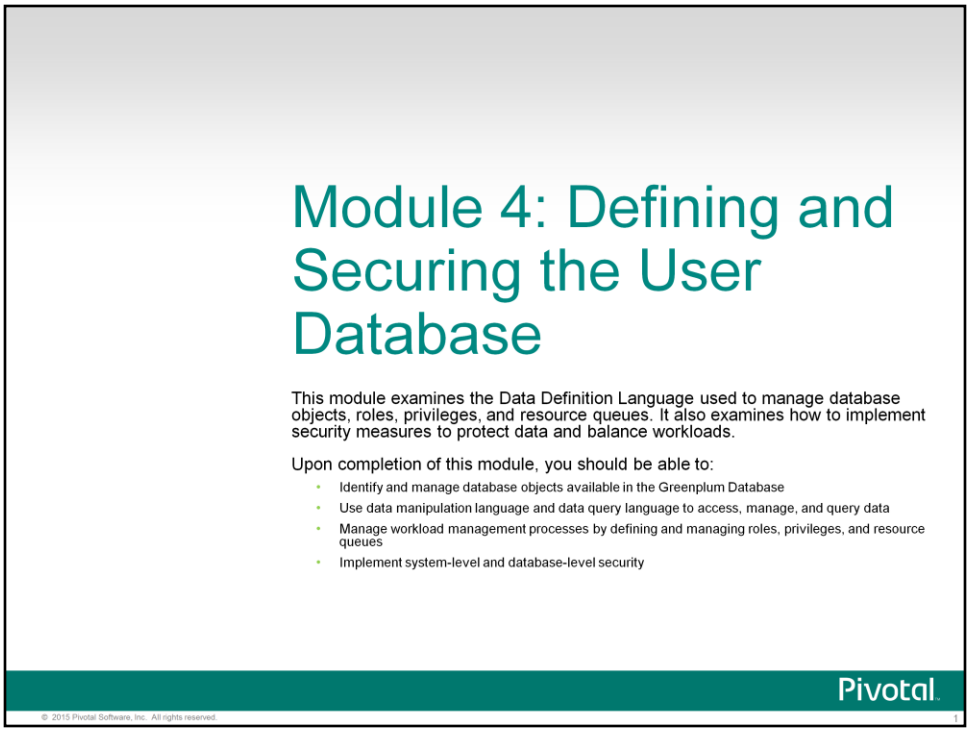
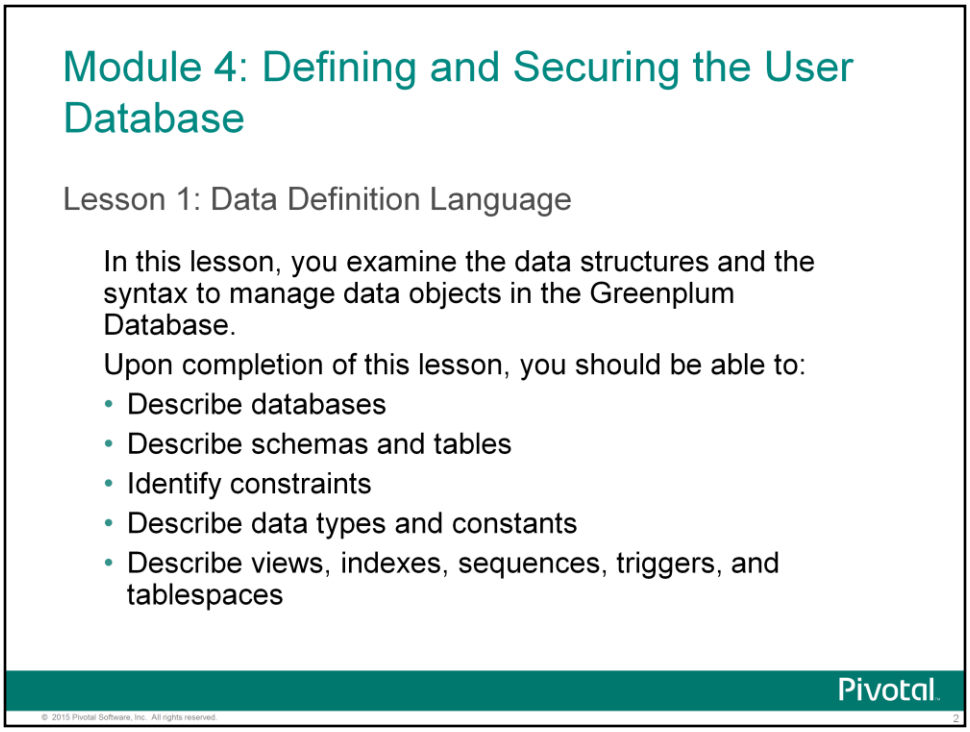


Data definition language (DDL) defines the data structures you will manage within the Greenplum Database. You use SQL syntax to create, alter, and remove database objects, including tables, schemas, and views.

As an administrator or developer, you must also learn the SQL syntax to manage the workload resources, which includes managing roles, privileges, and the resource queues. Once you know how to manage these items using SQL syntax, you can implement security measures to protect data and resources from unauthorized users.

In this module, you will:

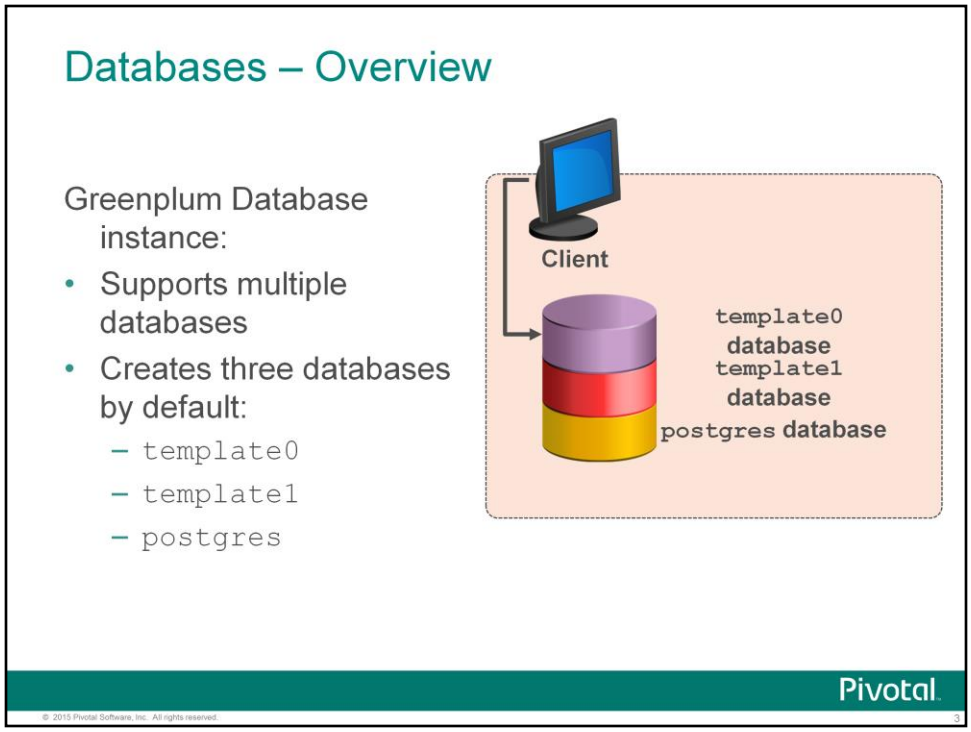
- Identify and manage database objects for the Greenplum Database.
- Use data manipulation language and data query language to access, manage, and query data
- Manage workload management processes by defining and managing roles, privileges, and resource queues.
- Implement system-level and database-level security.



Data Definition Language is the SQL syntax that you use to manage database objects. In this lesson, you examine the data structures and the syntax for managing database objects in Greenplum.

Once you have completed this lesson, you should be able to:

- Describe what a database is and how to manage it with SQL commands.
- Describe schemas and tables and manage these with SQL commands.
- Identify constraints.
- Describe data types and constants in the Greenplum Database.
- Describe other database objects, including views, indexes, sequences, triggers, and tablespaces.

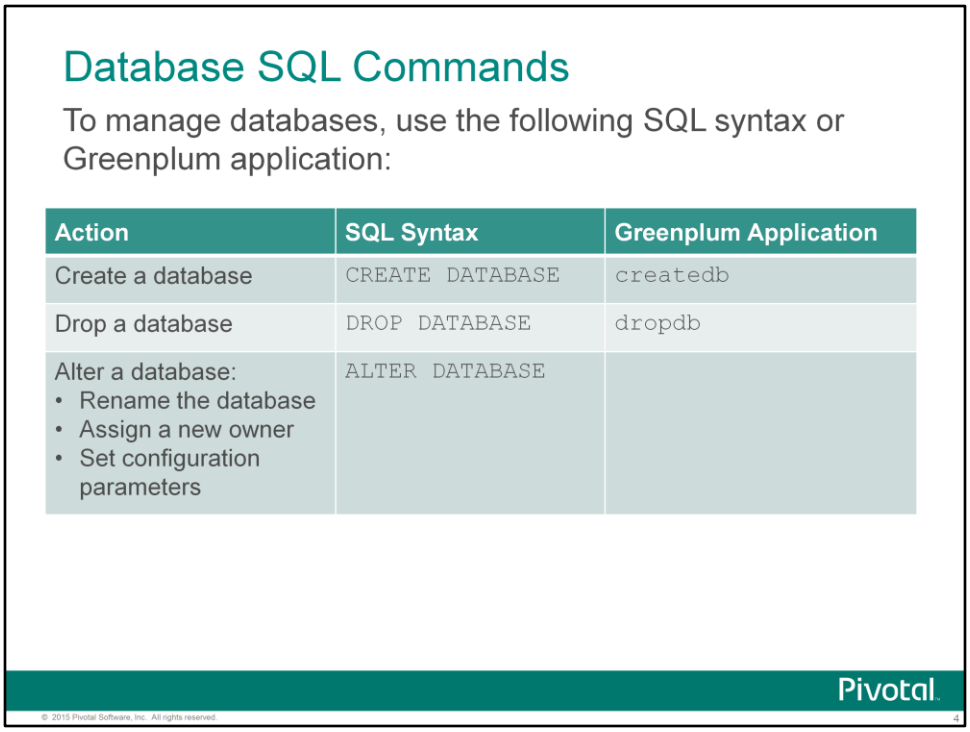


As in PostgreSQL, an Greenplum Database instance can have one or more user created databases. This differs from systems such as Oracle where the DBMS and database instance are one in the same. These databases reside on a single system but do not share data.

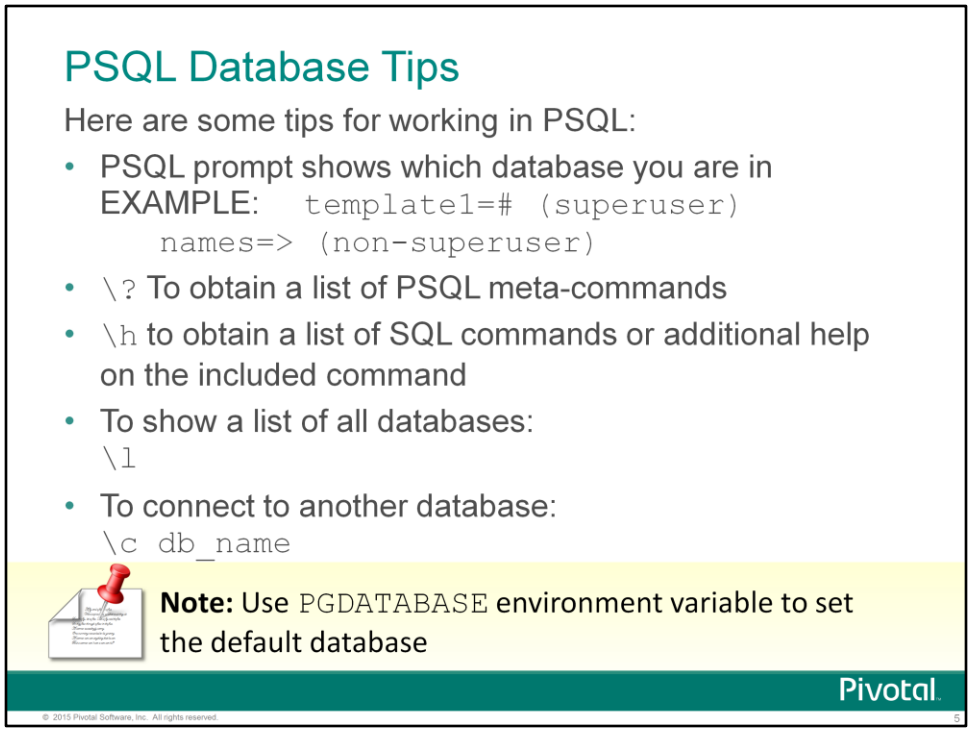
Clients can connect to and have access to one database at a time.

Three databases are created by default in a newly initialized array:

- **`template1`** – This default database is the template used to create other databases. Do not create any objects in this database unless you want those objects to also be in every other database you create moving forward.
  Any database can be used as a template. A template provides you with a method of cloning databases and carrying specific base objects forward. Use the SQL syntax, `CREATE DATABASE newname USING TEMPLATE clonename` to create a database based on a template.
- **`template0`** – This template is a blank template used to create `template1`. This should not be dropped or modified.
- **`postgres`** – This is a default PostgreSQL database introduced in PostgreSQL 8.1 and is used internally by the system. It must not be dropped or modified.



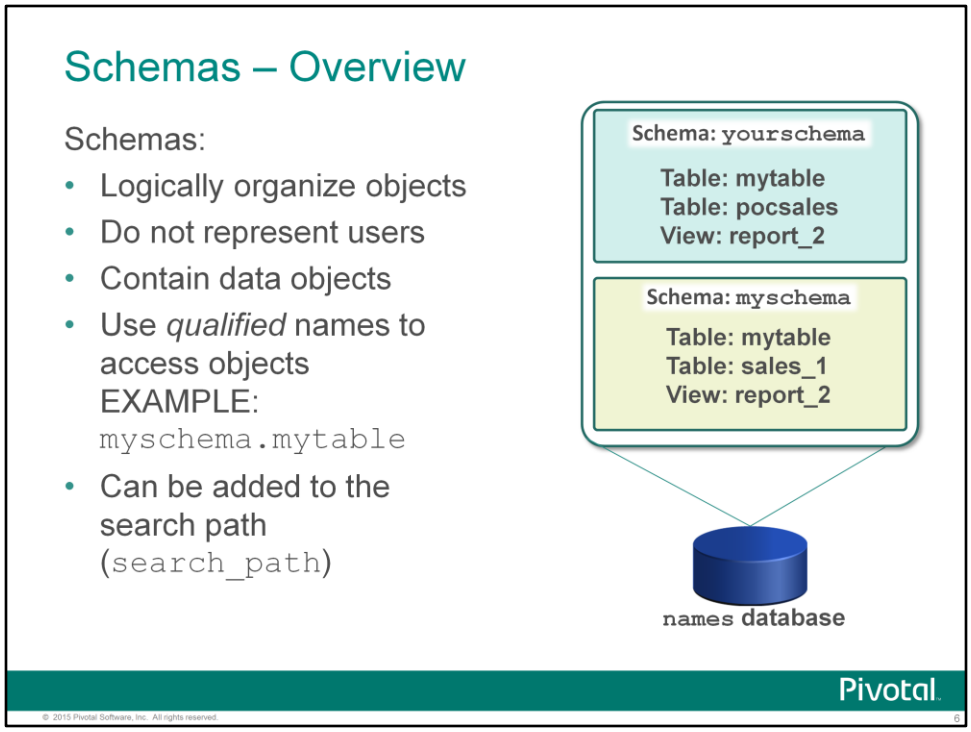
PostgreSQL and Greenplum Database share the same SQL commands for creating the database. You can use either SQL syntax in a PostgreSQL client to create, drop, or alter the database or use the Greenplum application clients available to create and drop the database.



Here are some helpful tips for working with databases in the PSQL client program:

- The psql prompt will show the name of the database you are connected to and if you are in as a superuser role or not. For this class, we are logged in as a superuser.
- You can obtain additional assistance with the list of available meta-commands with `\?`.
- `\h` provides a list of SQL commands and lets you obtain additional assistance on a specific command if the command is provided as part of the command line.
- `\l` shows a list of all databases, including the templates and defaults databases created.
- Use `\c` to connect to another database or to reconnect to the same database. For example, after setting a parameter on a database you will need to reconnect to the database to see the changes.

Note: If you have set the `PGDATABASE` variable in your shell, you do not have to type the database name when connecting with psql.



Schemas:

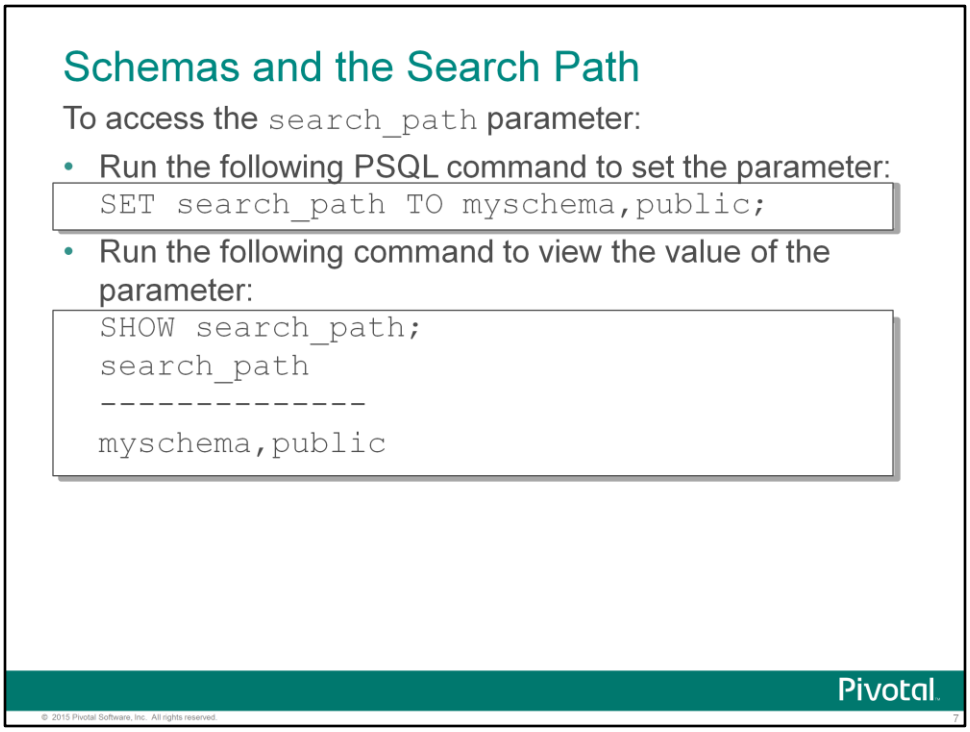
- Are a way to logically organize a database for users or for functional areas. For example, data from third-party applications can be stored in separate schemas to prevent collision with the names of other objects.
- Do not represent users, as is seen in Oracle.
- Contain named objects such as tables, views, indexes, data types, functions, and operators.

  Allow you to have more than one object, such as tables, with the same name in the database without conflict, as long as they are in different schemas.

For the database to know in which schema it should look for an object, you use the *qualified* name. If you do not want to type the qualified name all the time, you can set the `search_path` parameter. This tells the database in which order it should search the available schemas for objects. The schema listed first in the search path becomes the default schema, which is where new, unqualified objects will be created by default.

As a best practice, you should always specify a schema when allowed. Changes in data with the addition of more objects with the same name in different schemas may cause issues when the optimizer and the query attempts to resolve table names.

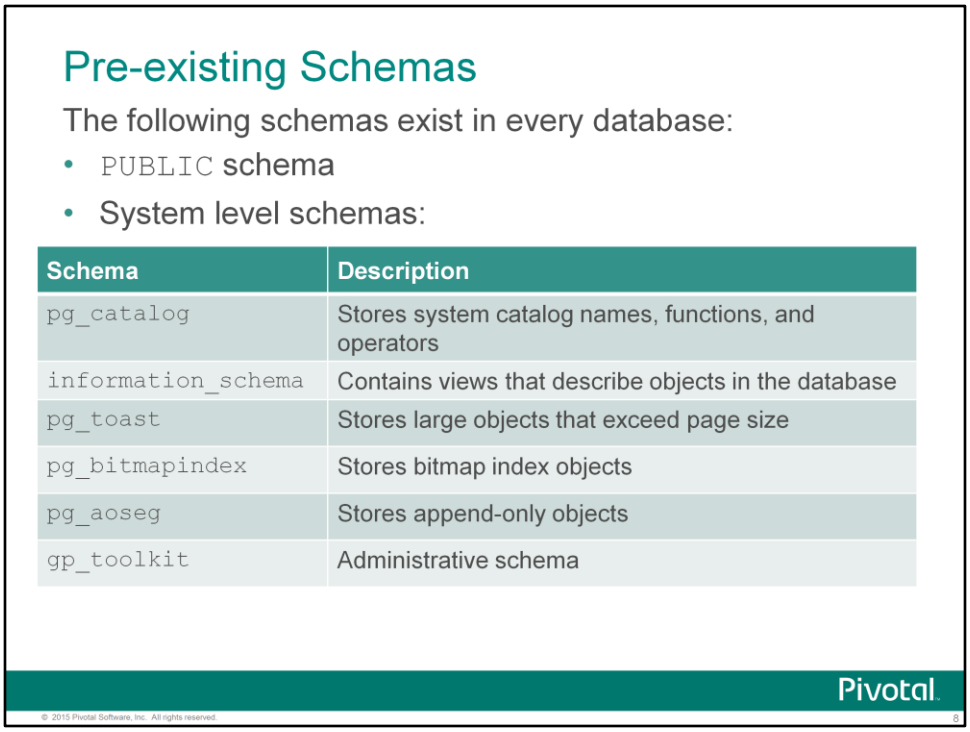
Unlike databases, schemas are not rigidly separated - a user can access objects in any of the schemas in the database he is connected to, if he has privileges to do so.



PostgreSQL will search for tables that are named without qualification by using the `search_path` parameter. Tables without qualifications are tables that do not have the schema name specified when calling the table name.

The following is used to access the `search_path` parameter from PSQL:

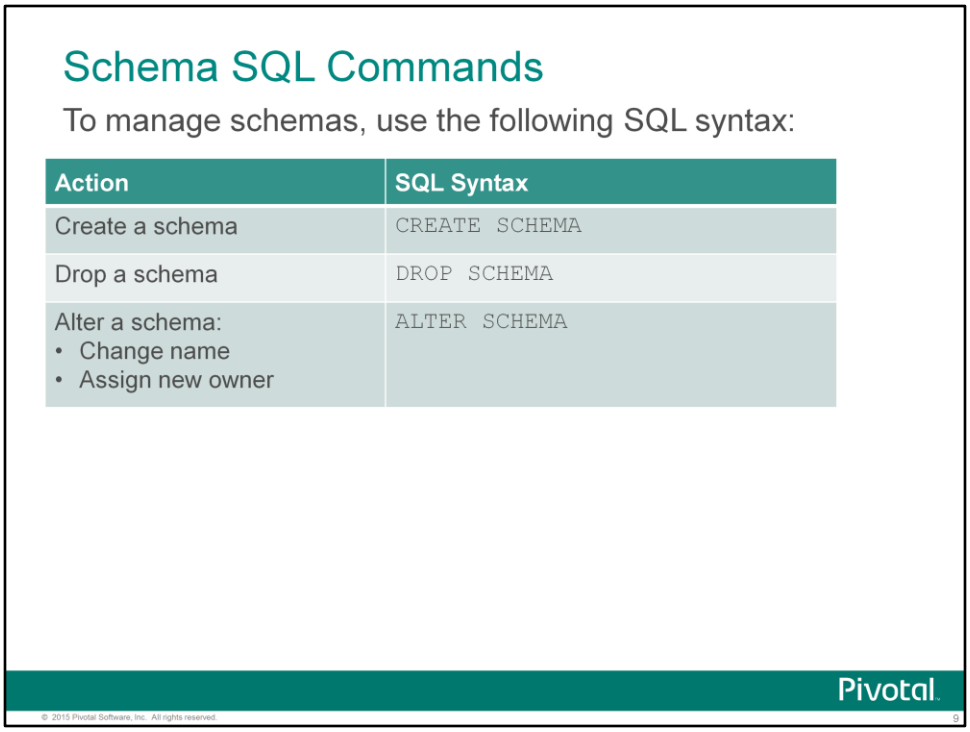
- To set the `search_path` parameter, issue the following command:
  `SET search_path TO myschema,public;`
- To view the contents of the `search_path` parameter, issue the following command:
  `SHOW search_path;`



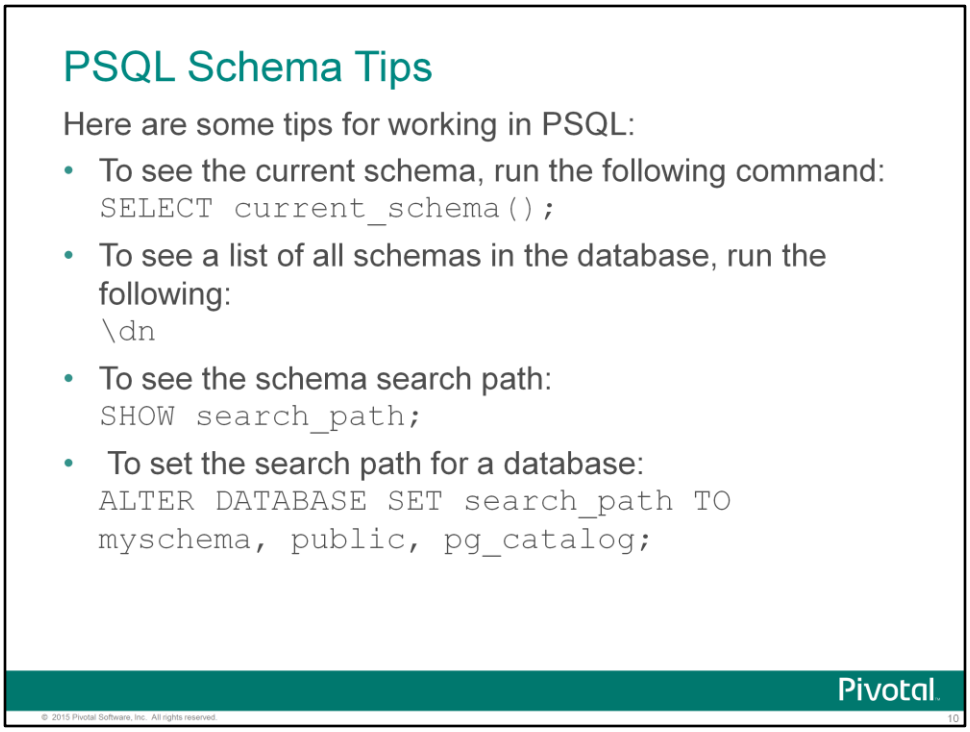
Every newly created database has a default schema named `PUBLIC`. By default, all roles have `CREATE` and `USAGE` privileges on the public schema. Any other schemas you create you will have to grant the appropriate privileges to roles other than the owner.

The following system-level schemas also exist in every database:

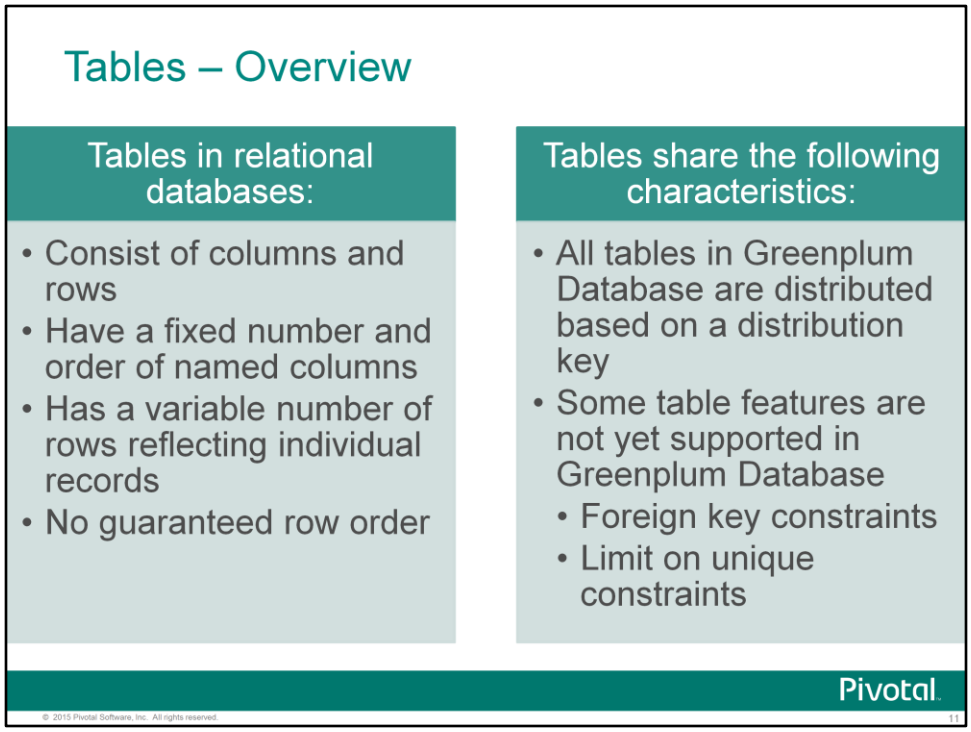
- **`pg_catalog`** is the schema that has the system catalog tables, built-in data types, functions, and operators. It is always part of the search path, even if it is not named.
- **`information`**`_schema` consists of a set of views that contain information about the objects in the database. This is used by psql meta-commands to get information from the system catalog tables.
- **`pg_toast`** are where large objects are stored, such as records that exceed the page size.
- **`pg_bitmapindex`** where bitmap index objects are stored (list of values, etc).
- **`pg_aoseg`** is the schema that stores append-only objects.
- **`gp_toolkit`** is the administrative schema that you use to view and query system log files and other system metrics. It contains a number of external tables, views, and functions that you can access with SQL commands and is accessible by all database users.



PostgreSQL and Greenplum Database share the same SQL commands for creating and managing schemas.



Here are some helpful tips for working with databases in the PSQL client program.



A table in a relational database:

- Is much like a table on paper where it consists of columns and rows.
- Has a fixed number and order of columns and each column is named.
- Has a variable number of rows reflecting how much data is stored at a given moment.

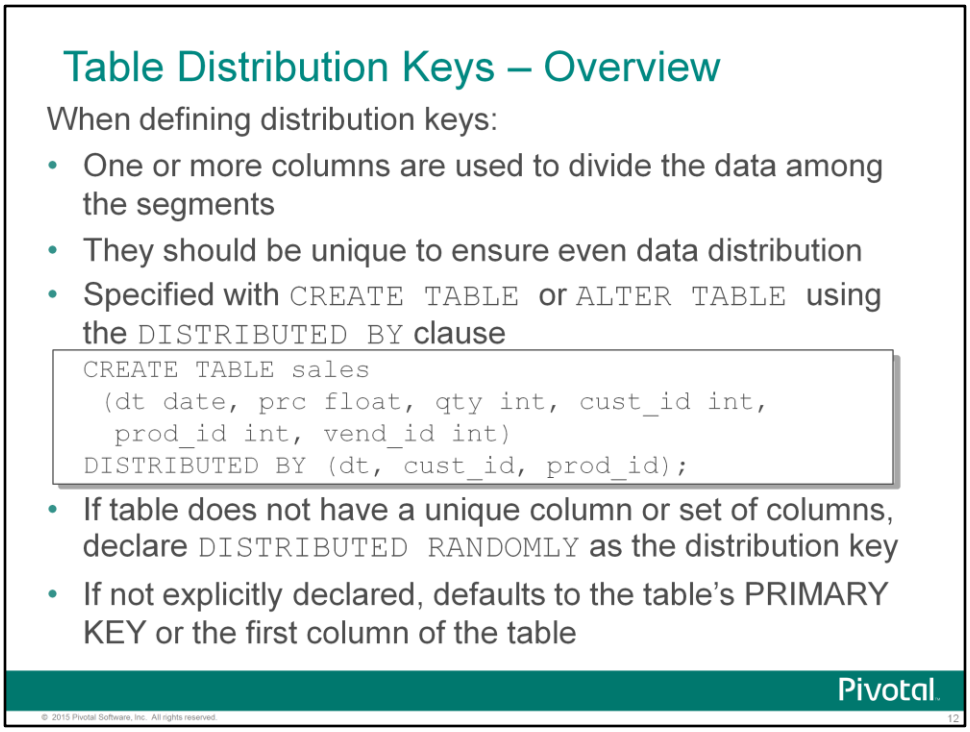
PostgreSQL makes no guarantees about the order of the rows in a table.

One important difference about tables in Greenplum Database is that all tables are distributed, meaning they are divided into parts amongst the primary segment instances. The distribution is done using a sophisticated proprietary hashing algorithm on the distribution key of the table.

The distribution key can be declared when the table is created or altered. It should be chosen with care.

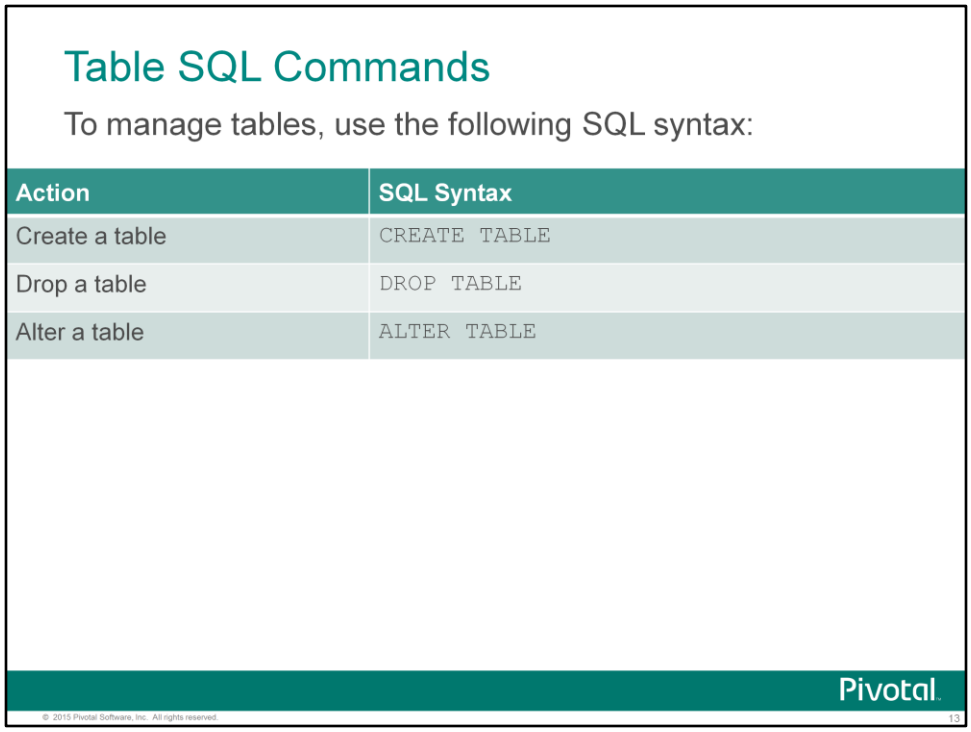
Because tables are distributed, some features are not yet supported Greenplum Database:

- Foreign key integrity cannot yet be enforced in a distributed table, so this is not supported.
- A table cannot have multiple columns with unique constraints, and any single column that has a unique constraint must also be declared as the distribution key.

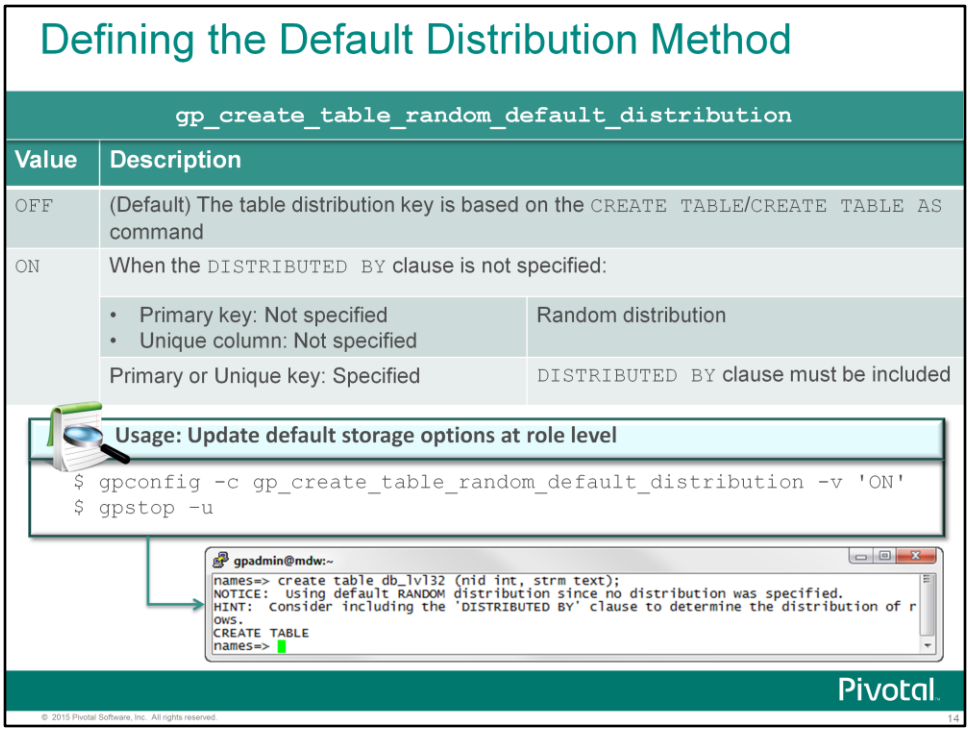


Let us examine distribution keys in greater detail, as this is a very important concept in Greenplum Database. Table distribution is critical to getting the best performance possible.

- In a table, one or more columns are used as the distribution key, meaning those columns are used to divide the data amongst all of the segments.
- To ensure an even distribution of data, you want to choose a distribution key that is unique for each record. If that is not possible, declare DISTRIBUTED RANDOMLY as the distribution key.
- The distribution key is chosen when the table is created or altered using the `DISTRIBUTED BY` clause. This is a Greenplum extension to the SQL standard. Note that you cannot alter the columns of a table that are used as the distribution key after the table is created.
- If a `DISTRIBUTED BY` clause is not supplied, then either the PRIMARY KEY, if the table has one, or the first eligible column of the table will be used as the distribution key. This may or may not be the desirable distribution key. Columns of geometric data types are not eligible as distribution key columns. If a table does not have a column of an eligible data type, the rows are distributed based on a random distribution.



PostgreSQL and Greenplum Database share the same SQL commands for creating and managing tables.



The `gp_create_table_random_default_distribution` parameter defines how the `CREATE TABLE` and `CREATE TABLE AS` statements behave with regards to table distribution.

When the parameter is set to `OFF`, which is the default value, the table distribution key follows the methods outlined earlier. If the `DISTRIBUTED BY` clause is not specified:

- The primary key is used, if the table has one, or the first eligible column is used.
- The table inherits the parent table or source table if the `LIKE`/`INERHITS` clause is used

If the parameter is set to `ON`:

- And neither the primary key or a unique column is defined, random distribution is chosen.
- Either the primary key or the unique column is defined, the `DISTRIBUTED BY` clause must be included in the `CREATE TABLE` command or the command will fail.

The parameter is settable with the `gpconfig` command and requires a reload if set.

Table SQL Commands – Creating a Table

The following is an example of creating a table:

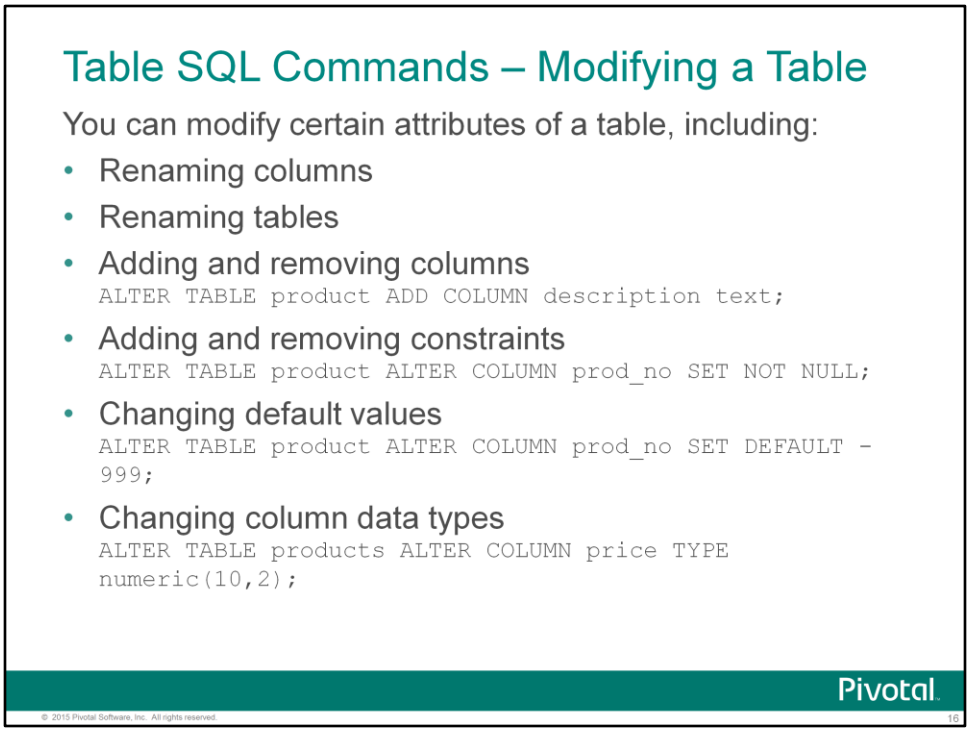
Example: Creating a Table

```
CREATE TABLE dimensions.store
(
  storeId      SMALLINT     NOT NULL,
  storeName    VARCHAR(40)  NOT NULL,
  address      VARCHAR(50)  NOT NULL,
  city         VARCHAR(40)  NOT NULL,
  state        CHAR(2)      NOT NULL,
  zipcode      CHAR(8)      NOT NULL
);
```

Pivotal

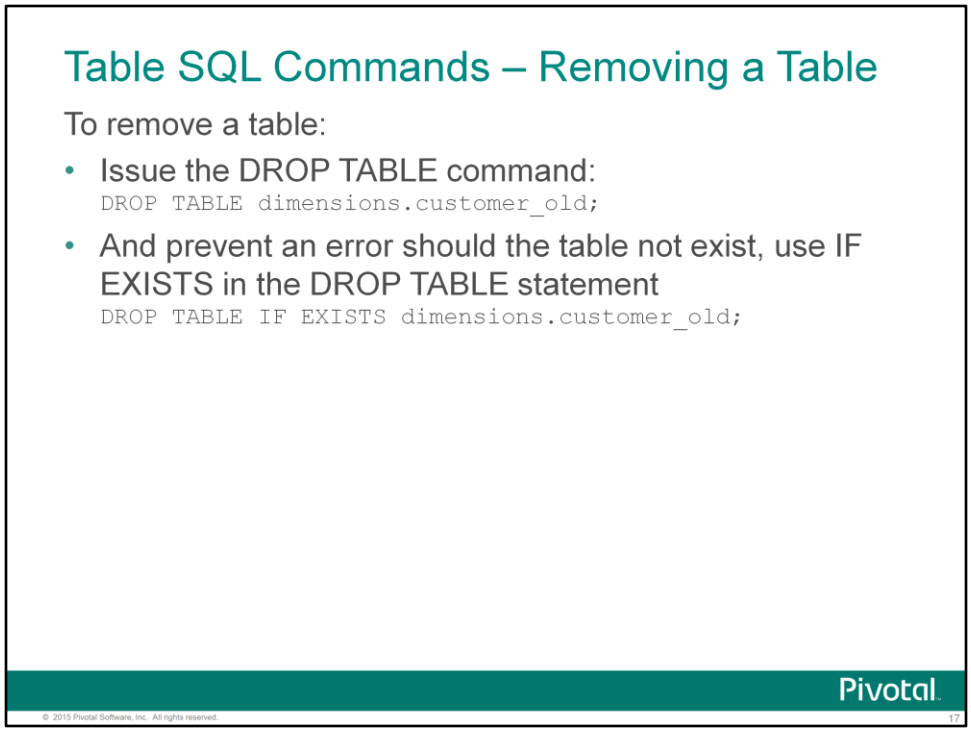
© 2015 Pivotal Software, Inc. All rights reserved.

In this example, the table, `store`, is created in the schema, `dimensions`. The table consists of 6 columns of varying data types.

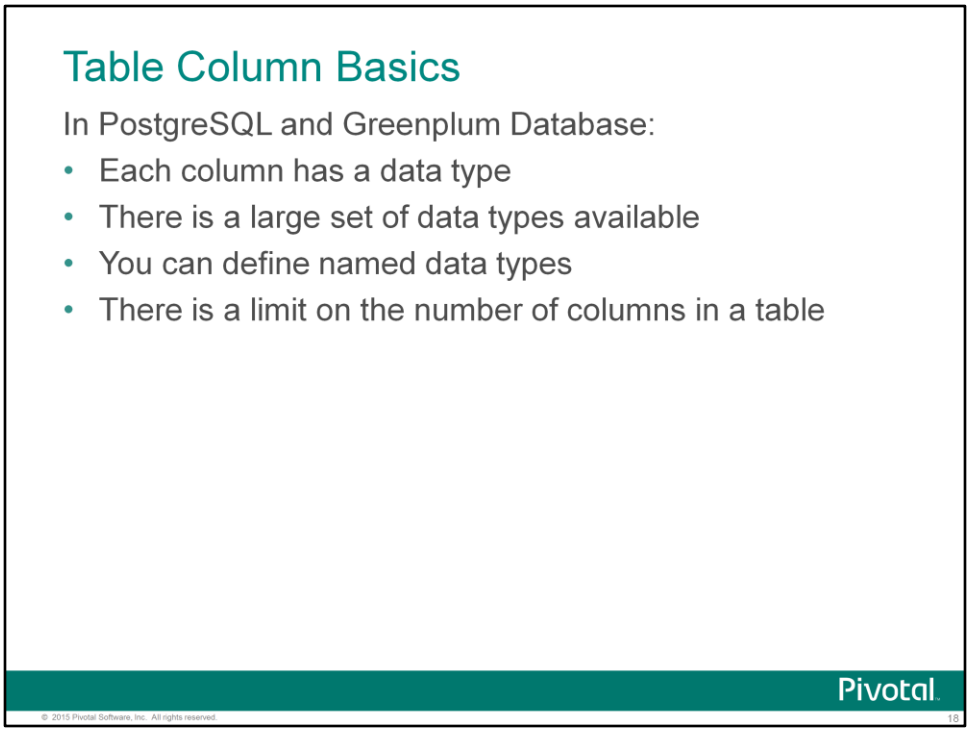


You can alter several properties of a table, including:

- Column names
- Table name
- The number of columns by adding or removing columns
- Constraints on the table by adding or removing constraints
- Default values for columns in the table
- Column data types

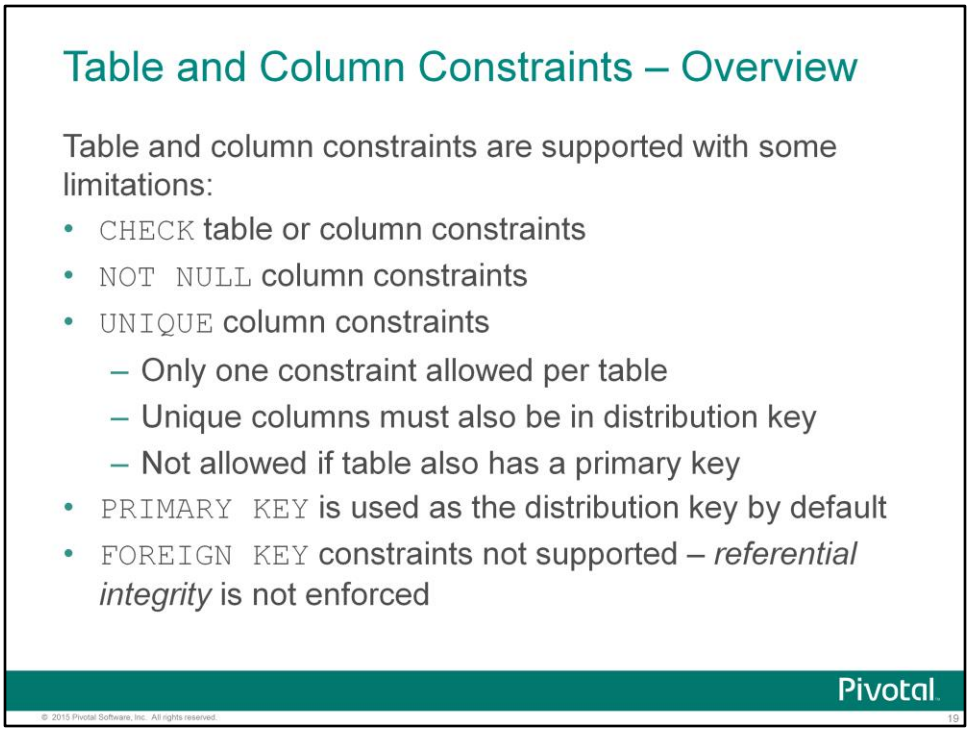


When removing the table, you should qualify the table name and specify the name of the table to be dropped. If the table does not exist, you can avoid error messages by using `IF EXISTS` in the `DROP TABLE` statement.



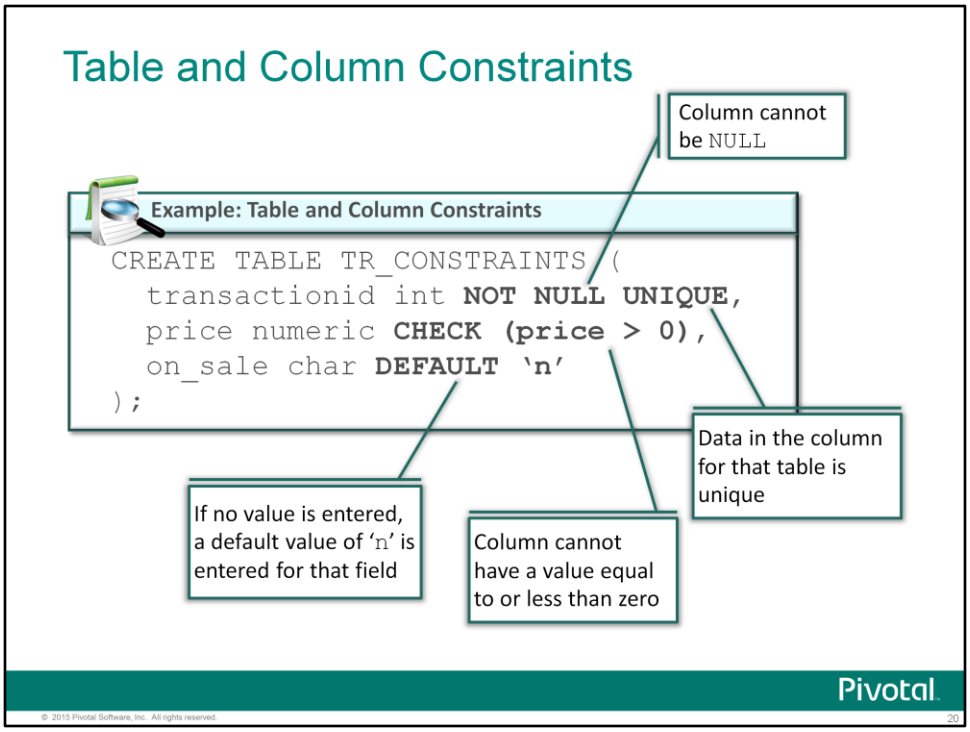
With PostgreSQL and the Greenplum Database, there are some criteria to column definition you should be aware of:

- Each column has a data type.
- PostgreSQL includes a sizable set of built-in data types that fit many applications.
- Users can define their own data types.
- There is a limit on the number of columns a table can contain. Depending on the column types, it is between 250 and 1600. However, defining a table with anywhere near this many columns is highly unusual and often a questionable design.
  The limit is due to the optimizer. An amount greater than 1600 columns will impact performance.



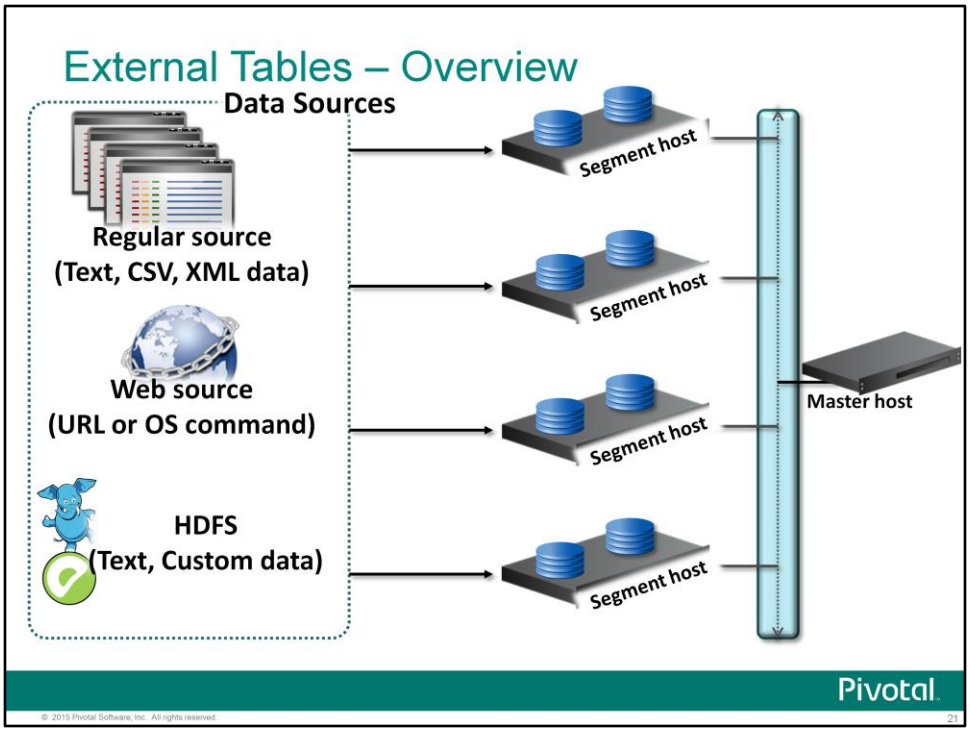
Constraints are a way to limit and control the data that a table contains, limiting the data to a set of valid values. PostgreSQL supports these table and column constraints as does Greenplum Database, with some exceptions:

- `CHECK` constraints limit data based on some condition you define.
- `NOT NULL` constraints verify that a column does not have a null value when a row is created. A way of forcing users to define a value for a column when a new record is created.
- A primary key indicates that a column or group of columns can be used as a unique identifier for rows in a table. Technically, a `PRIMARY KEY` constraint is simply a combination of a UNIQUE constraint and a `NOT NULL` constraint. If a table has a primary key, it will be chosen as the Greenplum distribution key by default. The only way Greenplum Database can enforce uniqueness is if a column is also in the distribution key. Therefore, a table cannot have more than one `UNIQUE` constraint and any `UNIQUE` column or set of columns must be in the distribution key.
- Also PostgreSQL and Greenplum Database always implicitly creates a `pkey` index for a table's primary key.
- `FOREIGN KEYS`, while they can be defined, are not currently supported in Greenplum Database, as there is currently no way to enforce referential integrity between the segment instances.



You can define the following types of constraints:

- A column can be assigned a `DEFAULT` value.
- `CHECK` constraint is the most generic constraint type. It allows you to specify that the value in a certain column must satisfy a Boolean expression.
- `NOT NULL` constraint specifies that a column must not assume the null value.
- `UNIQUE` constraints ensure that the data contained in a column or a group of columns is unique with respect to all the rows in the table.



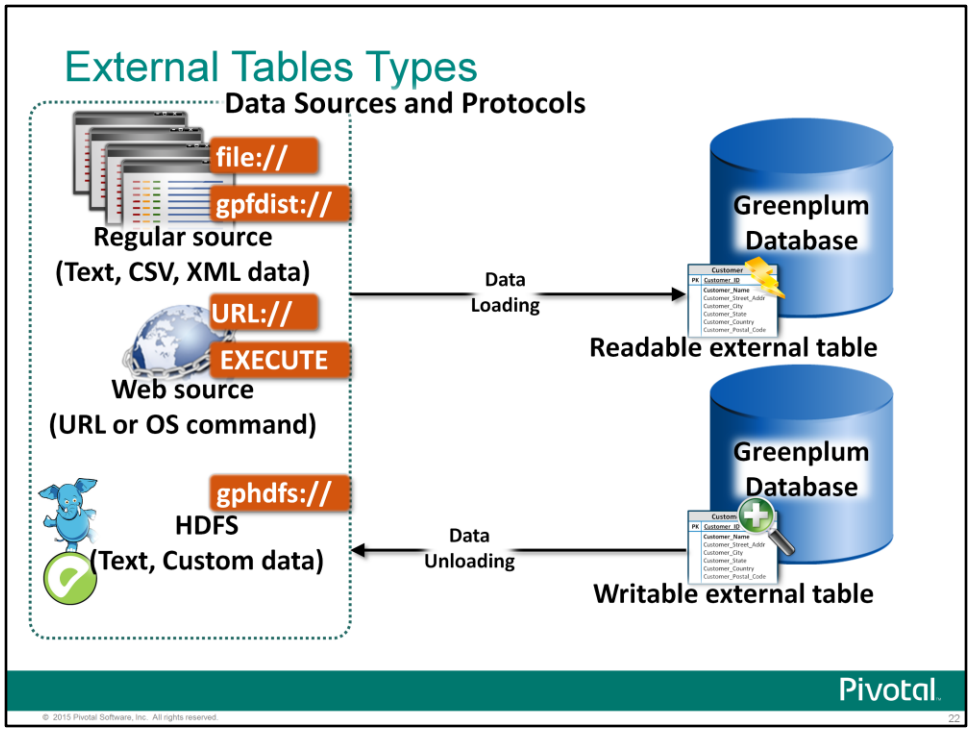
An external table is a definition of a table that specifies how to access a data source outside of the database as well as how that data is formatted. The format varies depending on the type of external table defined.

Large amounts of data can be loaded or unloaded using external tables. Greenplum supports fast, parallel data loading and unloading as well as non-parallel import and export of smaller amounts of data through one of several named protocols.

External tables lets you access data from a data source as though the data source was a regular table.

There are three main types of data sources that Greenplum works with:

- Regular or file-based external sources with support for text data, comma-separated values (CSV) formatted data, or XML data.
- Web-based external sources with support for text and CSV formatted data, as well as operating system commands and scripts.
- Hadoop file-system data sources or tables, with support for text-based data and custom or user-defined formatted data.



Greenplum supports two types of external tables:

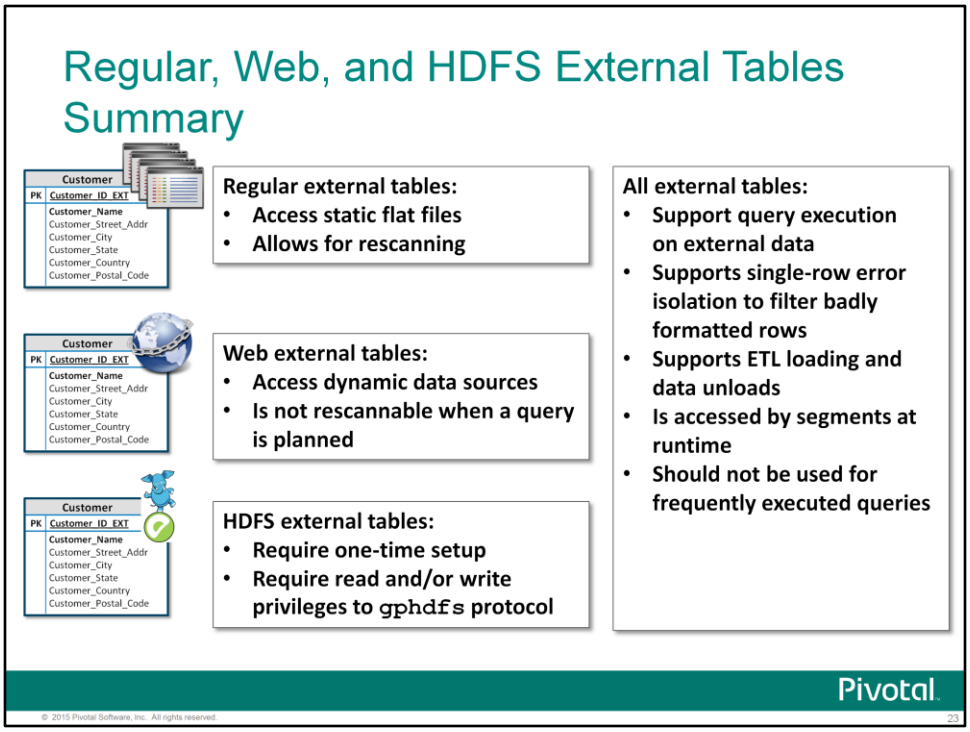
- Readable or read-only tables which are used for data loading and cannot be modified.
- Writable or write-only tables which are used for data unloading.

The protocol you use depends on the data source you are interfacing with:

- Regular or flat-file data sources are accessed using the file protocol, `file://`. The Greenplum parallel file distribution program, gpfdist, is referenced with the protocol, `gpfdist://`.
- External web tables access data sources with the HTTP protocol, `http://` or by accessing a command or script with `EXECUTE`. The command or script must exist on all segment servers if that is the method you are using.

Once an external table is defined, its data can be queried directly, and in parallel, using SQL commands. For example, you can issue `SELECT`, `JOIN`, or a `SORT` on external table data. You cannot modify the data on a readable external table.

A writable external table allows you to select rows from other database tables and output the rows to files, named pipes, or other executable programs. For example, you could unload data from Greenplum Database and send it to an executable that connects to another database or ETL tool to load the data elsewhere. Writable external tables can also be used as output targets for Greenplum parallel MapReduce calculations. Unlike a readable external table, `INSERT` operations are the only allowed operations. You cannot read from, update, delete, or truncate a writable external table.



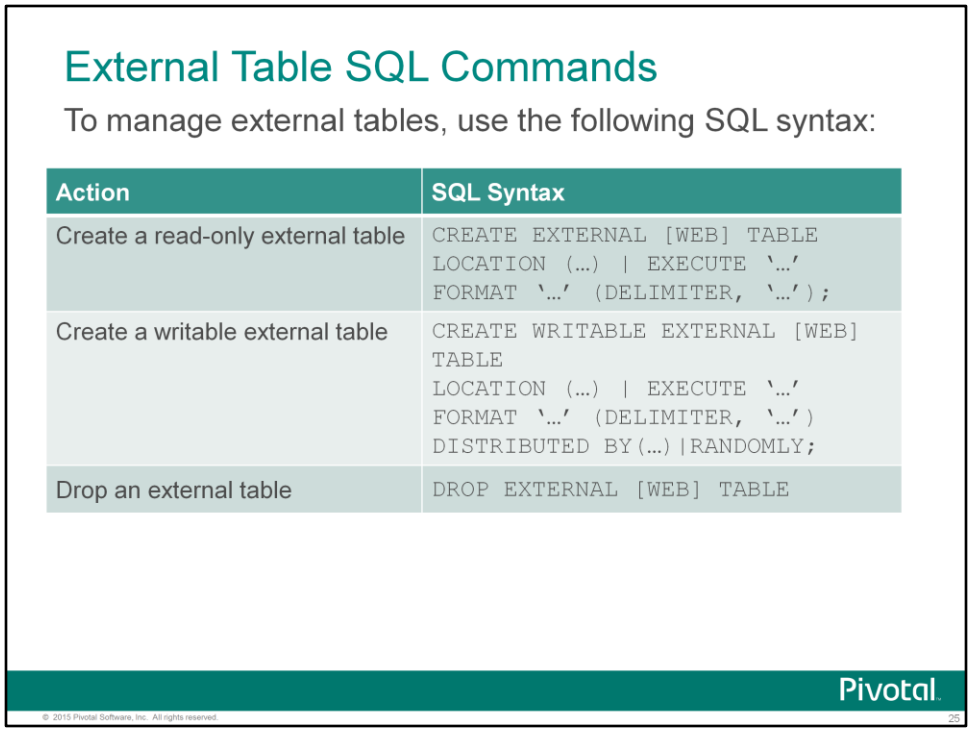
The main difference between regular external tables and web tables are their data sources:

- Regular tables access static flat files. The table is rescannable when a query is planned against the external table.
- Web tables access dynamic data sources, either on a web server or by executing OS commands or scripts. The data is not rescannable when a query is planned on the external table. This is because there is a possibility that the data will change during the course of the query's execution.
- Hadoop file system external tables lets you read and write data using the gphdfs protocol to and from Hadoop. As with other external tables, you can either read from or write to a data source. All data is read in parallel from the HDFS data nodes into the Greenplum segment servers for rapid processing of unstructured data. Each Greenplum segment instance reads one set of Hadoop data blocks and writes only the data contained within that segment to the Hadoop file system. The HDFS protocol requires an initial setup process, where Java is installed on all segment servers, Greenplum HD connectors and libraries are configured, gpadmin has read and write access to Hadoop libraries, and environmental variables and server configuration parameters are set. In addition to the initial configuration, any database users who will be using the HDFS protocol must be granted `SELECT` and/or `INSERT` privileges on the protocol if reading from or writing to data sources respectively.

Regular, Web, and HDFS External Tables Summary (continued)

An external table:

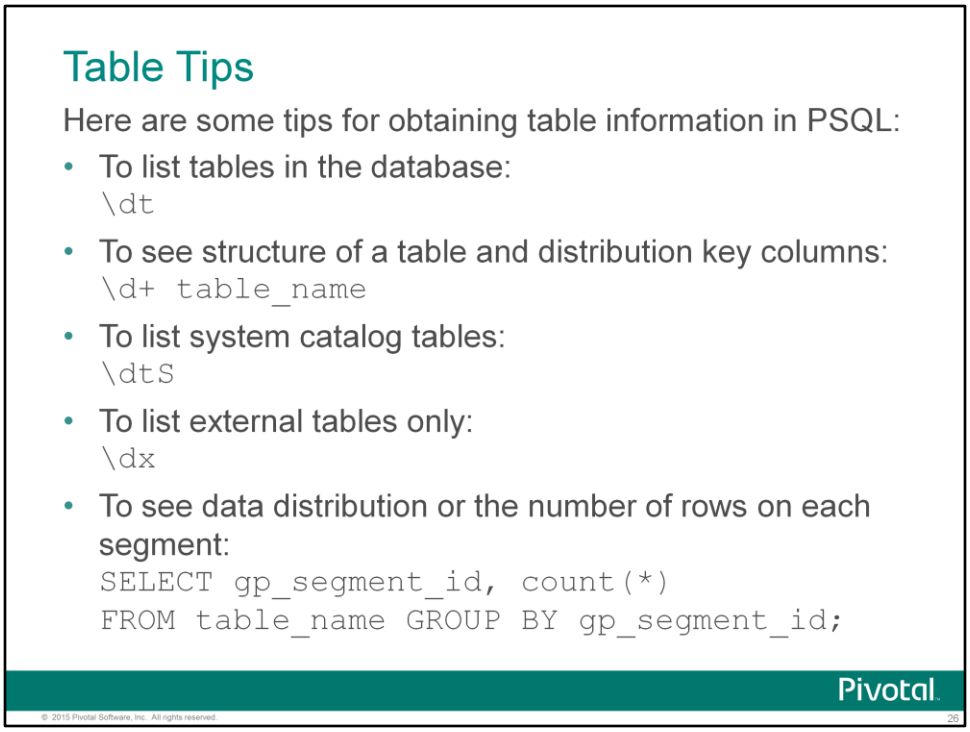
- Can be thought of as a view that allows you to run SQL query against external data without first loading the data into the Greenplum database. External tables provide an easy way to perform basic extraction, transformation, and loading (ETL) tasks that are common in data warehousing. External table files are read in parallel by the Greenplum Database segment instances, so they also provide a means for fast data loading.
- Can be defined in *single-row error isolation mode* allowing you to filter out badly formatted rows while still loading good rows.
- Is accessed by segments at runtime in parallel. The same mechanisms used for parallel query execution forms the data into tuples and processes them as a regular parallel query would.
- Should not be used for frequently queried tables. As of now, there is no way to gather detailed statistics for external tables, so the query planner is not able to plan complex queries on external tables effectively.



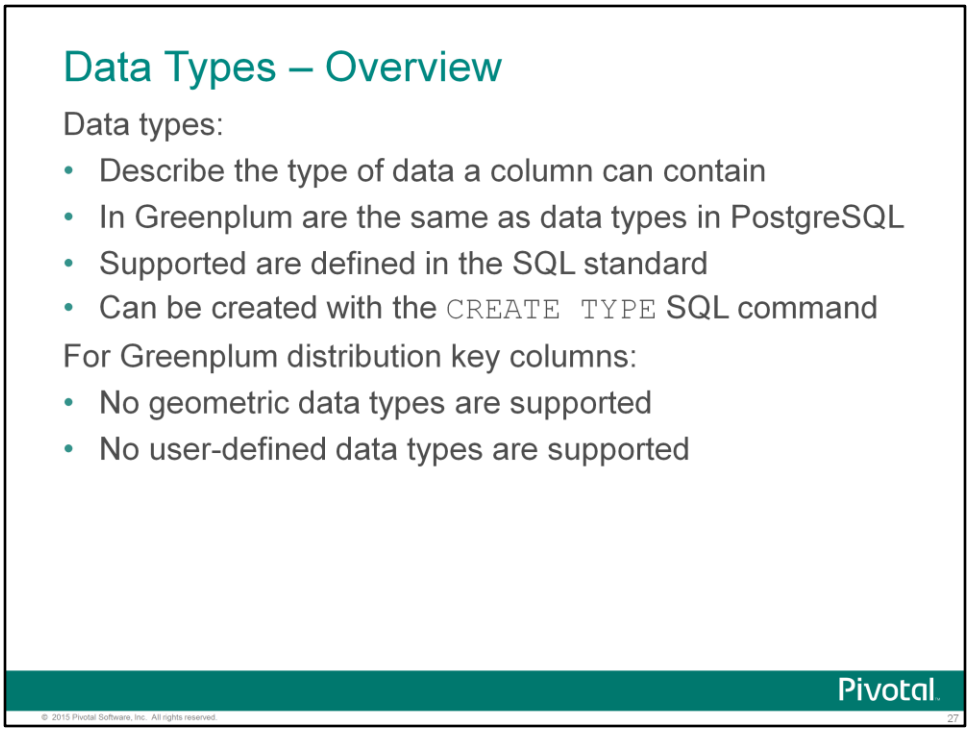
To create or remove an external table, use the Greenplum SQL syntax provided. When defining an external table:

- Table is given a name and columns are specified like a normal table definition. Unlike regular tables, external tables do not have column constraints or default values, so do not specify those.
- Use WEB to define the table as a web table. This is valid in both a read-only and write-only definition.
- The `LOCATION` clause specifies the location and type of the external data and the protocol used to access the external data (`file`, `gpfdist`, `http`, `EXECUTE` *os_command*, `gphdfs`). If using the `gpfdist://` protocol, you must have the Greenplum file distribution program, `gpfdist`, running on the host where the external data files reside. This program points to a given directory on the file host.
- The `SEGMENT REJECT LIMIT` clause is used to specify criteria for single row error handling. Normally a select from an external table is *all or nothing*, meaning it fails on the first format error encountered. This feature allows you to load good rows and ignore, or optionally log to an error table, badly formatted rows. Note this only applies to formatting errors, not constraint errors.
- `FORMAT` tells how the data is formatted, such as TEXT or CSV.

`DROP EXTERNAL TABLE` removes the table definition only – the data source remains intact and external files are not deleted.



Here are some helpful tips for working with tables in the PSQL client program.



Data types describe the type of data a table column can contain.

Greenplum supports all data types defined in the SQL standard, and all the built in data types supported in PostgreSQL. If desired, you can also define your own custom data types.

For columns that are part of the Greenplum table distribution key, they cannot be of geometric or user-defined data types.

Greenplum Database Data Types

Name	Size	Description
bigint	8 bytes	Large range integer
bigserial	8 bytes	Large autoincrementing integer
bit [(n)]	*n* bits	Fixed-length bit string
bit varying [(n)]	Actual # of bits	Variable-length bit string
boolean	1 byte	Logical boolean (true/false)
box	32 bytes	Rectangular box in the plane (*not allowed in distribution key columns)*
bytea	1 byte + binary	Variable-length binary string
character [(n)]	1 byte + *n*	Fixed-length, blank padded
character varying [(n)]	1 byte + *string size*	Variable-length with limit
cidr	12 or 24 bytes	IPV4 and IPV6 networks
circle	24 bytes	Circle in the plane (*not allowed in distribution key columns*)

Pivotal

© 2015 Pivotal Software, Inc. All rights reserved.

The table on this and the next two slides are for reference only. They describe the built-in data types available in Greenplum.

Each data type has an external representation determined by its input and output functions. Many of the built-in types have obvious external formats. However, several types are either unique to PostgreSQL and Greenplum Database, such as geometric paths, or have several possibilities for formats, such as the date and time types. Some of the input and output functions are not invertible. That is, the result of an output function may lose accuracy when compared to the original input.

For more details on the data types, including valid aliases and their defined range, refer to the *Greenplum Database Administrator Guide*.

Greenplum Database Data Types (Cont)

Name	Size	Description
date	4 bytes	Calendar date (year, month, day)
decimal [(p,s)]	variable	User-specified precision, exact
double precision	8 bytes	Variable-precision, inexact
inet	12 or 24 bytes	IPV4 and IPv6 hosts and networks
integer	4 bytes	Usual choice for integer
interval [(p)]	12 bytes	Time span
lseg	32 bytes	Line segment in the plane (*not allowed in distribution key columns)*
macaddr	6 bytes	MAC addresses
money	4 bytes	Currency amount
path	16+16*n* bytes	Geometric path in the plane (*not allowed in distribution key columns*)
point	16 bytes	Geometric point in the plane (*not allowed in distribution key columns*)

Pivotal

© 2015 Pivotal Software, Inc. All rights reserved.

Greenplum Database Data Types (Continued)

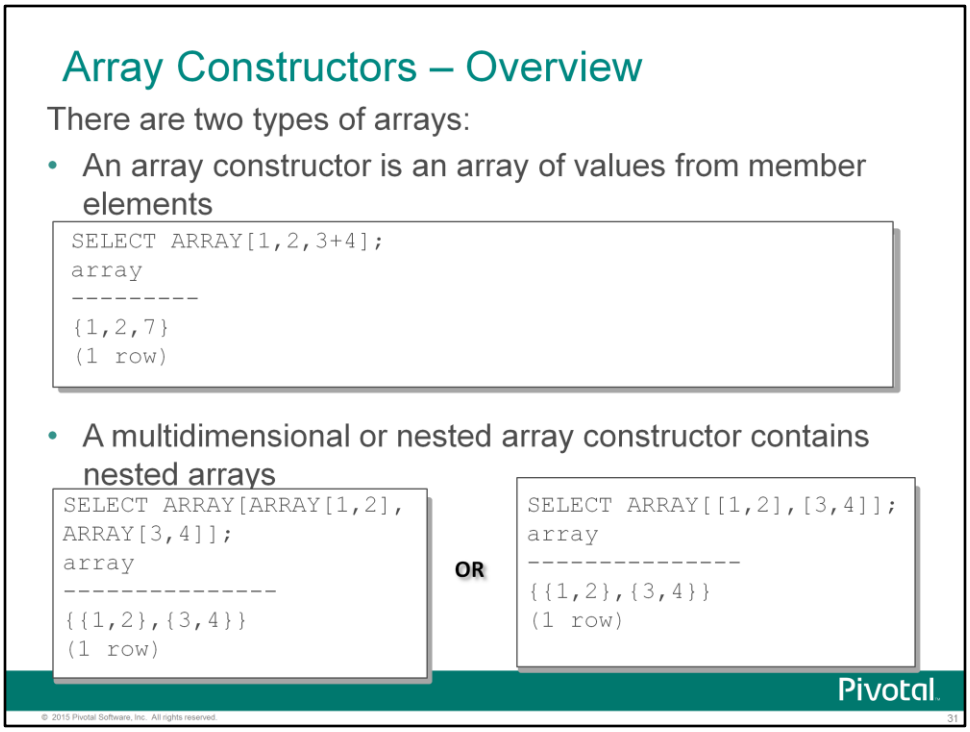
Greenplum Database Data Types (Cont)

Name	Size	Description
polygon	40+16*n* bytes	Geometric path in the plane (*not allowed in distribution key columns*)
real	4 bytes	Variable-precision, inexact
serial	4 bytes	Autoincrementing integer
smallint	2 bytes	Small range integer
text	1 byte + *string size*	Variable unlimited length
time [(p)] [without time zone]	8 bytes	Time of day only
time [(p)] [with time zone]	12 bytes	Time of day only, with time zone
timestamp [(p)] [without timezone]	8 bytes	Both date and time
timestamp [(p)] [with timezone]	8 bytes	Both date and time, with time zone

Pivotal

© 2015 Pivotal Software, Inc. All rights reserved.

Greenplum Database Data Types (Continued)



An array constructor is an expression that builds an array value from values for its member elements.

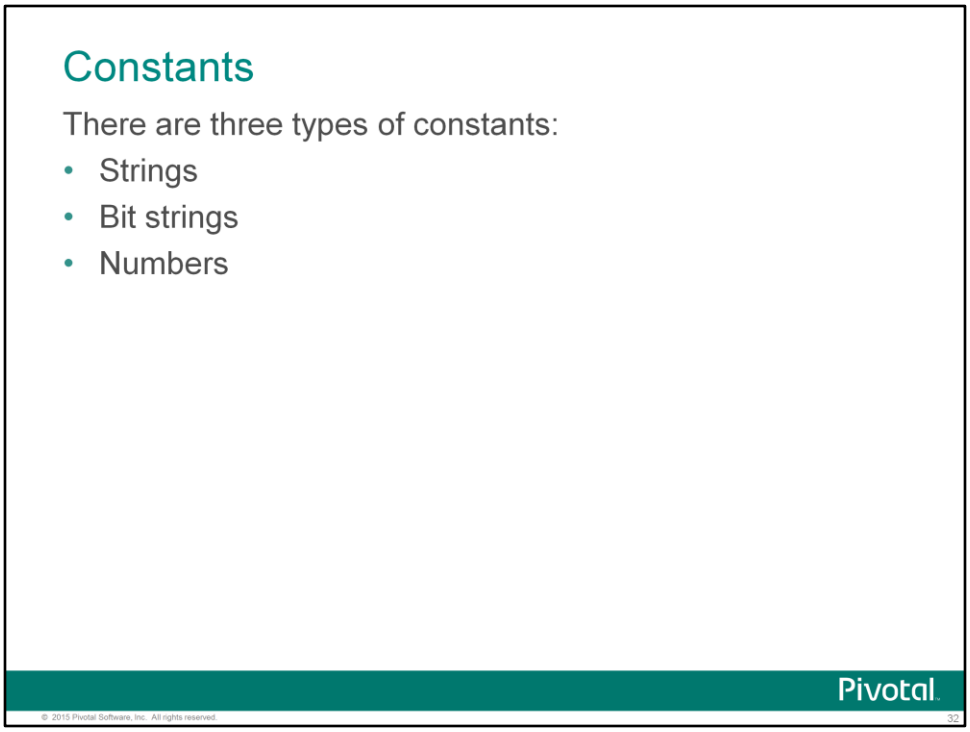
A simple array constructor consists of:

- The key word ARRAY
- A left square bracket (`[`)
- One or more comma separated expressions for the array element values
- A right square bracket (`]`)

For the example shown, there are three items in the array – the last item is a sum of two numbers.

Multidimensional arrays can be built by nesting array constructors. In the inner constructors, the key word `ARRAY` can be omitted.

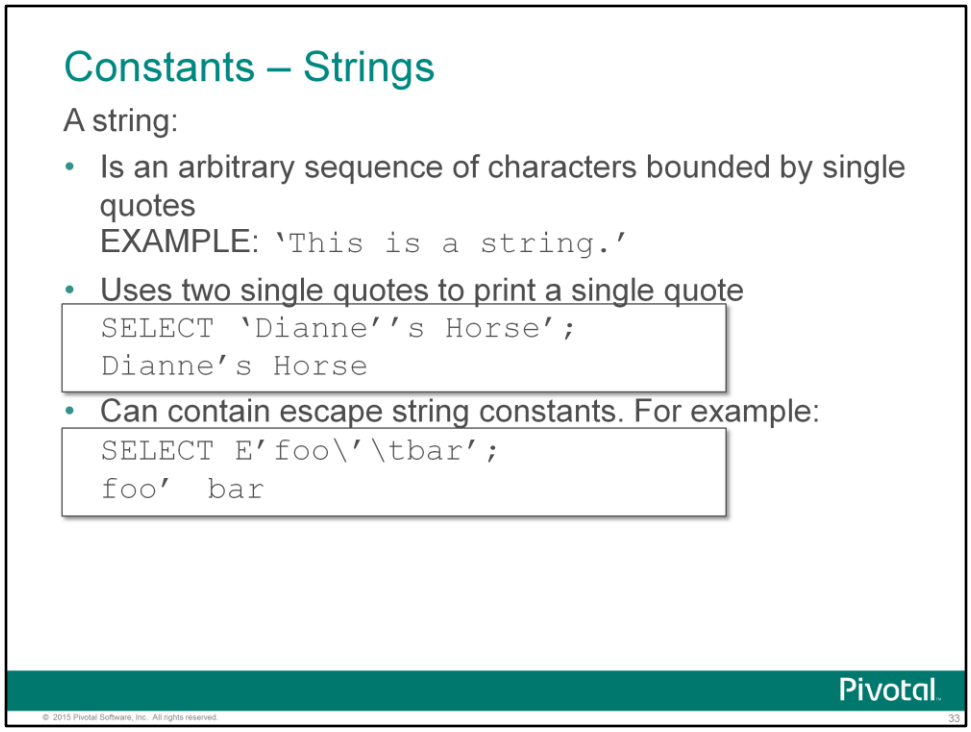
In the examples shown at the bottom of the slide, the syntax shown with two ARRAY definitions is the same as the syntax shown with one ARRAY definition and nested brackets.



There are three kinds of implicitly-typed constants or literals in PostgreSQL:

- Strings
- Bit strings
- Numbers

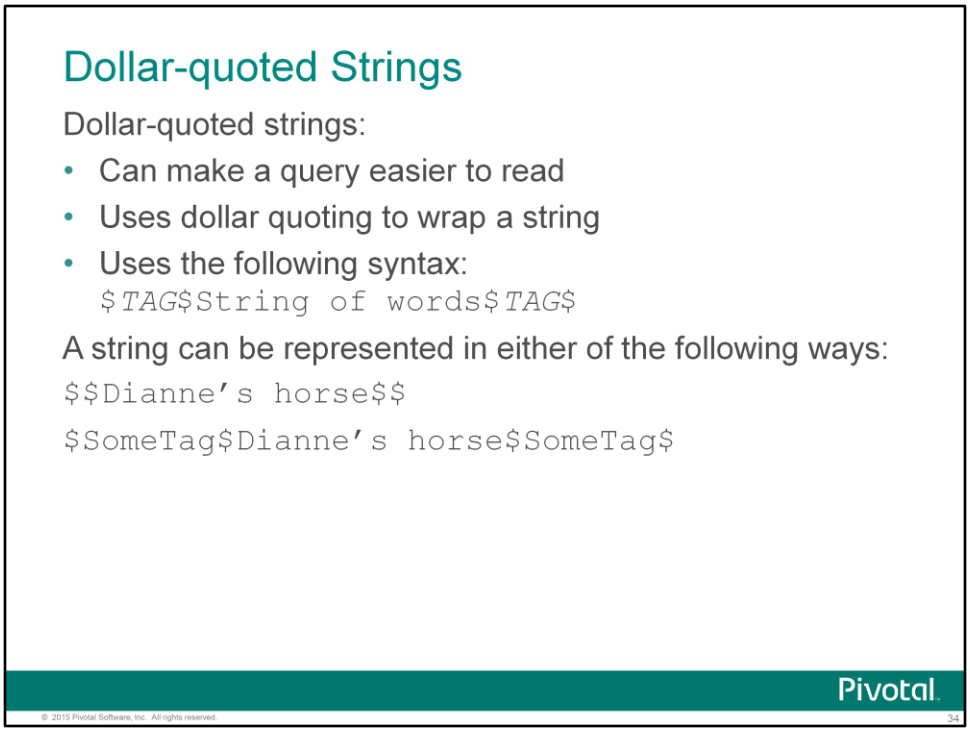
Constants can be specified with explicit types which enable more accurate representation and more efficient handling of the constant by the system.



A string constant in SQL is an arbitrary sequence of characters bounded by single quotes (’), for example `’This is a string’`.

To include a single-quote character within a string constant, write two adjacent single quotes, for example, `’Dianne”s horse’`.

PostgreSQL also accepts *escape* string constants, which are an extension to the SQL standard.

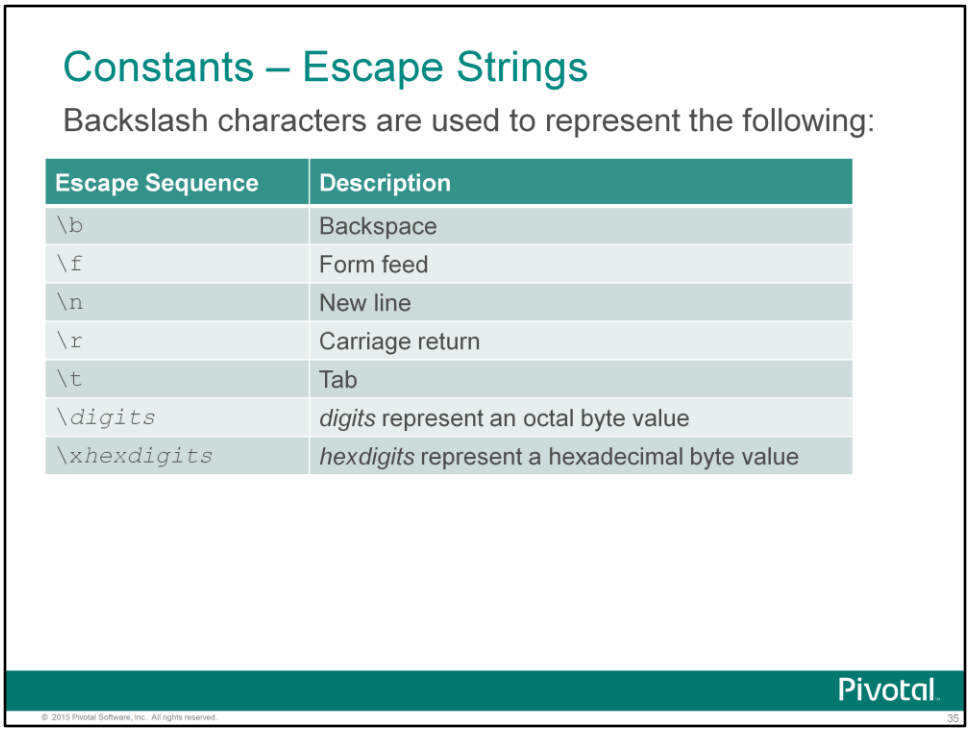


The standard syntax for specifying string constants is usually convenient, but it can be difficult to read, because of doubled quotes or backslashes.

To allow for more readable queries in such situations, PostgreSQL provides another way to write string constants, called *dollar quoting*.

A dollar-quoted string constant consists of:

- A dollar sign (`$`)
- An optional "tag" of zero or more characters
- Another dollar sign
- An arbitrary sequence of characters that makes up the string content
- A dollar sign
- The same tag that began this dollar quote
- A dollar sign

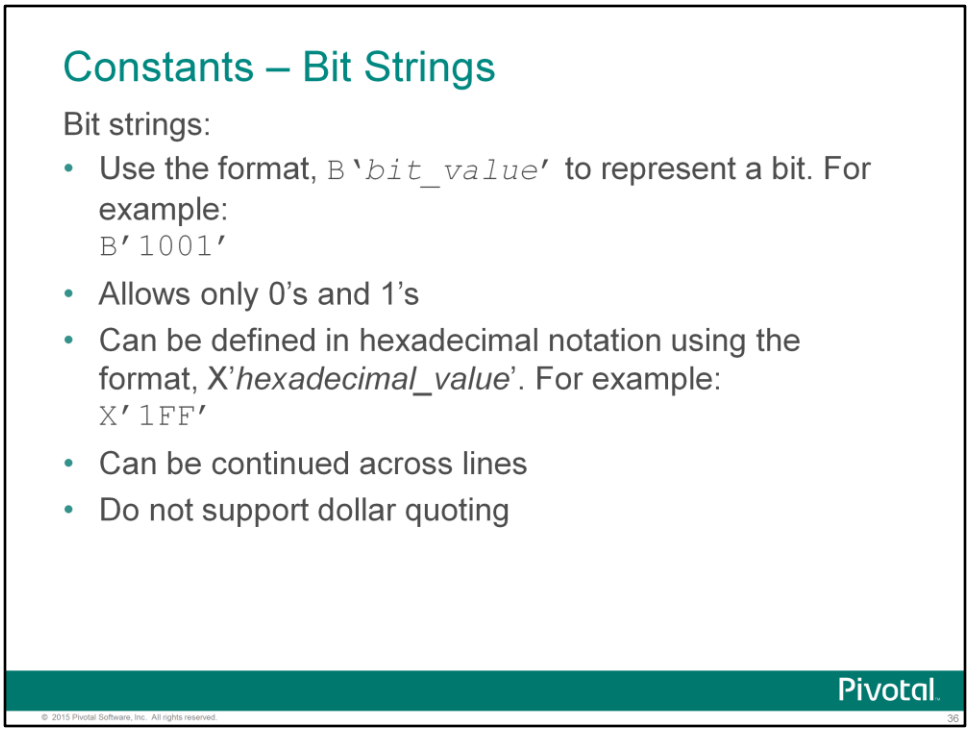


Within an escape string, a backslash character (\) begins a C-like backslash escape sequence, in which the combination of backslash and following characters represent a special byte value:

- `\b` is a backspace
- `\f` is a form feed
- `\n` is a newline
- `\r` is a carriage return
- `\t` is a tab

Also supported are:

- `\digits`, where digits represents an octal byte value
- `\xhexdigits`, where hexdigits represents a hexadecimal byte value.



Bit-string constants look like regular string constants with a B (upper or lower case) immediately before the opening quote (no intervening whitespace), e.g., `B'1001'`.

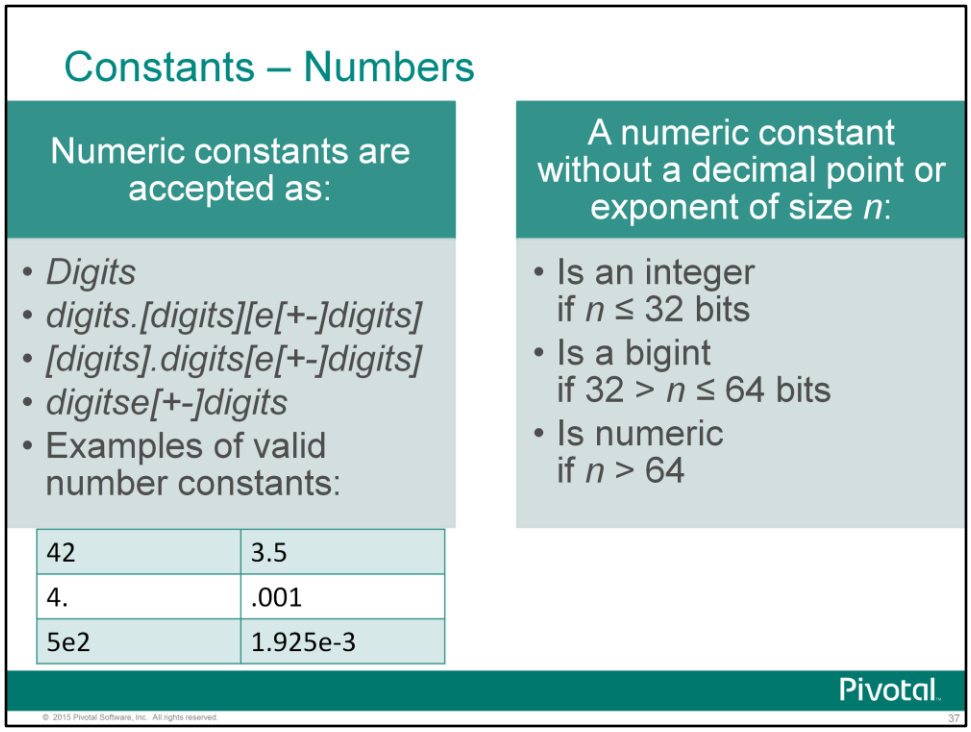
The only characters allowed within bit-string constants are 0 and 1.

Alternatively, bit-string constants can be defined in hexadecimal notation, using a leading X (upper or lower case), e.g., `X'1FF'`.

This notation is equivalent to a bit-string constant with four binary digits for each hexadecimal digit.

Both forms of bit-string constant can be continued across lines in the same way as regular string constants.

Dollar quoting cannot be used in a bit-string constant.



Numeric constants are accepted in these general forms:

- digits
- digits.[digits][e[+-]digits]
- [digits].digits[e[+-]digits]
- digitse[+-]digits

where digits is one or more decimal digits, 0 through 9.

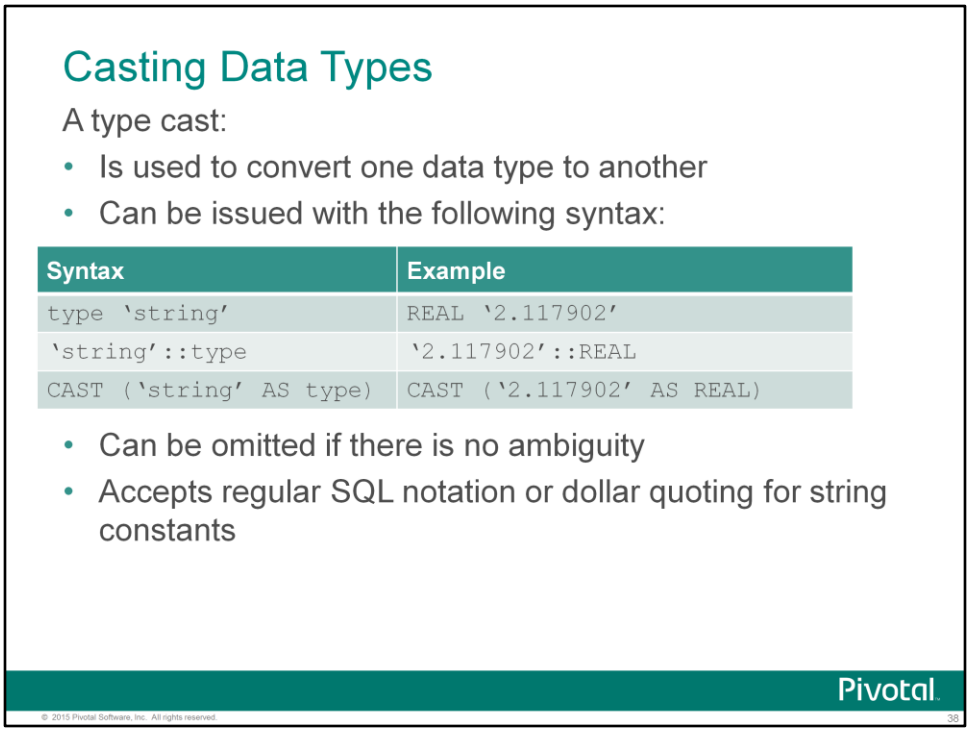
The following requirements must be met:

- At least one digit must be before or after the decimal point, if one is used.
- At least one digit must follow the exponent marker (e), if one is present.
- There cannot be any spaces or other characters embedded in the constant.

A numeric constant that contains neither a decimal point nor an exponent:

- Is presumed to be type `integer` if its value fits in the integer size, 32 bits.
- Is presumed to be type `bigint` if its value fits in bigint size, 64 bits.
- Is assumed to be numeric type if none of the above holds true

Constants that contain decimal points and/or exponents are always initially presumed to be type numeric.



A type cast specifies a conversion from one data type to another.

A constant of an arbitrary type can be entered using any one of the following explicit notations:

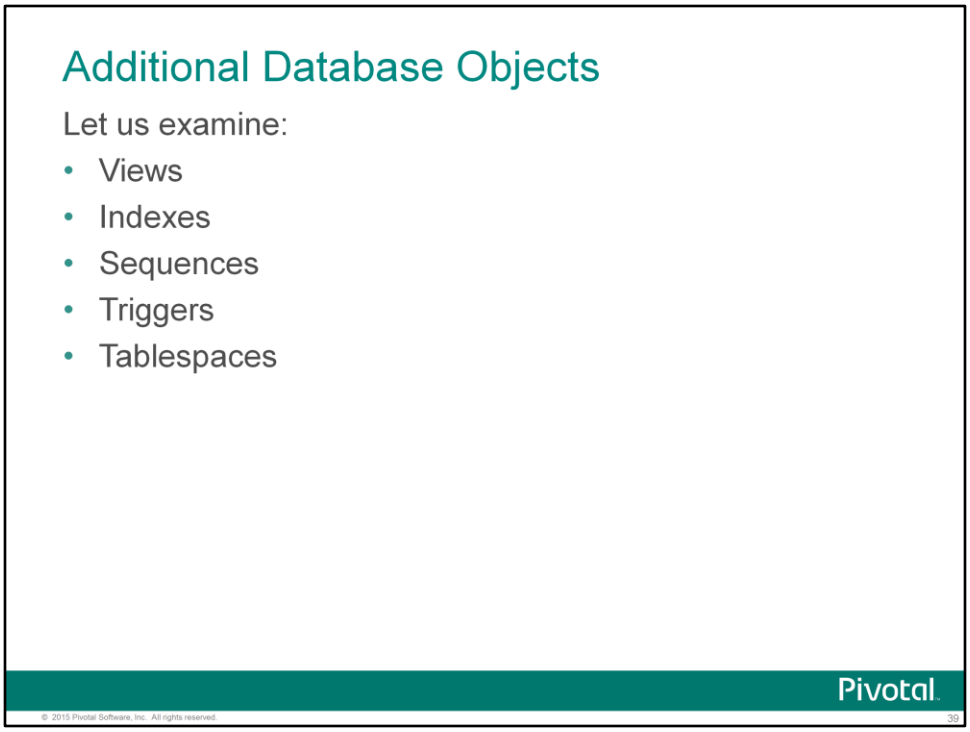
- `type 'string'` — `REAL '2.117902'`
- `'string'::type` — `'2.117902'::REAL`
- `CAST ( 'string' AS type )` — `CAST('2.117902' AS REAL)`

The result is a constant of the indicated type.

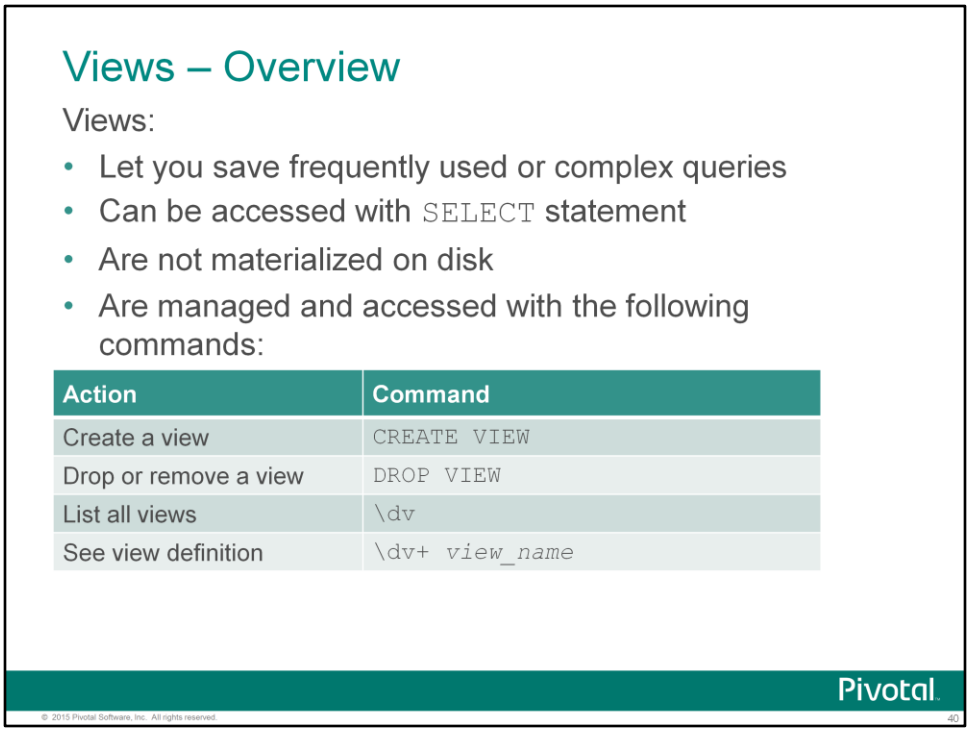
The CAST syntax conforms to SQL; the syntax with :: is historical PostgreSQL usage.

If there is no ambiguity the explicit type cast can be omitted (for example, when constant it is assigned directly to a table column), in which case it is automatically converted to the column type.

The string constant can be written using either regular SQL notation or dollar-quoting.



Now we will briefly cover the other database objects we haven't covered yet.



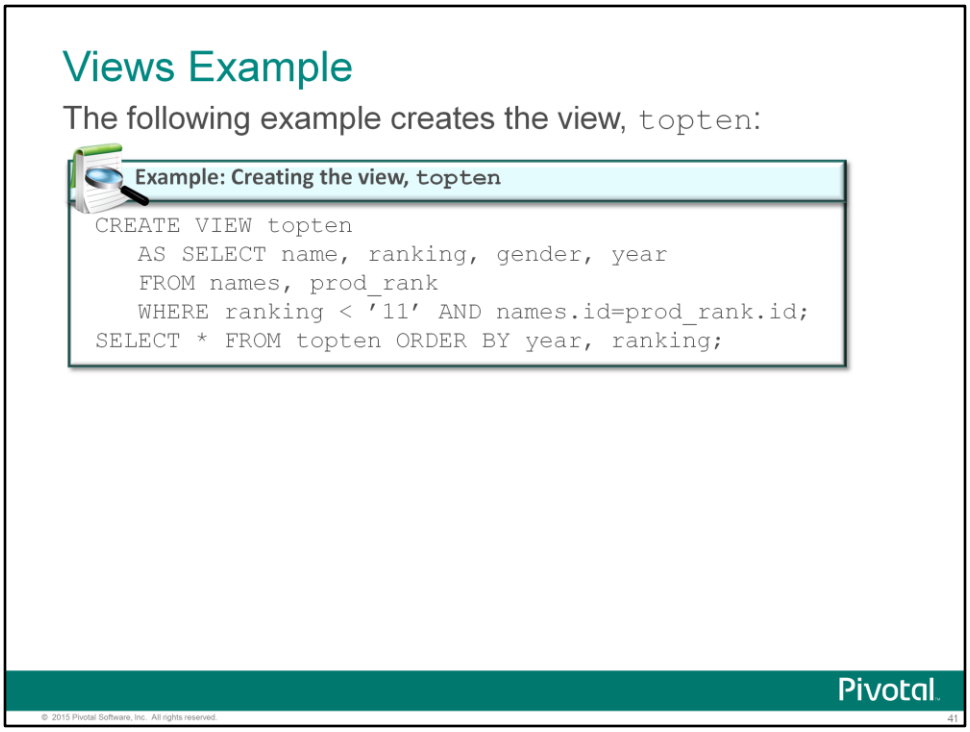
Views are a way to save frequently used or complex queries and then access them in a SELECT statement as if they were a table. A view is not physically materialized on disk as is a table. The query is instead ran whenever the view is accessed.

Note that currently Views ignore ORDER BY or SORT operations stored in the view, so run those in the query that calls the view.

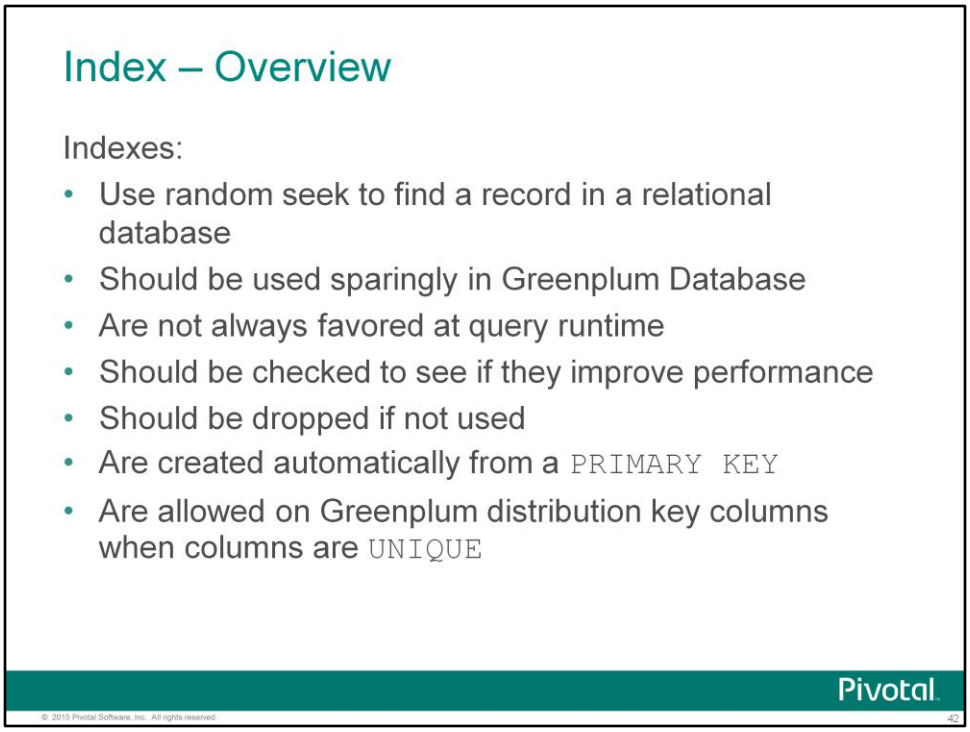
Working with views in Greenplum Database is the same as in PostgreSQL or as you would expect in any other DBMS.
You can access and manage views with the following commands:

- `CREATE VIEW` – Creates a view
- `DROP VIEW` – Drops a view
- `\dv` – Meta-command to list all of the views in a database
- `\dv+ view_name` – Describes the view

At the bottom of the slide, you will find an example of a view being created.



In this example, the view, `topten`, is created based on two tables: `names` and `prod_rank`. The data is restricted where the ID exists in both tables and ranking field is less than 11. After creating the view, you can issue a `SELECT` statement to view the rows extracted from the original table.



Greenplum Database is very fast at sequential scanning. Unlike a traditional database, the data is distributed so that each segment scans a smaller portion of the overall data in order to get the result. Add in partitioning and the total data to scan gets even smaller. An index, which uses a random seek to find a record may not always improve query performance in Greenplum Database.

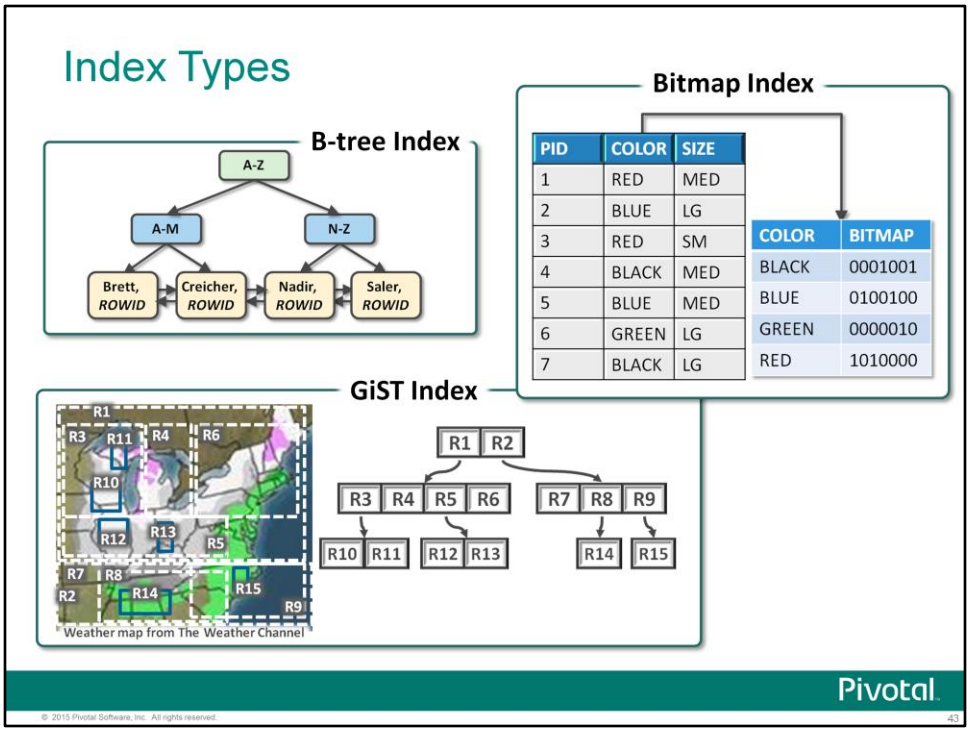
When deciding what indexes to create, examine the `WHERE` clauses of your query workload. On very large tables, these columns might still be good candidates for indexes.

Note that the query planner tends to favor a sequential scan over an index scan because a sequential scan is often determined by the planner to be faster. If the planner determines an index to be faster, it will use it.

An index does add some database overhead (the index must be maintained whenever a table is updated) so you want to ensure that the indexes you create are being used by your query workload and that they are improving query performance as compared to a sequential scan of the table.

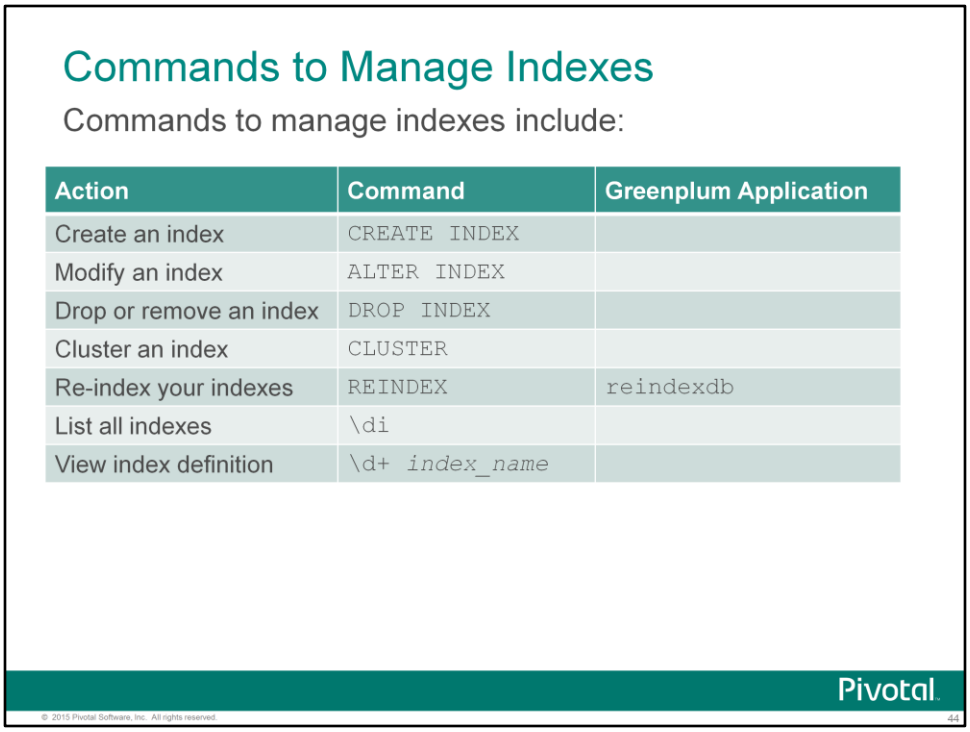
Greenplum Database does have one limitation regarding `UNIQUE` indexes. To enforce the uniqueness of an index across the segments, unique indexes are only allowed on distribution key columns.

If you create a table with a `PRIMARY KEY`, a `pkey` index will be automatically created for that table.



Each index type uses a different algorithm that is best suited to different types of queries. Greenplum Database supports the following index types:

- B-tree (the default index type) fits the most common situations. B-trees can handle equality and range queries on data that can be sorted into some ordering.
- Bitmap is a new type of index only found in Greenplum and not in PostgreSQL.
- GiST indexes are not a single kind of index, but rather an infrastructure within which many different indexing strategies can be implemented.



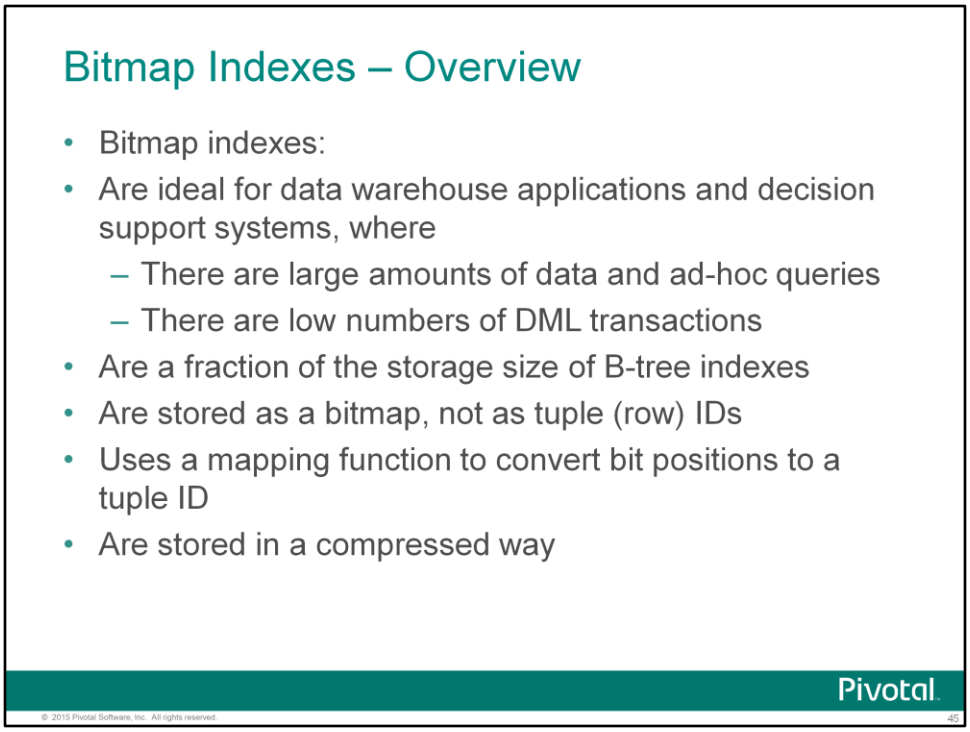
You create, modify, or delete an index using the `CREATE INDEX`, `ALTER INDEX`, and `DROP INDEX` commands respectively.

`CLUSTER` physically reorders a table on disk based on the index information. This can be helpful for queries that access a range of data (by date for example – get June1 –June 30 2007). When an index is in order, the system does not have to hop around the disk seeking for records, but can access a range of records in sequence. Although the `CLUSTER` command is supported in Greenplum Database, it takes a long time to physically reorder large indexes across several segments. It is much faster to use `CREATE TABLE …AS SELECT … ORDER BY <the index criteria>` to physically reorder the table on disk in this situation.

The SQL command, `REINDEX`, and the Greenplum application, `reindexdb` are used to rebuild an index using the data stored in a table and replaces the old copy of an index. This should be used when:

- An index has become corrupted, and no longer contains valid data.
- An index has become bloated, that it is contains many empty or nearly empty pages. This can occur with B-tree indexes in Greenplum Database under certain uncommon access patterns.
- You have altered the fill factor storage parameter for an index and wish to ensure that the change has taken full effect.

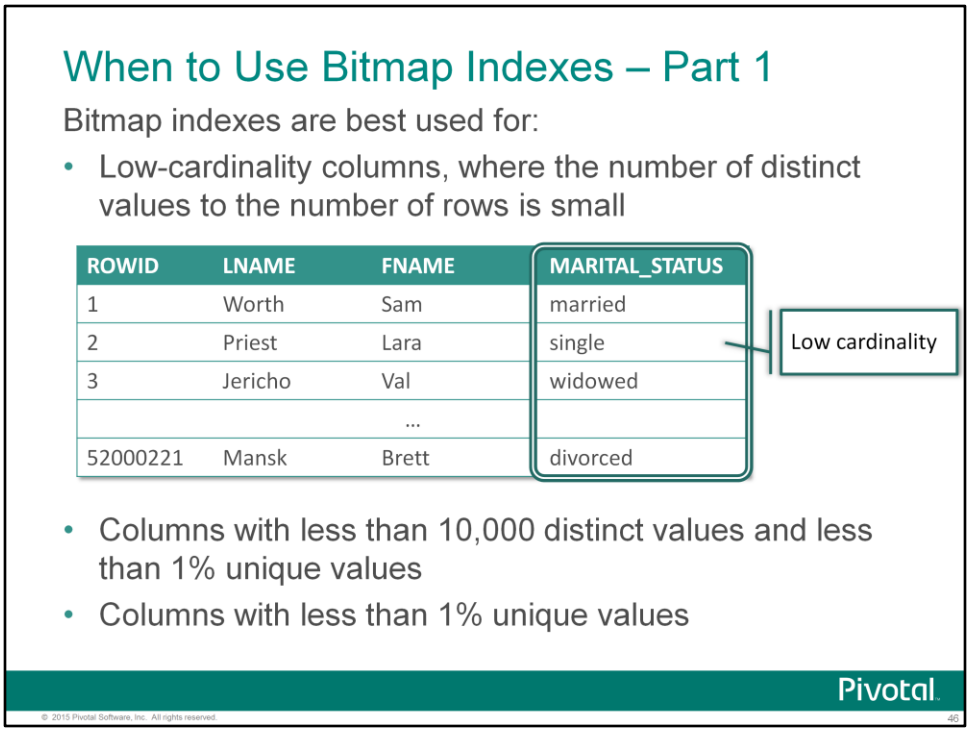
Use the `\di` and `\d+ index_name` meta-commands to list all of the indexes in a database and view details on the indexes respectively.



Bitmap indexes are one of the most promising strategies for indexing high dimensional data in data warehousing applications and decision support systems. These types of applications typically have large amounts of data and ad hoc queries, but a low number of DML transactions.

Fully indexing a large table with a traditional B-tree index can be expensive in terms of space because the indexes can be several times larger than the data in the table. Bitmap indexes are typically only a fraction of the size of the indexed data in the table.

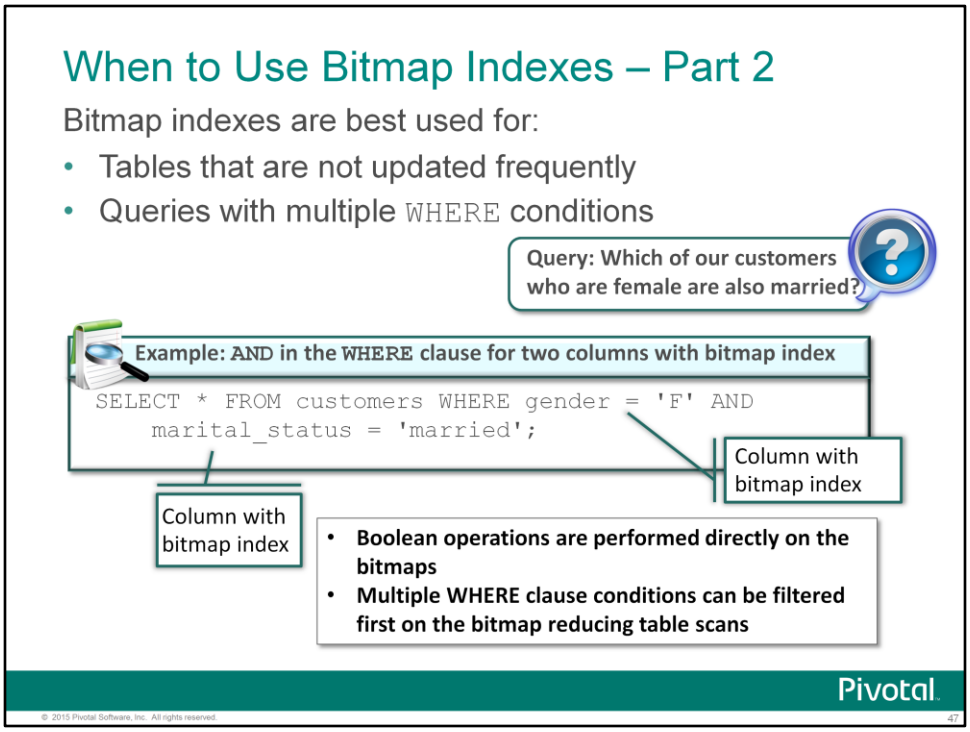
An index provides pointers to the rows in a table that contain a given key value. A regular index stores a list of tuple ids for each key corresponding to the rows with that key value. In a bitmap index, a bitmap for each key value replaces a list of tuple ids. Each bit in the bitmap corresponds to a possible tuple id, and if the bit is set, it means that the row with the corresponding tuple id contains the key value. A mapping function converts the bit position to an actual tuple id, so that the bitmap index provides the same functionality as a regular index. Bitmap indexes store the bitmaps in a compressed way. If the number of distinct key values is small, bitmap indexes compress better and the space saving benefit compared to a B-tree index becomes even better.



Bitmap indexes perform best for columns in which the ratio of the number of distinct values to the number of rows in the table is small. This ratio is known as the degree of cardinality. A marital status column, which has only a few distinct values (`single`, `married`, `divorced`, or `widowed`), is a good choice for a bitmap index, and is considered low cardinality.

However, columns with higher cardinalities can also have bitmap indexes. For example, on a table with one million rows, a column with 10,000 distinct values is also a good candidate for a bitmap index. A bitmap index on this column can outperform a B-tree index, particularly when this column is often queried in conjunction with other indexed columns. However, the performance gains start to diminish on columns with 10,000 or more unique values, regardless of the number of rows in the table.

A best practice rule to follow is that when less than 1% of the row count is distinct, use bitmap indexes.

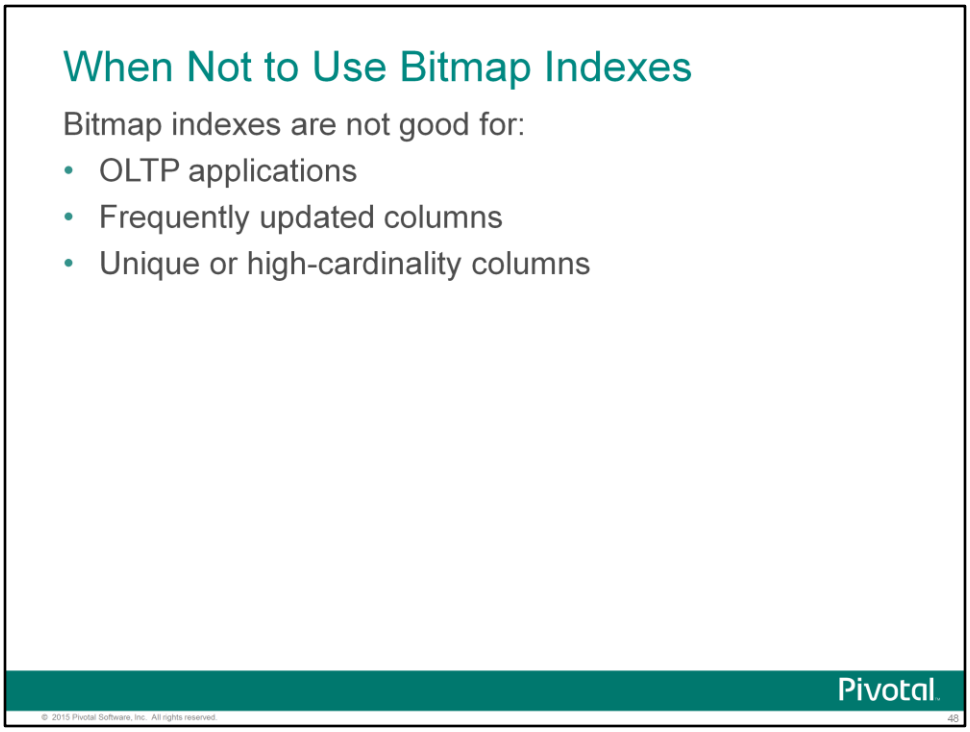


Bitmap indexes can dramatically improve query performance for ad-hoc queries. `AND` and `OR` conditions in the `WHERE` clause of a query can be resolved quickly by performing the corresponding Boolean operations directly on the bitmaps before converting the resulting bitmap to tuple ids. If the resulting number of rows is small, the query can be answered quickly without resorting to a full table scan.

Bitmap indexes are most effective for queries that contain multiple conditions in the `WHERE` clause. Rows that satisfy some, but not all, conditions are filtered out before the table itself is accessed. This improves response time, often dramatically. FOR EXAMPLE - An analyst for the company might want to know, "Which of our customers who are female are also married?" This corresponds to the following SQL query:

```
SELECT * FROM customers WHERE gender = 'F' AND
marital_status = 'married';
```

The bitmaps for gender='F' and marital_status='married' are used to identify the rows that have the correct bit values, and the results are merged before accessing the actual tuple id in the customer table. This is more efficient than scanning the entire table multiple times, as only the exact rows that are needed are retrieved.



Bitmap indexes should not be used for unique columns or columns with high cardinality data, such as customer names or phone numbers. B-tree indexes are a better choice for these types of columns.

Bitmap indexes are primarily intended for data warehousing applications where users query the data rather than update it. They are not suitable for OLTP applications with large numbers of concurrent transactions modifying the data.

Bitmap Index – Example

user_id	gender=M	gender=F
1	1	0
2	0	1
3	1	0
4	1	0
5	1	0
6	0	1
7	1	0
8	0	1

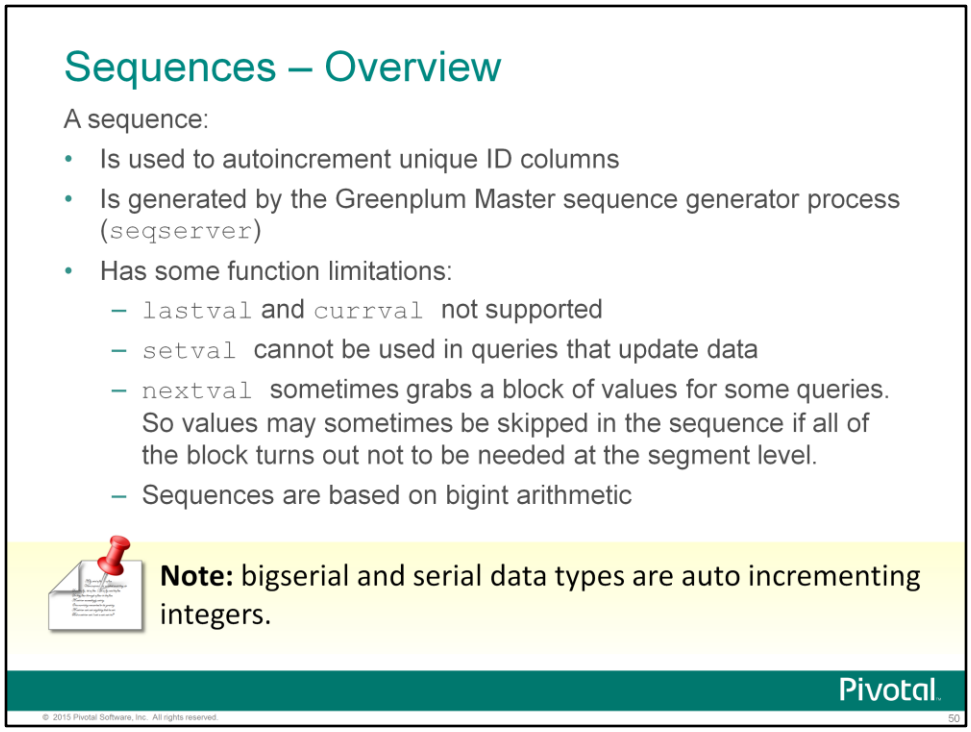
Bitmap for **gender=M**: 10111010
Bitmap for **gender=F**: 01000101

Pivotal

© 2015 Pivotal Software, Inc. All rights reserved.

Here is a simple table that illustrates a bitmap index for the gender column.

Each entry (or bit) in the bitmap corresponds to a single row of the table. The value of each bit depends upon the values of the corresponding row in the table. For example, the bitmap for gender='M' would look like this. It contains a '1' as its first bit because the gender is 'M' in the first row of the customer table. The bitmap index also keeps a list of values, which maps the bit to the actual column value. At runtime the value is mapped to a bit number, the bitmap is scanned, rows are identified by their position in the bitmap, and only the tuple ids that match the scan are retrieved.



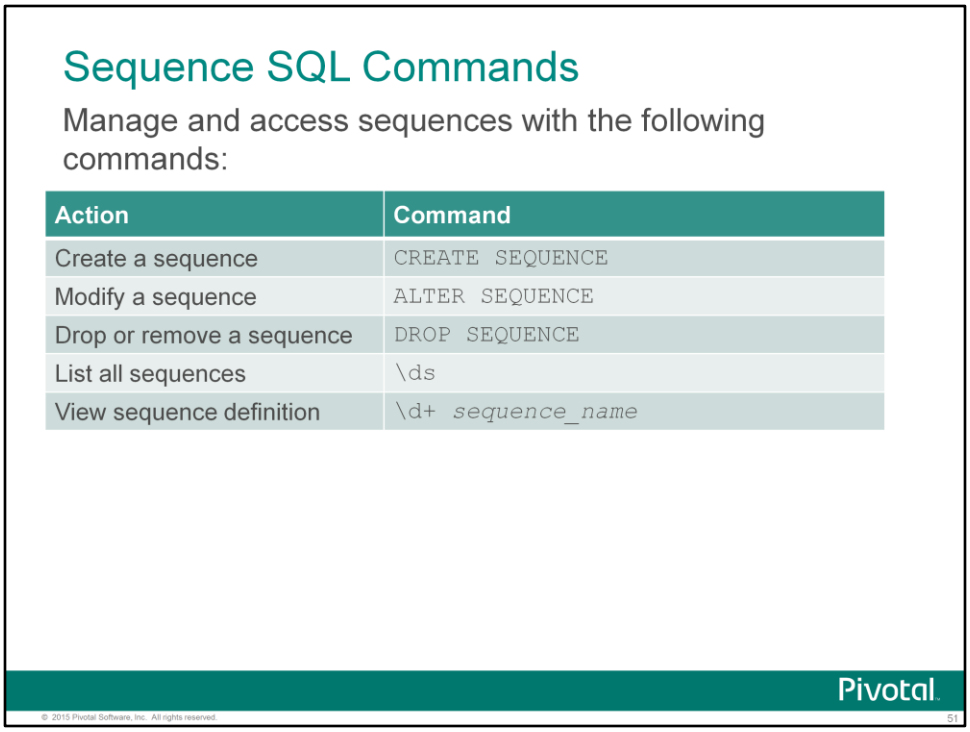
Creating a sequence generator involves creating and initializing a new special single-row table with the given sequence name. Sequences are often used to increment unique ID columns of a table whenever a new record is added.

In a regular, non-distributed, database, functions that operate on the sequence go to the local sequence table to get values as they are needed. In Greenplum, each segment is its own distinct PostgreSQL database. Therefore they need a single point of truth to go for sequence values so that all segments get incremented correctly and the sequence moves forward in the right order. A sequence server process runs on the master and is the point-of-truth for a sequence in Greenplum distributed database. Segments get sequence values at runtime from the master.

Because of this distributed sequence design, there are some limitations on the functions that operate on a sequence in Greenplum Database:

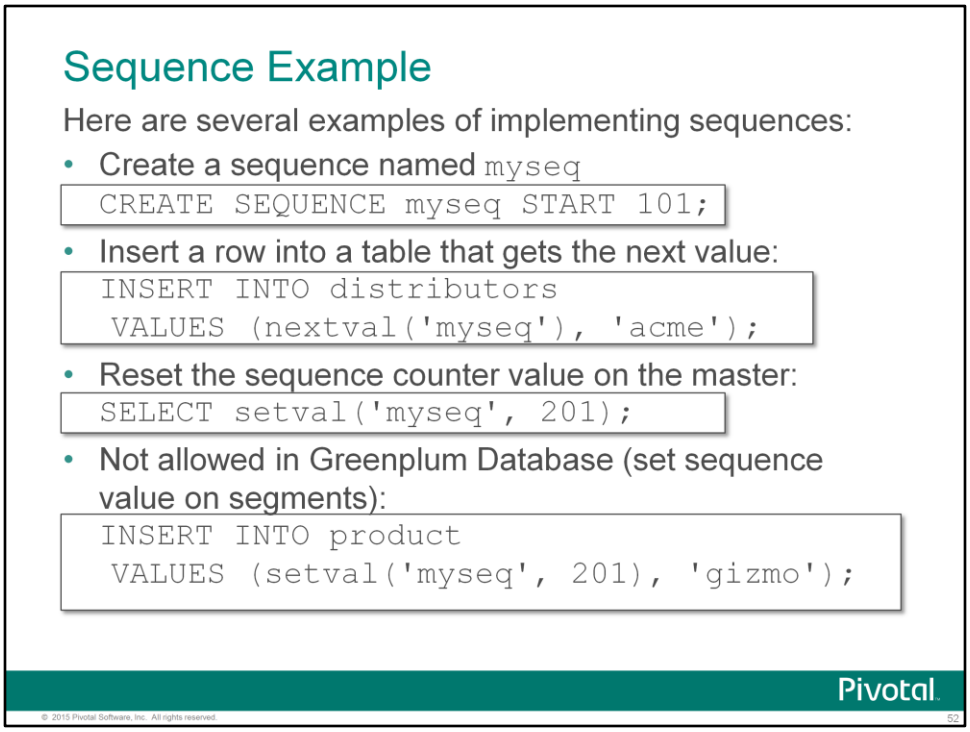
- `lastval` and `currval` are not supported.
- `setval` can only be used to set the value of the sequence generator on the master, it cannot be used in subqueries to update records on distributed table data.
- `nextval` will sometimes grab a block of values from the master for a segment to use, depending on the query. Values may sometimes be skipped in the sequence if all of the block turns out not to be needed at the segment level. Note that regular PostgreSQL does this as well, so this is not unique to Greenplum.

The bigserial and serial data types are auto incrementing integers. These can be used like sequences.



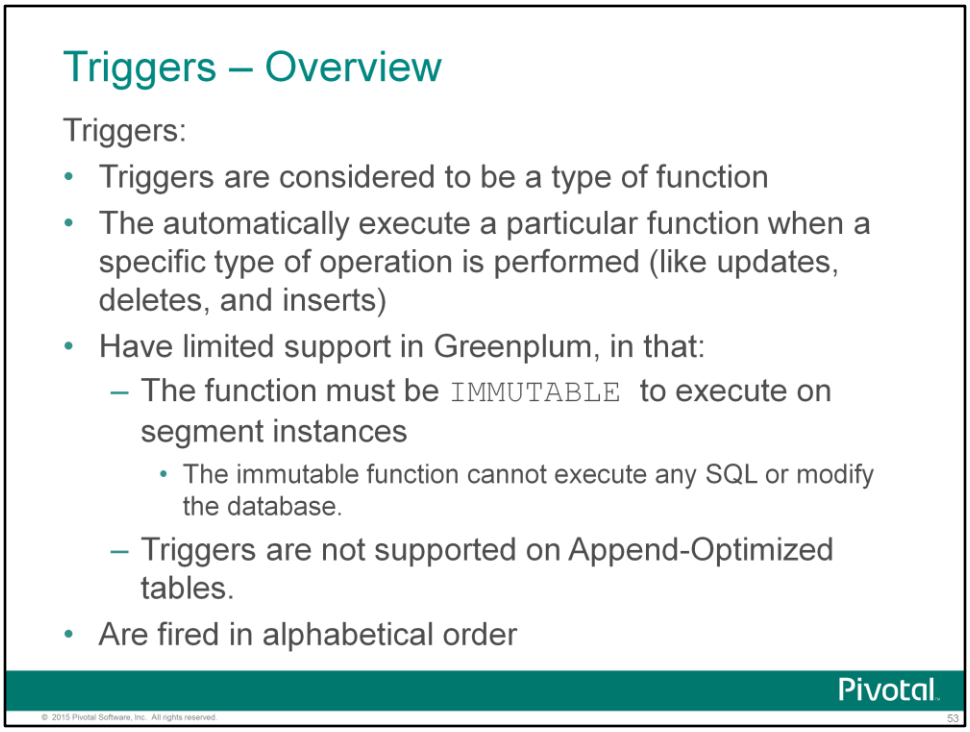
Working with sequences in Greenplum is the same as in PostgreSQL or as you would expect in any other DBMS. Creating, altering, and dropping an index are managed with SQL commands.

Use the `\ds` and `\d+` sequence_name meta-commands to list all of the sequences in a database and view the details on a sequence respectively.



The examples on the slide show how to:

- Create a sequence using the `CREATE SEQUENCE` syntax.
- Insert a row into the distributors table and call the next value of the sequence created. That value is inserted in the first column of the table.
- Reset the sequence counter to the value shown in the slide, `201`.
- Reset the value of the sequence to `201` and insert that value into the `product` table.

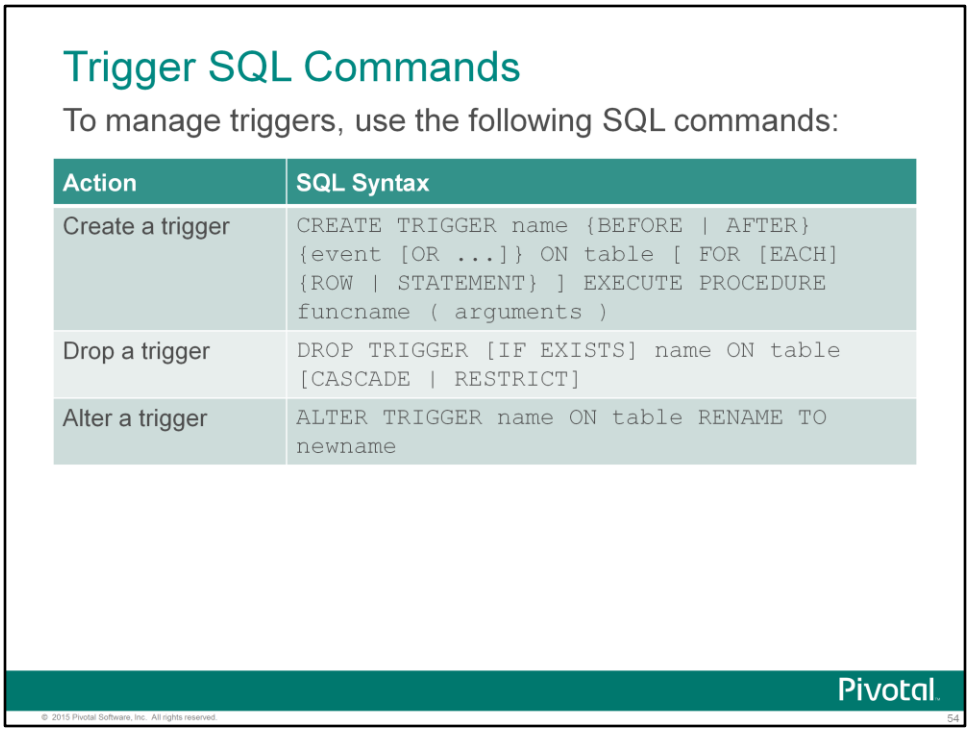


Triggers are functions or procedural code that automatically executes whenever a specific type of operation is performed in the database. Triggers are often used for maintaining the integrity of data within the database.

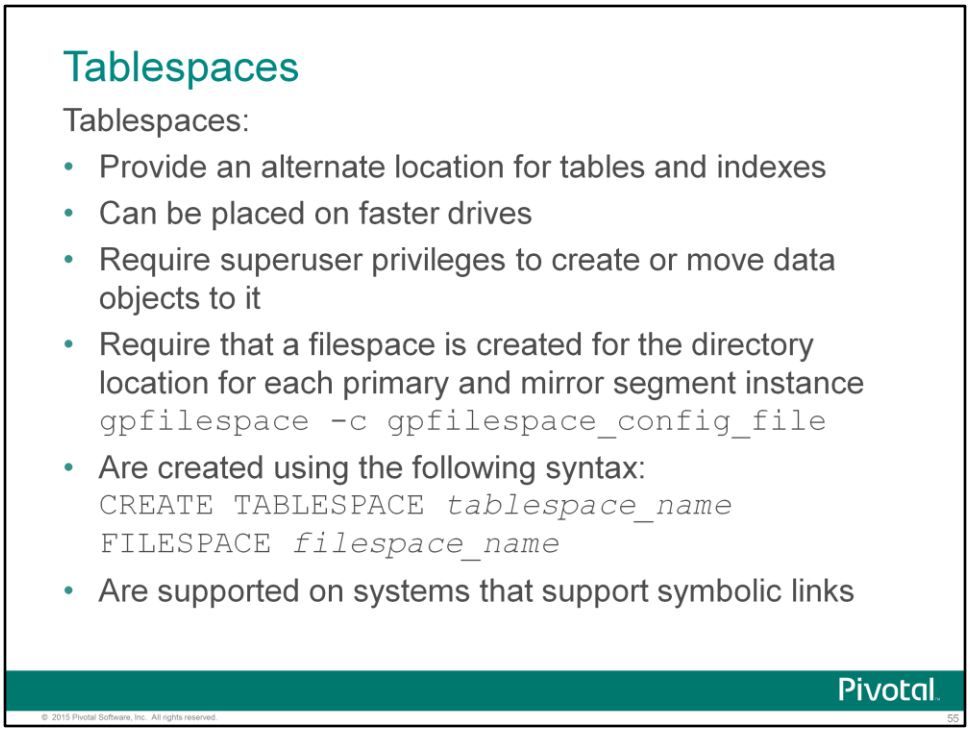
Due to the distributed nature of a Greenplum Database system, the use of triggers is somewhat limited. The function used in the trigger must be `IMMUTABLE`, meaning it cannot use information not directly present in its argument list. The function specified in the trigger also cannot execute any SQL or modify the database in any way. Given that triggers are most often used to alter the database (for example, update these other rows when this row is updated), these limitations offer very little practical use of triggers in Greenplum Database.

If multiple triggers of the same kind are defined for the same event, they will be fired in alphabetical order by name.

Triggers are not supported on append-optimized tables.



To create, drop, or modify a trigger, use the `CREATE TRIGGER`, `DROP TRIGGER`, and `ALTER TRIGGER` commands respectively.



A tablespace allows superusers to define an alternative location on the file system where the data files containing database objects, such as tables and indexes, may reside. This usage allows administrators to store specific tablespaces on faster drives, if the service level agreement requires it.

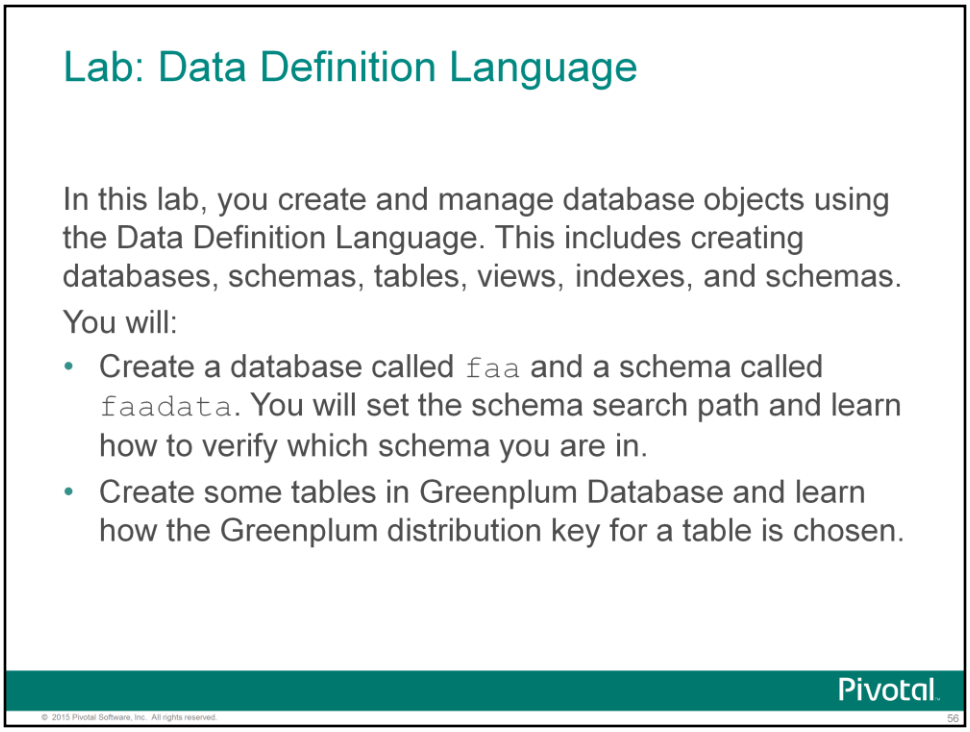
A user with appropriate privileges can pass the tablespace name to `CREATE DATABASE`, `CREATE TABLE`, or `CREATE INDEX` to have the data files for these objects stored within the specified tablespace.

In Greenplum Database, you must first create a filespace before creating a tablespace that is associated with the directories defined in the filespace. The filespace is the directory location for each segment instance (primary and mirror) in your Greenplum Database array. You use the `gpfilespace` command to first create the filespace. You can use the same location for all you primaries if you want and a separate location for your mirrors (if they are using the same set of hosts as your primaries). The directory must be created before creating the tablespace. Use the `gpfilespace` command to create that filespace.

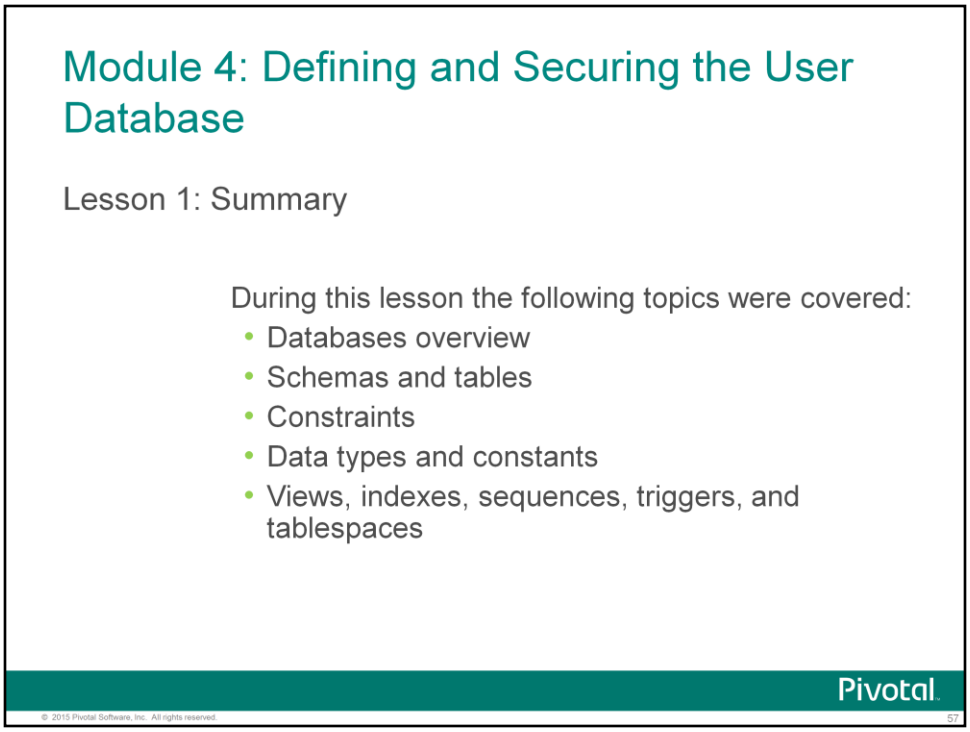
Tablespaces are only supported on systems that support symbolic links.

Note that the Greenplum implementation of tablespaces is an extension of the SQL standard.

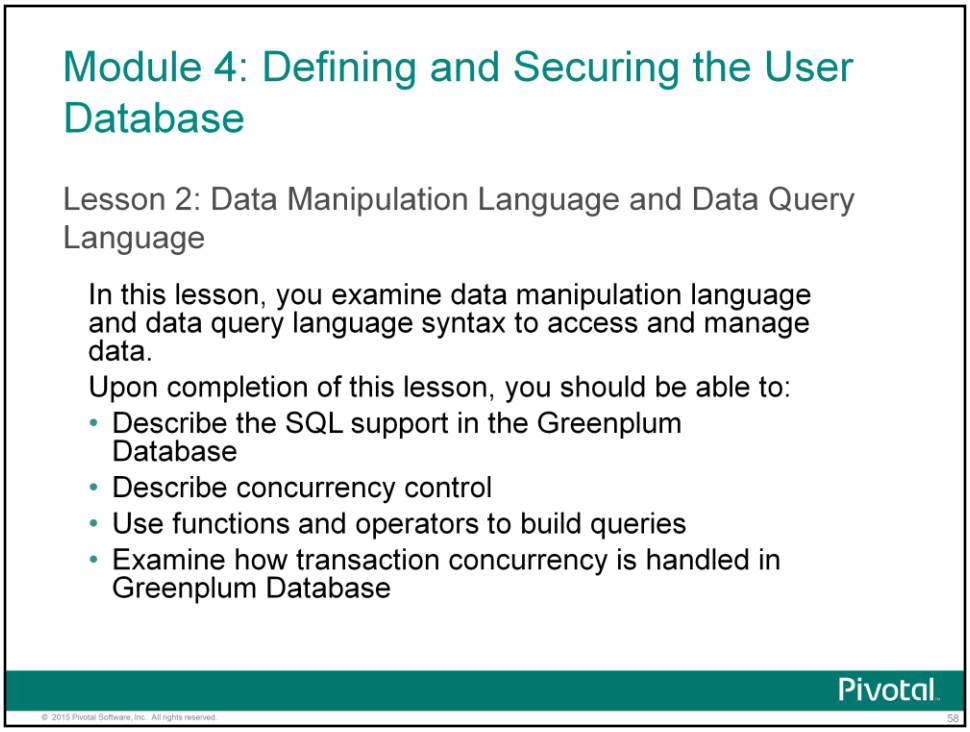
Once the tablespace has been created, data objects can be created in the table space.



In this lab, you create and manage database objects using the Data Definition Language. This includes creating databases, schemas, tables, views, indexes, and schemas.



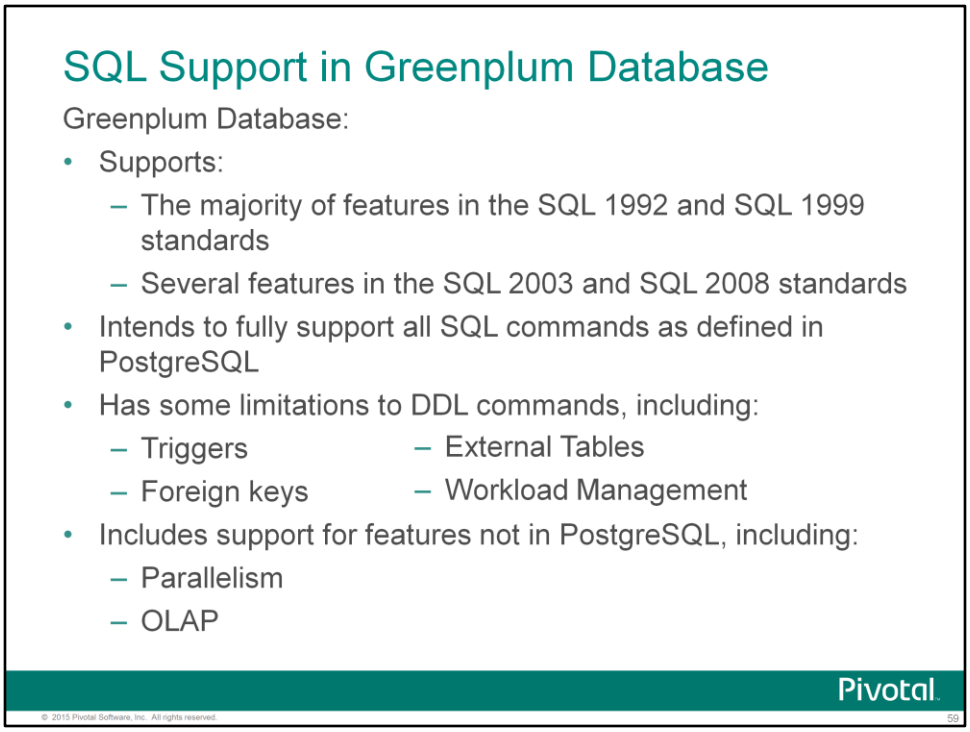
This lesson covered an overview of databases, schemas, tables, and how to create and manage these objects. A description of constraints and the various types supported by Greenplum Database, as well as a list and description of various data types supported within the environment. Additionally, the lesson covered descriptions and syntax for creating and managing views, indexes, sequences, triggers, and tablespaces



Data manipulation language consists of SQL syntax that is used to manage and change data. This includes inserting, deleting, and updating data in tables. Data query language is the art of extracting and retrieving data.

Upon completion of this lesson, you should be able to:

- Describe the SQL support available in the Greenplum Database
- Describe concurrency control
- Insert, update, and delete data using SQL syntax
- Use functions and operators to build queries
- Examine how transaction concurrency is handled in Greenplum Database using MVCC



Both data manipulation language (DML) and data query language (DQL) are supported as in PostgreSQL with some minor limitations. Greenplum supports:

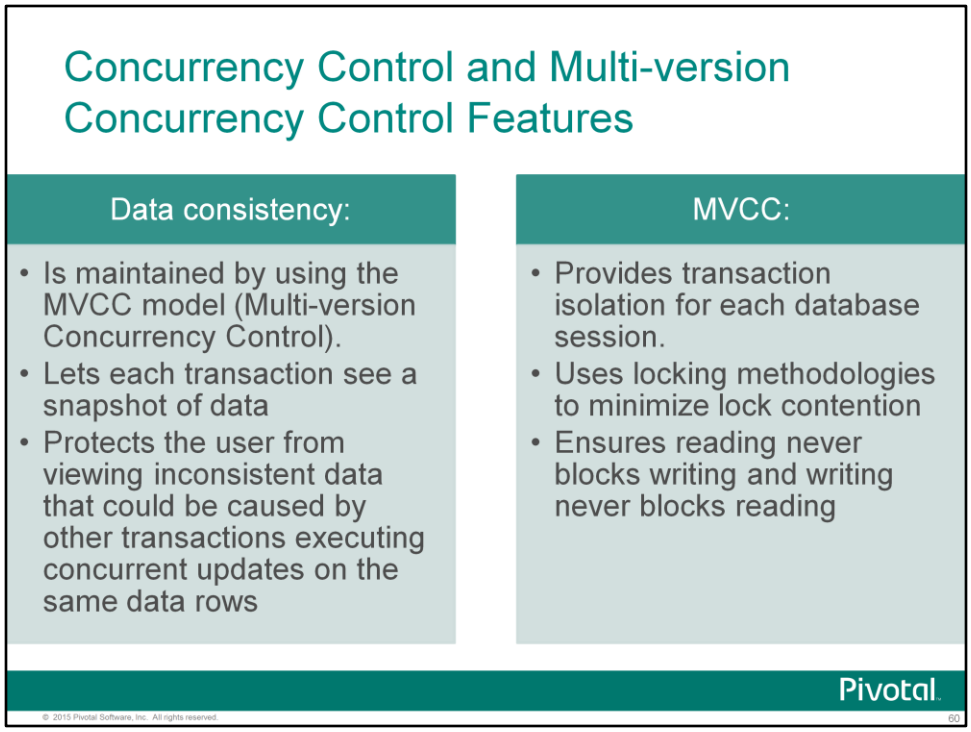
- The majority of features of the SQL 1992 and SQL 1999 standards
- Several of the features of the SQL 2003 and SQL 2008 standards

All other SQL commands are considered Data Definition Language (DDL) or utility commands. Most DDL is supported as in PostgreSQL, but there are a few limitations regarding things like foreign keys and triggers.

Greenplum has also added SQL to support features not in regular PostgreSQL, including:

- Parallelism
- OLAP SQL extensions to `SELECT`
- Managing external tables
- Managing workload management with resource queues

For details on the supported features in the SQL standards, refer to the *Greenplum Database Administrator Guide*.



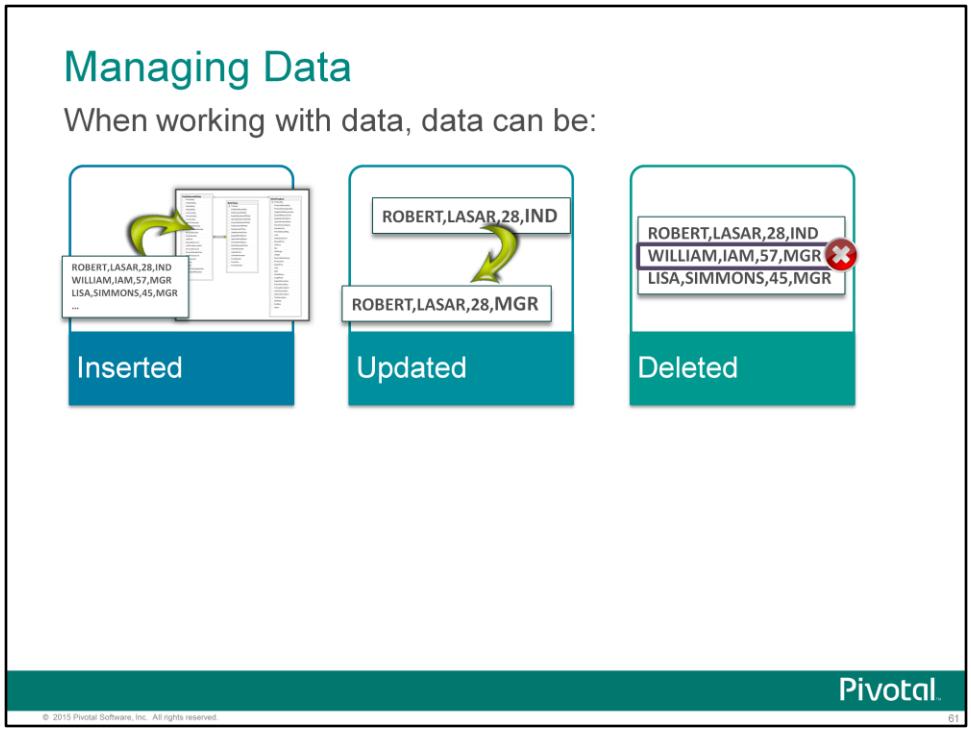
When you start a transaction, you snapshot the data at the time in point you are looking at it. The advantage of this is that all of the data is captured at that time. If a query takes 3 hours to complete, instead of working on the data that exists after the query completes, it works on the data at the time of the snapshot. This eliminates inconsistencies that can occur when data is changed during that time period, such as when:

- Inserting new rows
- Updating existing rows
- Deleting rows

This provides transaction isolation for each database session. By eschewing explicit locking methodologies of traditional database systems, multi-version concurrency control, or MVCC:

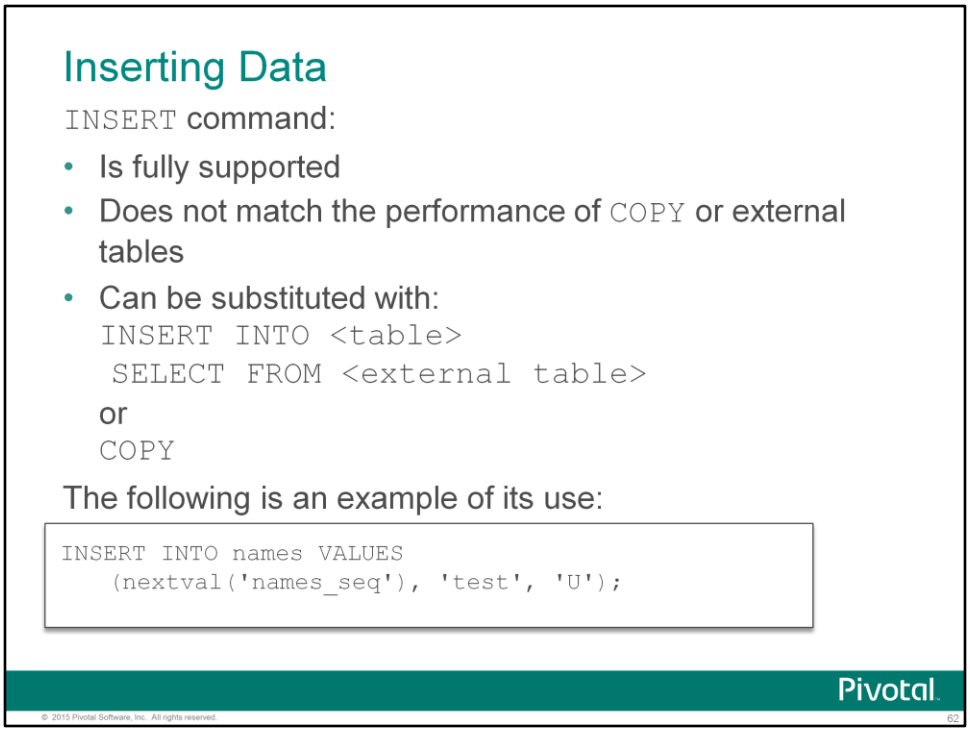
- Minimizes lock contention in order to allow for reasonable performance in multiuser environments.
- Ensures locks acquired for querying, or reading, data do not conflict with locks acquired for writing data. Reading never blocks writing and writing never blocks reading.

Greenplum Database provides various lock modes to control concurrent access to data in tables. Most Greenplum Database SQL commands automatically acquire locks of appropriate modes to ensure that referenced tables are not dropped or modified in incompatible ways while the command executes. For applications that cannot adapt easily to MVCC behavior, the `LOCK` command can be used to acquire explicit locks. However, proper use of MVCC will generally provide better performance than locks.

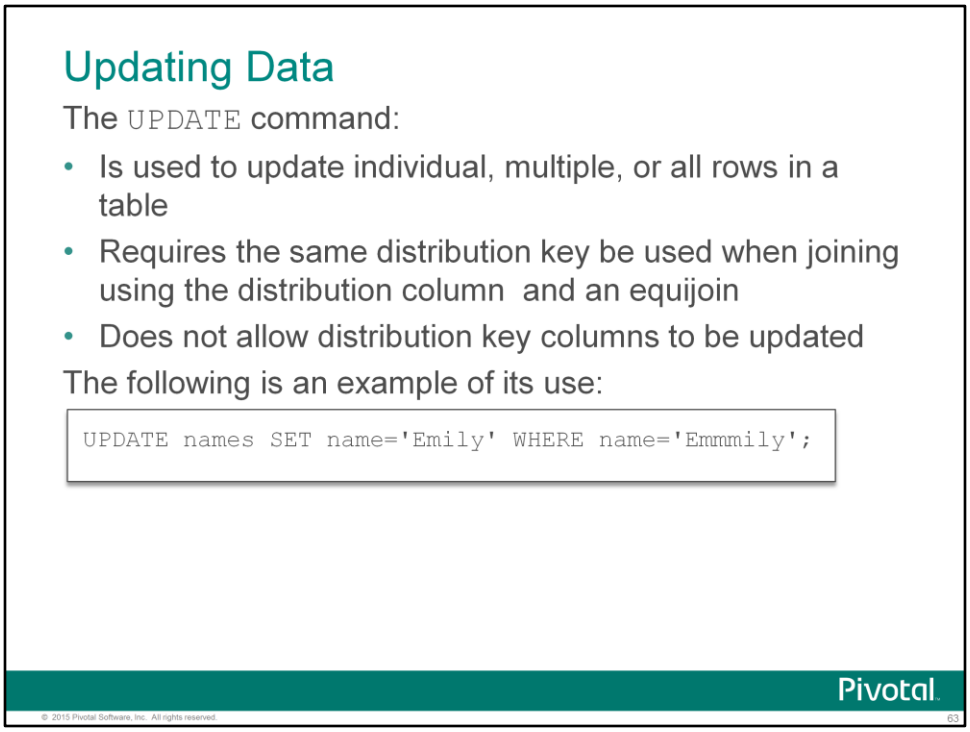


Data is managed using the following concepts:

- Insert – Data is inserted into tables
- Update – Existing data is updated or modified
- Delete – Existing data is removed from a table



While the `INSERT` command contains the same level of support as in PostgreSQL, the performance is not as good in the distributed environment. If you have a lot of rows to insert, consider using `COPY` or external tables for faster load performance.

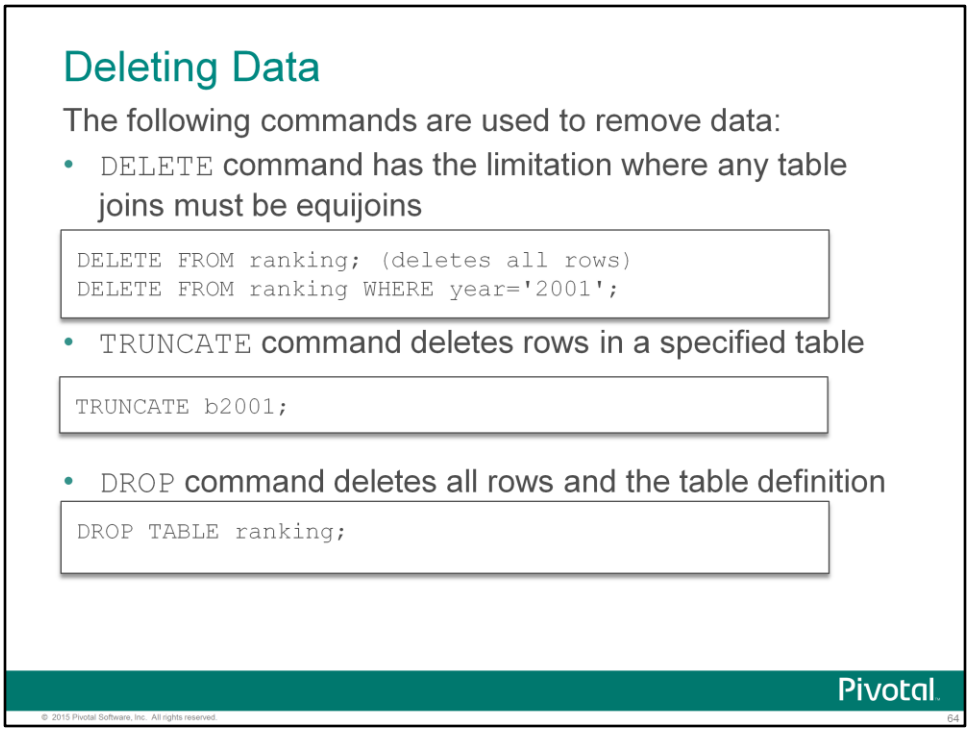


The `UPDATE` command can be used to update rows in Greenplum DB.

Updating data refers to the modification of data that already exists in the database. You can update individual, multiple, or all rows in a table.
Greenplum has the following restrictions or limitations:

- When joining two tables using the `UPDATE` command, the tables must have the same Greenplum distribution key and be joined on the distribution key column(s). This is referred to as an equijoin.
- Greenplum distribution key columns may not be updated.

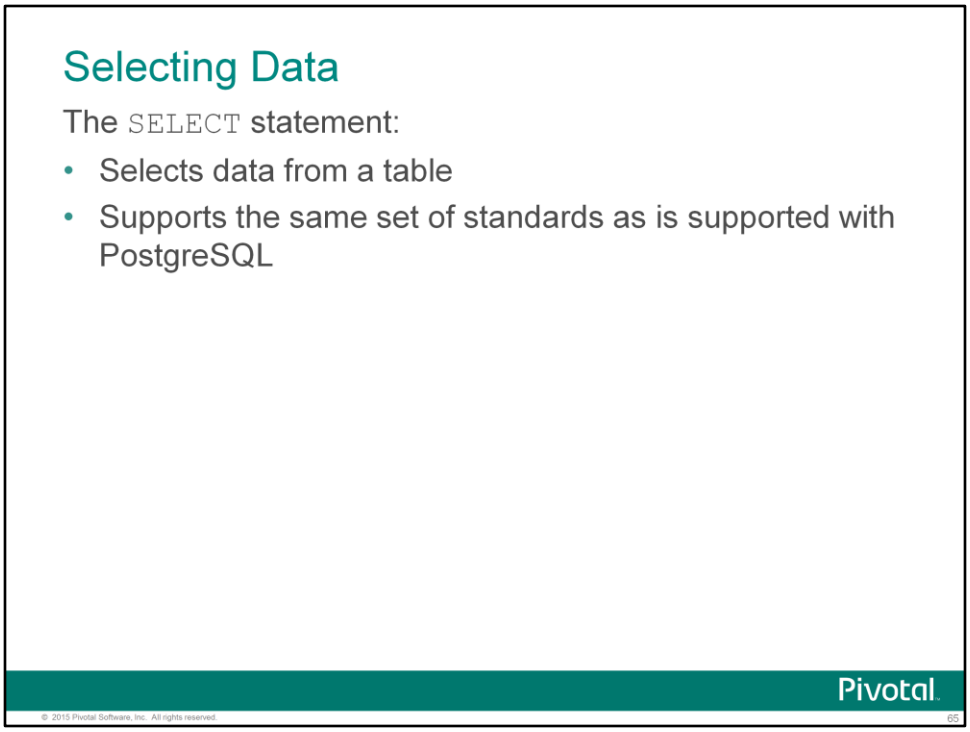
With the MVCC transaction model, every update is essentially an upsert, where the updated row's visibility information is marked stale and a newly visible row is inserted.



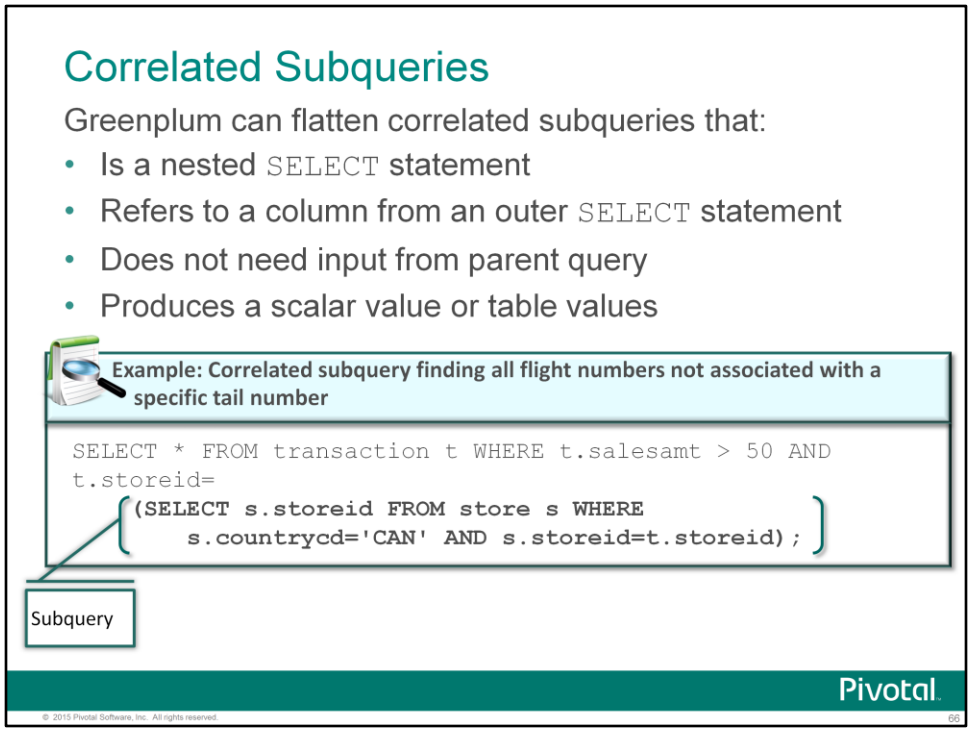
The following commands can be used to delete data in Greenplum DB:

- **DELETE** – Greenplum has a limitation when joining two tables in a `DELETE` statement. The tables to be joined must have the same distribution key columns if the join is being performed on the distribution column and be joined using those columns (equijoins).
- **TRUNCATE** – This is like an unqualified delete, but it is faster since it does not scan the tables as the `DELETE` command does. As `TRUNCATE` does not scan tables, it does not truncate child tables, working only on the table specified.
- **DROP** – This command drops the rows and the table definition specified.

For parallel deletes, the `WHERE` clause must be on the distribution columns. The same for is true for the `UPDATE` command.



The support for `SELECT` statements in the Greenplum Database is the same as PostgreSQL. This now includes correlated subquery, which enjoyed less support in Greenplum Database in earlier revisions of the database.



A correlated subquery is a nested SELECT statement that refers to a column from an outer SELECT statement. A subquery can be scalar, in that it returns a single row with a single value, or a table subquery, where multiple rows can be returned.

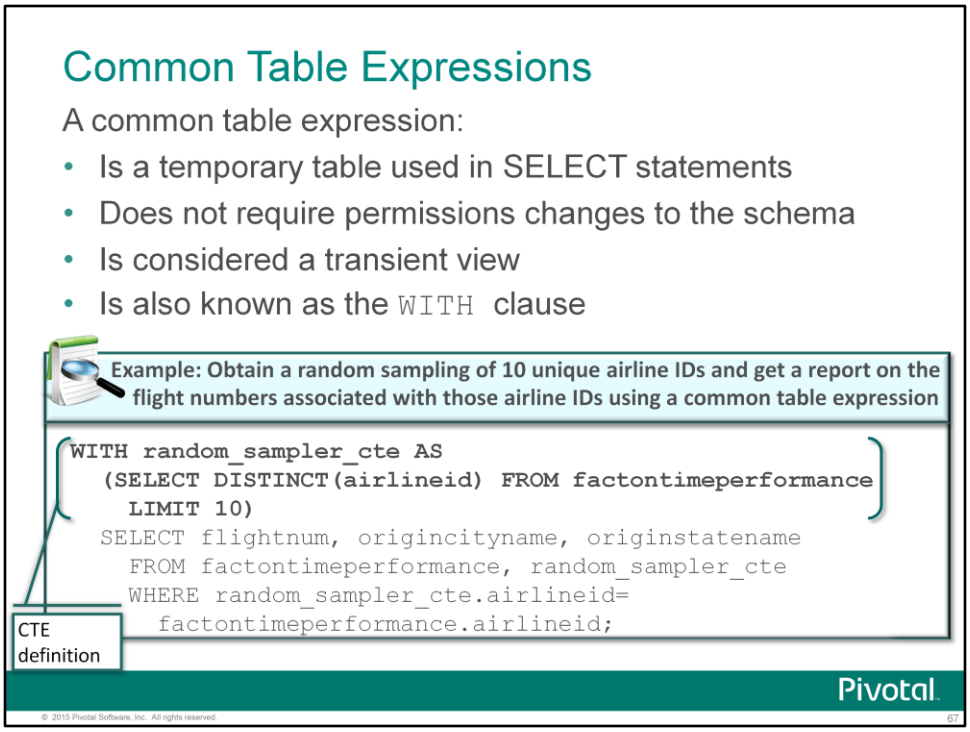
Greenplum Database supports correlated subqueries by unnesting the subquery where possible, or transforming the subquery into a join with the tables of the outer query.

If a correlated subquery has characteristics that do not prevent flattening, then it will be transformed to a join and be executed as expected.

It is important to understand the potential performance impacts on a query. A correlated query may require a rewrite in order to improve the overall performance of the query.

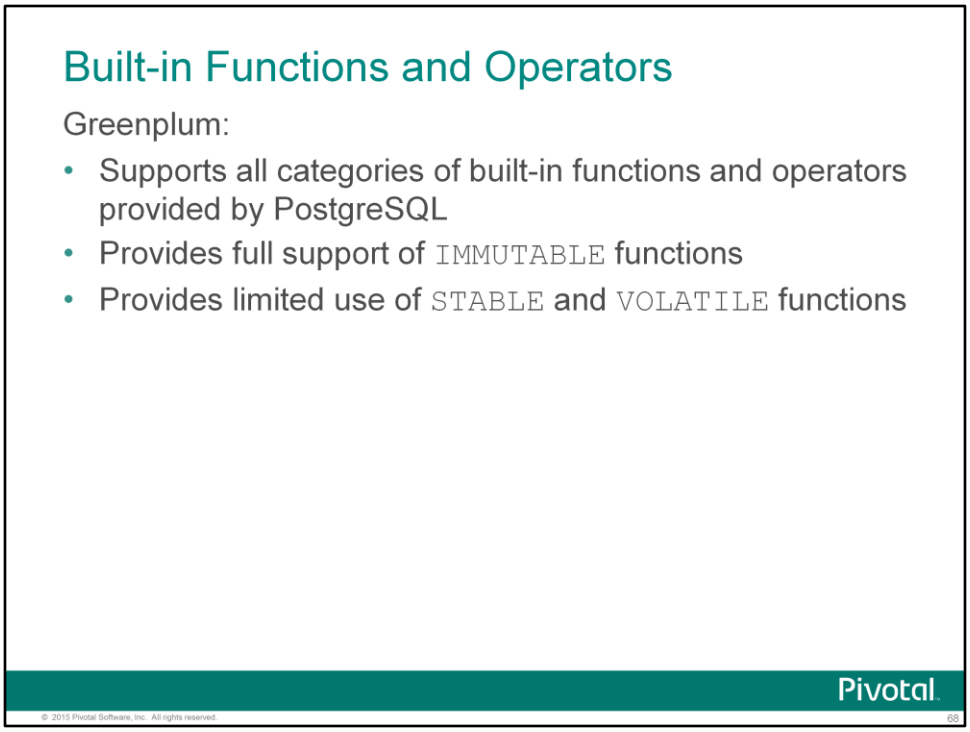
In this specific case, the cost of the correlated subquery is greater than a rewrite. A rewrite on the query displayed on the screen is:

```
SELECT t.* FROM transaction t, store s
WHERE t.salesamt > 50
   AND s.countrycd='CAN'
   AND s.storeid=t.storeid;
```



The common table expression, introduced in SQL standard 2003, is a temporary table set used in subsequent SELECT statements. It is often preferred over views, in that views require permissions to the schema, necessitating privilege changes. Common table expressions however are transient views, derived from a query. It is often called the *`WITH` clause*.

CTEs address queries that are generated by business intelligence tools. These are not used as often in adhoc queries because of their potential complexities.

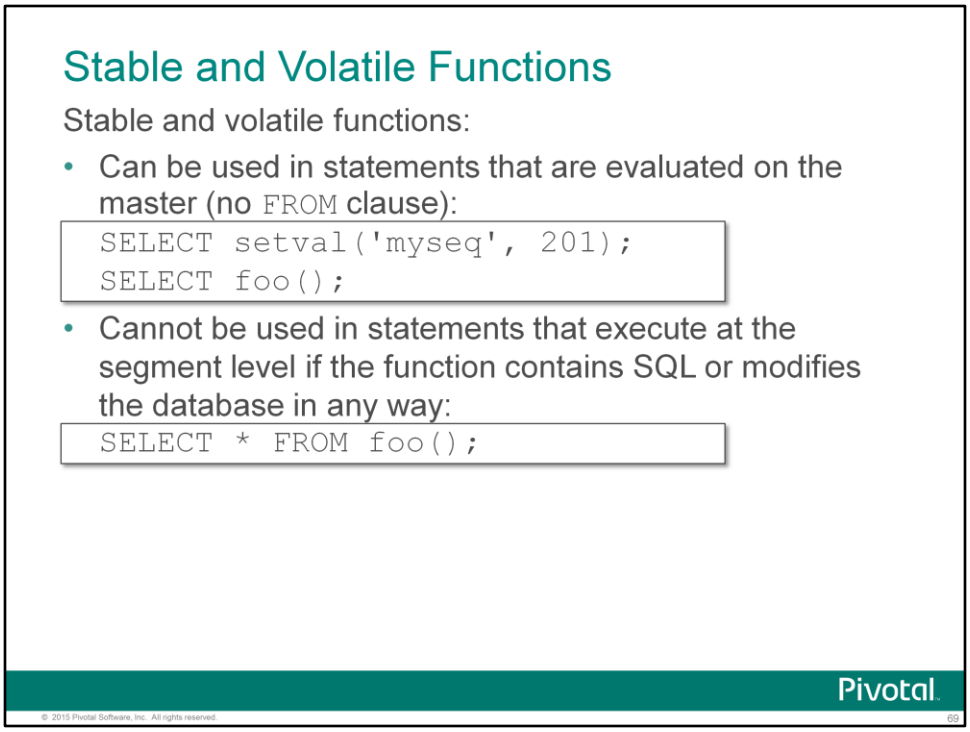


PostgreSQL provides a large number of functions and operators for the built-in data types. Greenplum fully supports all built-in functions that are defined as immutable. An immutable function always returns the same result when given the same argument values; that is, it does not do database lookups or otherwise use information not directly present in its argument list.

For functions that are stable or volatile, they can be used in a limited capacity in Greenplum Database.

The function types are defined as follows:

- **`STABLE`** indicates that within a single table scan the function will consistently return the same result for the same argument values, but that its result could change across SQL statements. Functions whose results depend on database lookups or parameter variables are classified as `STABLE`. Also note that the `current_timestamp` family of functions qualify as stable, since their values do not change within a transaction.
- **`VOLATILE`** indicates that the function value can change even within a single table scan. Relatively few database functions are volatile in this sense; some examples are `random()`, `currval()`, `timeofday()`. But note that any function that has side-effects must be classified volatile, even if its result is quite predictable (for example, `setval()`).



In Greenplum Database, the data is divided up amongst the segments. Each segment is, in a sense, its own distinct PostgreSQL database. To prevent data from becoming out-of-sync across the segments, any function classified as `STABLE` or `VOLATILE` cannot be executed at the segment level if it contains SQL or modifies the database in any way. For example, functions such as `random()` or `timeofday()` are not allowed to execute on distributed data in Greenplum Database because they could potentially cause inconsistent data between the segment instances. Anything in a `FROM` or `WHERE` clause is evaluated at the segment level. The following statement executes on the segments (has a `FROM` clause), however it may be allowed provided that the function used in the `FROM` clause simply returns a set of rows.

To ensure data consistency, `VOLATILE` and `STABLE` functions can safely be used in statements that are evaluated on and execute from the master. For example, statements without a FROM clause are always evaluated at and dispatched from the master.

Built-in Functions (SELECT)

Function	Description	Example
CURRENT_DATE	Returns the current system date	2006-11-06
CURRENT_TIME	Returns the current system time	16:50:54
CURRENT_TIMESTAMP	Returns the current system date and time	2008-01-06 16:51:44.430000+00:00
LOCALTIME	Returns the current system time with time zone adjustment	19:50:54
LOCALTIMESTAMP	Returns the current system date and time with time zone adjustment	2008-01-06 19:51:44.430000+00:00
CURRENT_ROLE ROLE	Returns the current database user	jdoe

Pivotal

© 2015 Pivotal Software, Inc. All rights reserved.

The functions shown are built-in functions that can be used as part of a SELECT statement.

Mathematical Functions

Function	Returns	Description	Example	Results
+ - * /	same	Add, Subtract, Multiply & Divide	1 + 1	2
%	Integer	Modulo	10%2	0
^	Same	Exponentiation	2^2	4
| /	Numeric	Square Root	|/9	3
| | /	Numeric	Cube Root	||/8	2
!	Numeric	Factorial	!3	6
& | # ~	Numeric	Bitwise And, Or, XOR, Not	91 & 15	11
<< >>	Numeric	Bitwise Shift left, right	1 << 4 8 >> 2	16 2

Pivotal

© 2015 Pivotal Software, Inc. All rights reserved.

Commonly used mathematical functions shown on this and the next slide are supported in Greenplum Database and PostgreSQL. These tables are provided as reference only. For a detailed listing of the functions supported, refer to the *Greenplum Database Administrator Guide*.

Mathematical Functions (Cont)

Function	Returns	Description	Example	Results
`abs`	same	Absolute Value	abs(-998.2)	998.2
`ceiling (numeric)`	Numeric	Returns smallest integer not less than argument	ceiling(48.2)	49
`floor (numeric)`	Numeric	Returns largest integer not greater than argument	floor(48.2)	48
`pi()`	Numeric	The π constant	pi()	3.1419…
`random()`	Numeric	Random value between 0.0 and 1.0	random()	.87663
`round()`	Numeric	Round to nearest integer	round(22.7)	23

Pivotal

© 2015 Pivotal Software, Inc. All rights reserved.

Mathematical Functions (Continued)

String Functions

Function	Returns	Description	Example	Results
string || string	Text	String concatenation	'my' || 'my'	'mymy'
char_length (string)	Integer	number of chars in string	char_length('mym y')	4
position (string in string)	Integer	Location of specified sub-string	position('my' in 'ohmy')	3
lower(string)	Text	Converts to lower case	lower('MYMY')	'mymy'
upper(string)	Text	Converts to upper case	upper('mymy')	'MYMY'
substring (string from n for n)	Text	Displays portion of string	substring('myohm y' from 3 for 2)	'oh'
trim(both,leading, trailing from string)	Text	Remove leading and/or trailing characters	trim(' mymy ')	'mymy'

Pivotal

© 2015 Pivotal Software, Inc. All rights reserved.

73

These commonly used string functions are supported in Greenplum Database and PostgreSQL. For a complete listing of the string functions available, refer to the *Greenplum Database Administrator Guide*.

String Functions (Cont)

Function	Returns	Description	Example	Results
initcap (string)	Text	Changes case	initcap ('my my')	'My My'
length (string)	Integer	Returns string length	length ('mymy')	4
split_part (string, delimiter, occurrence)	Text	Separates delimited list	split_part('one|two|three','|',2)	'two'

Pivotal

© 2015 Pivotal Software, Inc. All rights reserved.

String Functions (Continued)

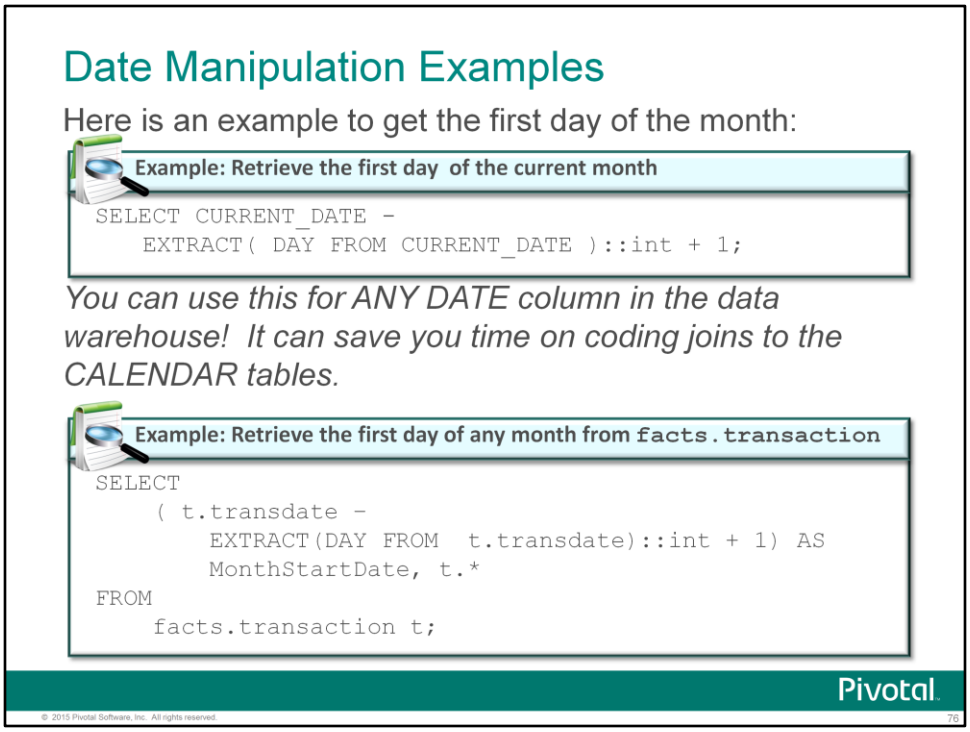
Date Functions

Function	Returns	Description	Example	Results
age (timestamp,timestamp)	Timestamp	Difference in years, months and days	age('2008-08-12' timestamp, current_timestamp)	0 years 1 month 11 days
extract (field from timestamp)	Integer	Returns year, month, day, hour, minute or second	extract(day from current_date)	11
now()	Timestamp	Returns current date & time	now()	2008-09-22 11:00:01
overlaps	Boolean	Simplifies comparing date ranges	WHERE ('2008-01-01','2008-02-11') overlaps ('2008-02-01','2008-09-11')	TRUE

Pivotal

© 2015 Pivotal Software, Inc. All rights reserved.

Commonly used date functions are displayed on the slide. For detailed information and a full listing of all date functions, refer to the *Greenplum Database Administrator Guide.*



In the examples shown, you can retrieve the first day of the month using the examples shown on the slide. The `CURRENT_DATE` and `EXTRACT` functions are used to derive that data in the first example.

The second example allows you to retrieve the first day of any month provided in the data.

Statistical or Aggregate Functions

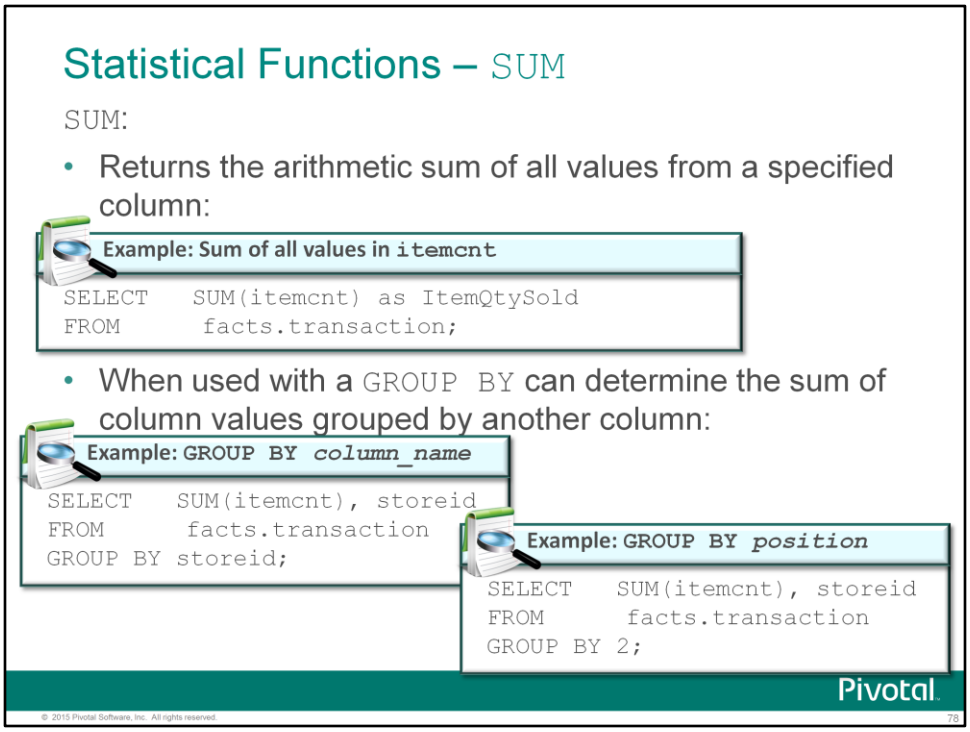
Function	Returns	Description
sum	bigint for smallint or int arguments, numeric for bigint arguments, double precision for floating-point arguments, otherwise the same as the argument data type	Sum of *expression* across all input values
count	bigint	Number of input rows for which the value of *expression* is not null
avg	numeric for any integer type argument, double precision for a floating-point argument, otherwise the same as the argument data type	the average (arithmetic mean) of all input values
min	same as argument type	Minimum value of *expression* across all input values
max	same as argument type	Maximum value of *expression* across all input values

Pivotal

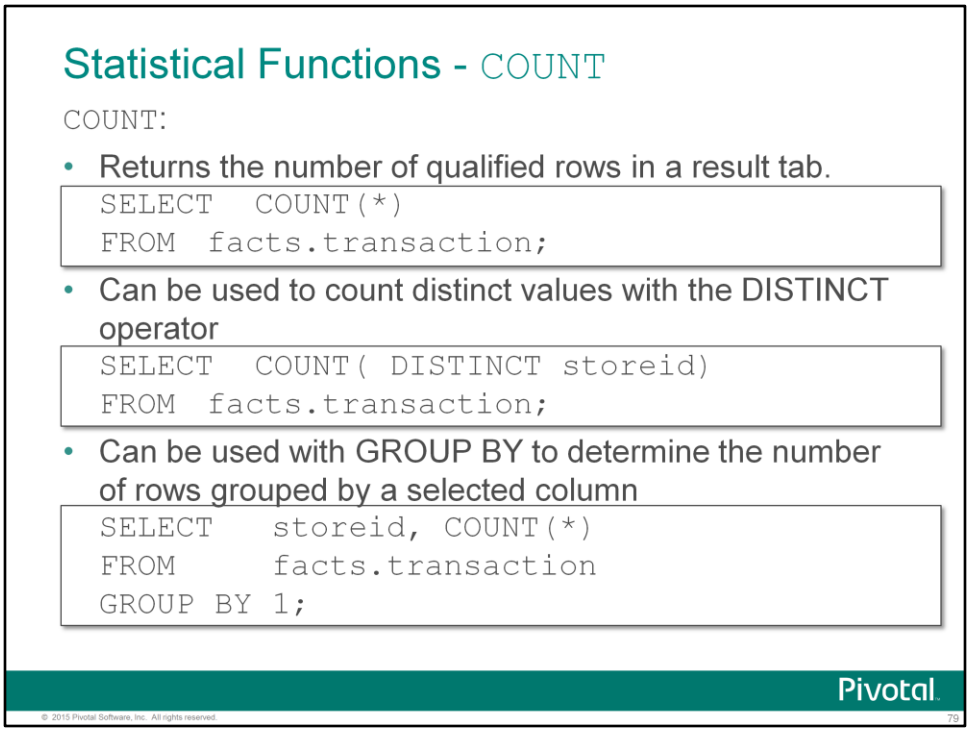
© 2015 Pivotal Software, Inc. All rights reserved.

An aggregate function is a function that acts on values within a column. The aggregate function computes multiple input values to a single output value. For example, calculating the total of values in a column can be performed with the `SUM` aggregate function.

The table shows some of the more commonly used aggregates: `SUM`, `COUNT`, `AVG`, `MIN`, and `MAX`. Aggregates are not allowed in the `WHERE` clause, but are allowed in the `HAVING` clause.

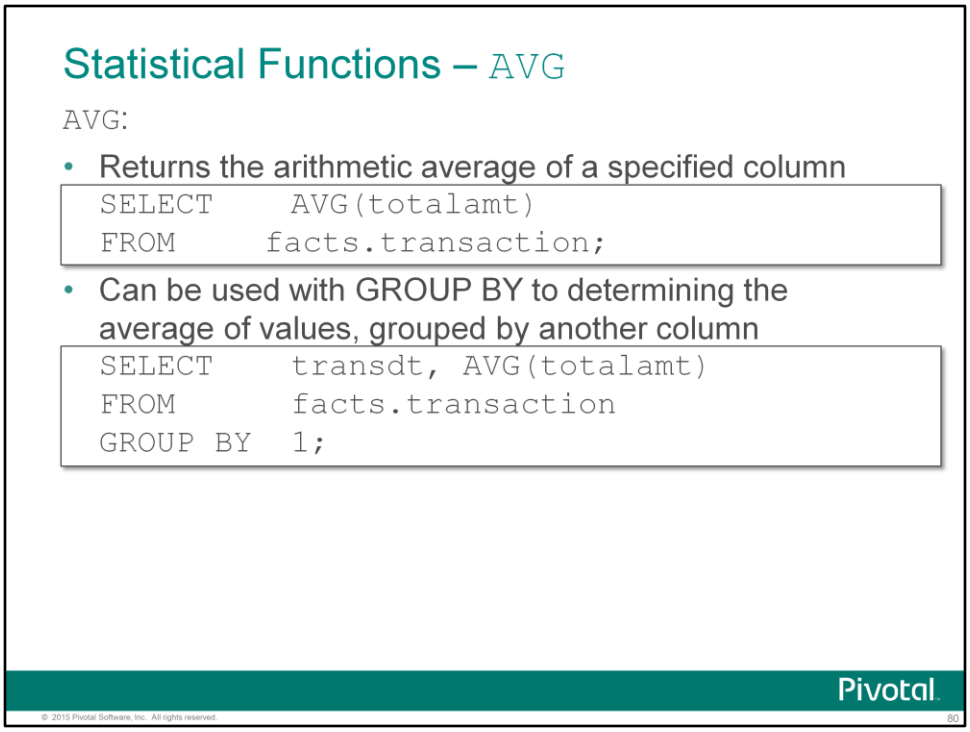


The SUM function returns the arithmetic function of all values from a specified column. It can also be used with the GROUP BY expression to determine the sum of column values grouped by another column.



The `COUNT` function returns the number of qualified rows in a result tab. You can obtain the number of unique values in a column by adding the `DISTINCT` operator to the `COUNT` function. The example shown returns a count of the number of unique `storeids` in the transaction table.

The `COUNT` function used with a `GROUP BY` can be useful for determining the number of rows grouped by a selected column, as shown by the third example on the slide.



`AVG` returns the arithmetic average of a specified column in a result table. The query in the first example on the slide shows the average of the total sales amounts from the transaction table.

The `AVG` function, used with `GROUP BY` can be useful for determining the average amounts of values, grouped by another column. The second example on the slide obtains the daily average of the `totalamt` column from the transaction table.

Statistical Functions – MIN and MAX

MIN returns the minimum value of a given column

```
SELECT   MIN(transdt)
FROM     facts.transaction;
```

MAX returns the maximum value of a given column

```
SELECT   MAX(transdt)
FROM     facts.transaction;
```

Note: Adding additional dimensions to the select statement requires a GROUP BY for each dimension. This returns the MIN or MAX (or both), aggregated over all dimensions.

Pivotal

© 2015 Pivotal Software, Inc. All rights reserved.

MIN returns the minimum value of a given column. The query in the first example shows the earliest transaction date value from the transaction table.

MAX returns the maximum value of a given column. The query in the second example on the slide returns the maximum transaction date value from the transaction table.

Note that adding additional dimensions to the select statement requires a GROUP BY for each dimension. This returns the MIN or MAX (or both), aggregated over all dimensions.

Logical Operators

x	y	x AND y	x OR y
TRUE	TRUE	TRUE	TRUE
TRUE	FALSE	FALSE	TRUE
FALSE	FALSE	FALSE	FALSE
TRUE	NULL	NULL	TRUE
FALSE	NULL	NULL	FALSE
NULL	NULL	NULL	NULL

x	NOT x
TRUE	FALSE
FALSE	TRUE
NULL	NULL

Pivotal

© 2015 Pivotal Software, Inc. All rights reserved.

Logical operators `AND` and `OR` are displayed on the slide. These return a boolean (TRUE or FALSE) based on the operation performed.

POSIX Pattern Matching

Operator	Description	Example
~	Matches regular expression Case sensitive	'thomas' ~ '.*thomas.*'
~*	Matches regular expression Case insensitive	'thomas' ~* '.*Thomas.*'
!~	Does not match regular expression Case sensitive	'thomas' !~ '.*Thomas.*'
!~*	Does not match regular expression Case insensitive	'thomas' !~* '.*vadim.*'

Pivotal

© 2015 Pivotal Software, Inc. All rights reserved.

Pattern matching and the use of regular expression can greatly simplify your string manipulation and comparison needs.

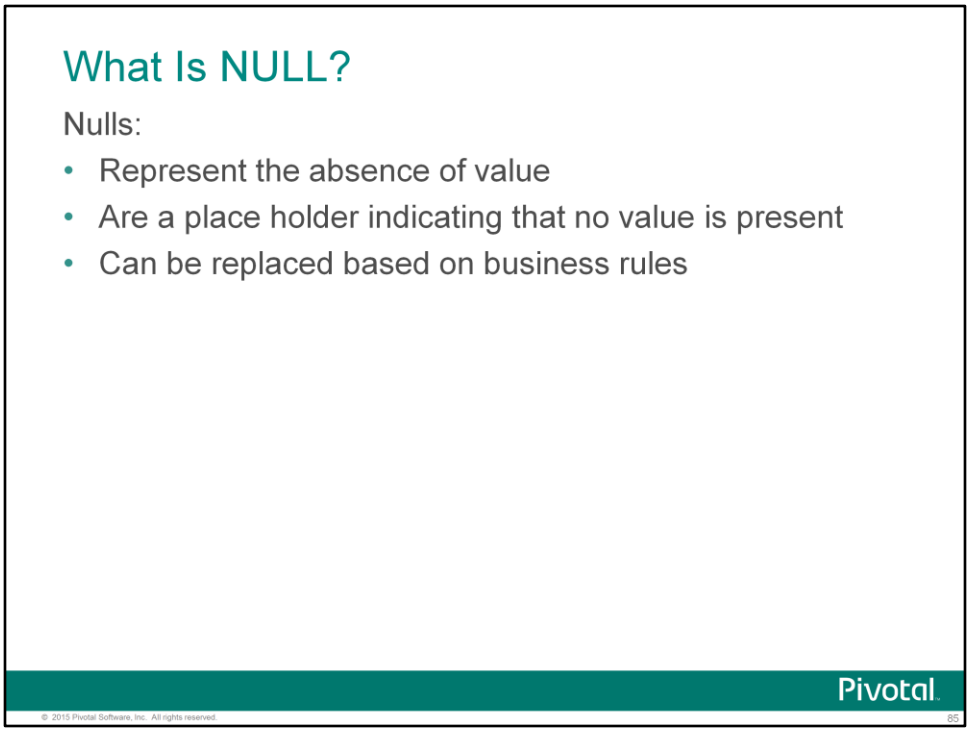
Comparison Operators

Operator	Description
=	Equal to
!= OR <>	NOT Equal to
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
x BETWEEN y AND z	Short hand for x >= y and x <= z
x IS NULL	True if x has NO VALUE
'abc' LIKE '%abcde%'	Pattern Matching
POSIX Comparisons	POSIX Pattern Matching

Pivotal

© 2015 Pivotal Software, Inc. All rights reserved.

Comparison operators are used to compare two objects to each other.



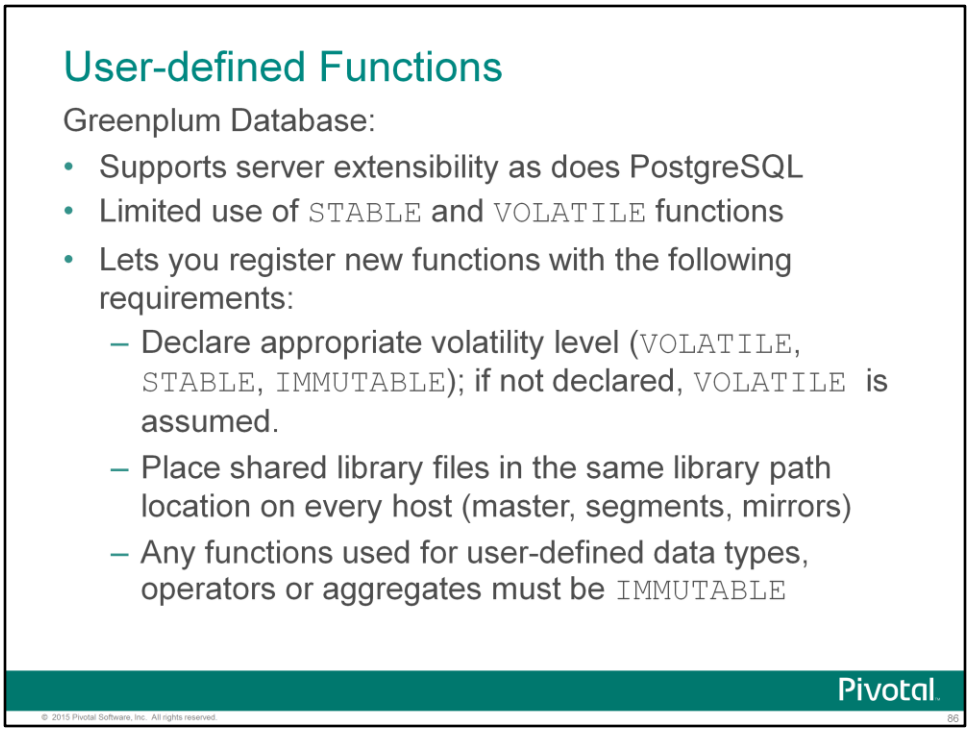
`NULL` is the value used to represent an unknown piece of data in a column or field. It represents the absence of a value.

Within your data files you can designate a string to represent null values.

The default string is:

- `\N` (backslash-N) in TEXT mode
- An empty value with no quotations in CSV mode

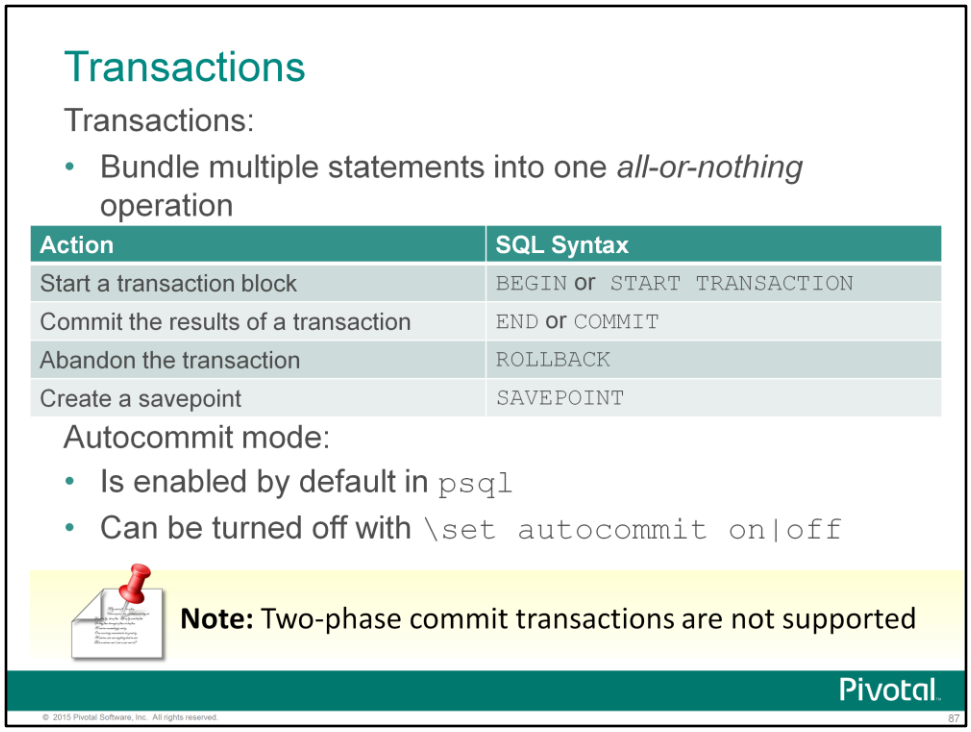
You can also declare a different string using the `NULL` clause of `COPY`, `CREATE EXTERNAL TABLE` or `gpload` when defining your data format. It acts as a placeholder indicating that no value is present. For example, you might prefer an empty string for cases where you do not want to distinguish nulls from empty strings. When using the Greenplum Database loading tools, any data item that matches the designated null string will be considered a null value.



Both Greenplum Database and PostgreSQL support user-defined functions. The process for extending the server functionality with new functions, and thereby data types, aggregates and operators, is the same as in PostgreSQL.

After you write a function, you must register it with the server using `CREATE FUNCTION`. When doing this make sure to choose the correct volatility level and keep in mind the limitations of `STABLE` and `VOLATILE` functions we discussed on the previous slide. The default level is `VOLATILE`. Any function used in a user-created data type, aggregate, or operator must be `IMMUTABLE` in Greenplum Database or else you will get errors when trying to use the function.

Note that in Greenplum Database, the shared library files for C user-created functions must reside in the same library path location on every host in the Greenplum Database array (masters, segments, and mirrors). Best to place these files relative to the `$GPHOME/lib` directory.



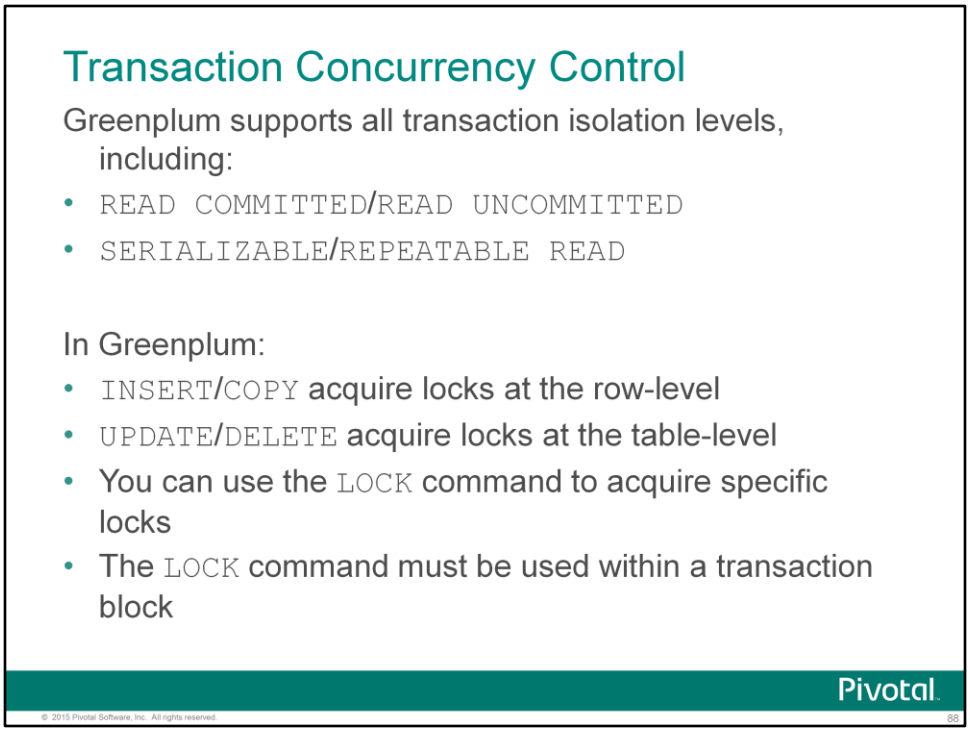
Transactions allow you to bundle together multiple SQL statements in one all-or-nothing operation. They also ensure that the changes are indeed made to permanent storage won't be lost if a crash occurs.

The SQL commands used to perform transactions are:

- **`BEGIN` or `START TRANSACTION`** – Start a transaction block
- **`END` or `COMMIT`** – Commit the results of the transaction
- **`ROLLBACK`** – Abandon the transaction without changes.
- **`SAVEPOINT`** – Savepoints allow you to selectively discard parts of the transaction while committing the rest. After defining a savepoint with the `SAVEPOINT` command, you can roll back to a savepoint with the `ROLLBACK TO` command.

By default, `psql` runs in `autocommit` mode. This means each command is implicitly wrapped in a `BEGIN` and `COMMIT` (or a ROLLBACK if there is an error). Each statement issued in psql is its own transaction. You may want to run several commands in a single transaction, so it may be desired to sometimes turn autocommit off. With autocommit off, you will have to explicitly use `BEGIN` to start a transaction and `COMMIT` or `ROLLBACK` to end it.

Postgres 8.1 introduced the concept of 2 phase commit transactions. After a transaction is prepared, the transaction is no longer associated with the current session; instead, its state is fully stored on disk, and there is a very high probability that it can be committed successfully, even if a database crash occurs before the commit is requested. This feature is currently not supported in Greenplum DB.



The SQL standard defines four transaction isolation levels:

- **`READ COMMITTED`** – This is the default behavior for transaction isolation. A statement can only see rows committed before it began executing.
- **`READ UNCOMMITTED`** – In Greenplum Database, READ UNCOMMITTED is the same as `READ COMMITTED`.
- **`SERIALIZABLE`** – All statements of the current transaction can only see rows committed before the first statement was executed in the transaction. `SERIALIZABLE` is the strictest transaction isolation. This level emulates serial transaction execution, as if transactions had been executed one after another, serially, rather than concurrently. Applications using this level must be prepared to retry transactions due to serialization failures.
- **`REPEATABLE READ`** – In Greenplum Database, `REPEATABLE READ` is the same as `SERIALIZABLE`.

Row-level locks are acquired with the `INSERT` and `COPY` commands, while table-level locks are acquired using the `UPDATE` and `DELETE` commands.

If the MVCC model does not provide the level of concurrency protection you need, you can acquire explicit locks on a table using the `LOCK` command. Once a lock is acquired, it is held until the end of the transaction. The LOCK command must be used within a `BEGIN/END` transaction block to hold the lock on the object you specify.

Lock Modes

Lock Mode	SQL Commands	Conflicts With
ACCESS SHARE	SELECT	ACCESS EXCLUSIVE
ROW SHARE	SELECT FOR UPDATE, SELECT FOR SHARE	EXCLUSIVE, ACCESS EXCLUSIVE
ROW EXCLUSIVE	INSERT, COPY	SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE, ACCESS EXCLUSIVE
SHARE UPDATE EXCLUSIVE	VACUUM (without FULL), ANALYZE	SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE, ACCESS EXCLUSIVE
SHARE	CREATE INDEX	ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE ROW EXCLUSIVE, EXCLUSIVE, ACCESS EXCLUSIVE

Pivotal

© 2015 Pivotal Software, Inc. All rights reserved.

The table shows the available lock modes and the contexts in which they are used automatically by Greenplum Database, and the lock modes they conflict with.

You can acquire any of these locks explicitly with the LOCK command.

Remember that all of these lock modes are table-level locks, even if the name contains the word *row*; the names of the lock modes are historical. To some extent the names reflect the typical usage of each lock mode — but the semantics are all the same. The main difference between one lock mode and another is the set of lock modes with which each conflicts.

Two transactions cannot hold locks of conflicting modes on the same table at the same time. However, a transaction never conflicts with itself. For example, it may acquire `ACCESS EXCLUSIVE` lock and later acquire `ACCESS SHARE` lock on the same table.

Non-conflicting lock modes may be held concurrently by many transactions.

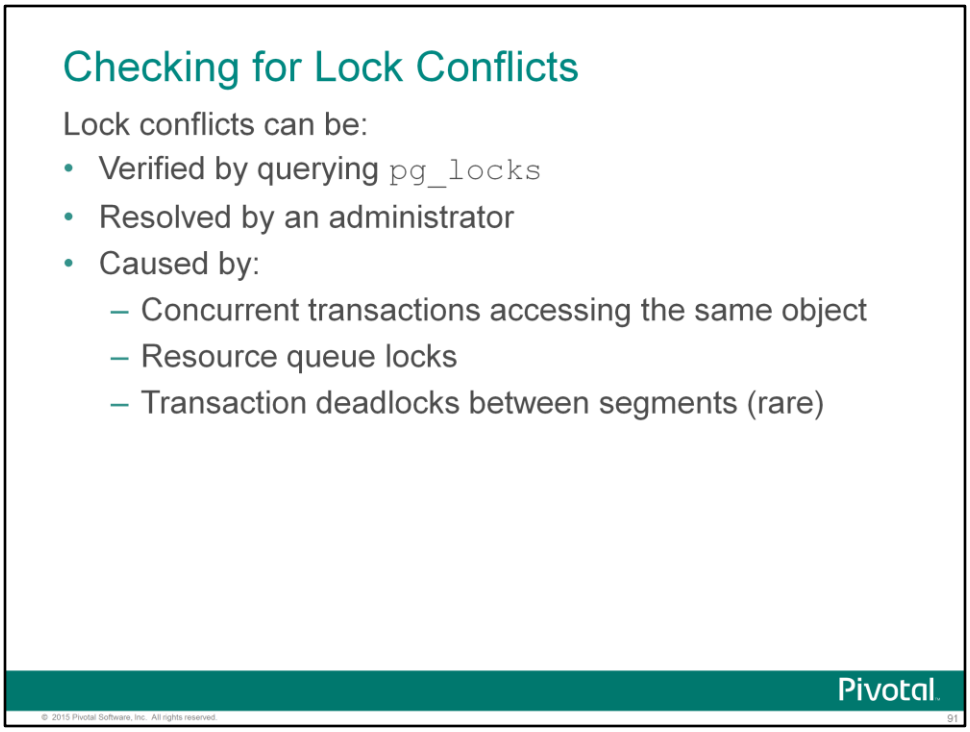
Lock Modes (Cont)

Lock Mode	SQL Commands	Conflicts With
SHARE ROW EXCLUSIVE		ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE, ACCESS EXCLUSIVE
EXCLUSIVE	DELETE, UPDATE	ROW SHARE, ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE, ACCESS EXCLUSIVE
ACCESS EXCLUSIVE	ALTER TABLE, DROP TABLE, TRUNCAGE, REINDEX, CLUSTER, VACUUM FULL	ACCESS SHARE, ROW SHARE, ROW EXCLUSIVE, SHARE UPDATE, EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE, ACCESS EXCLUSIVE

Pivotal

© 2015 Pivotal Software, Inc. All rights reserved.

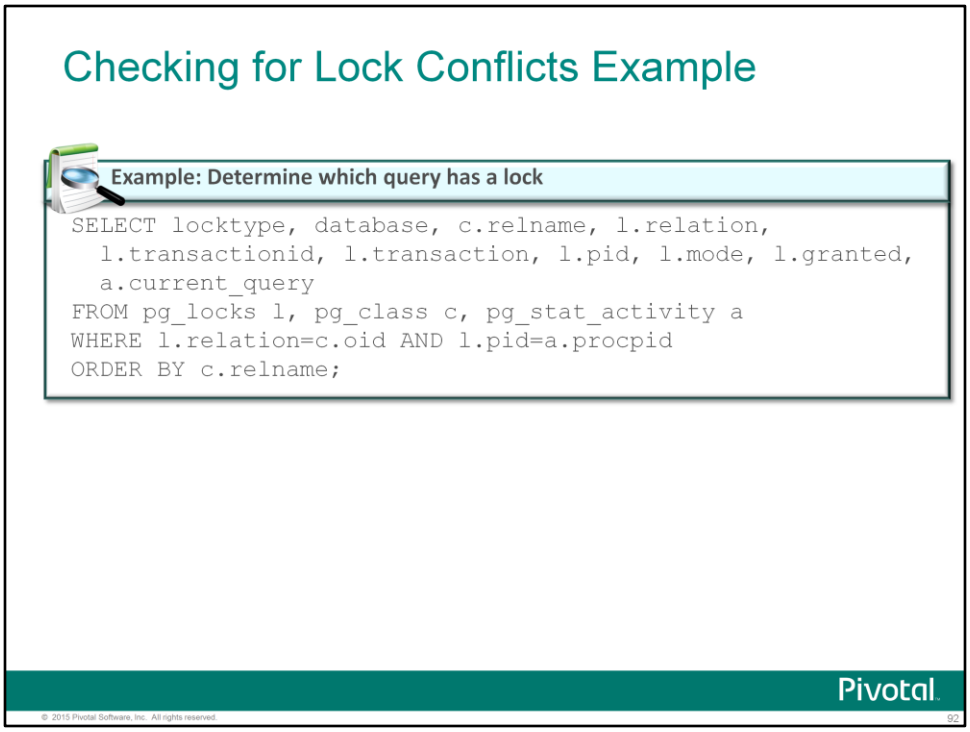
Lock Modes (Continued)



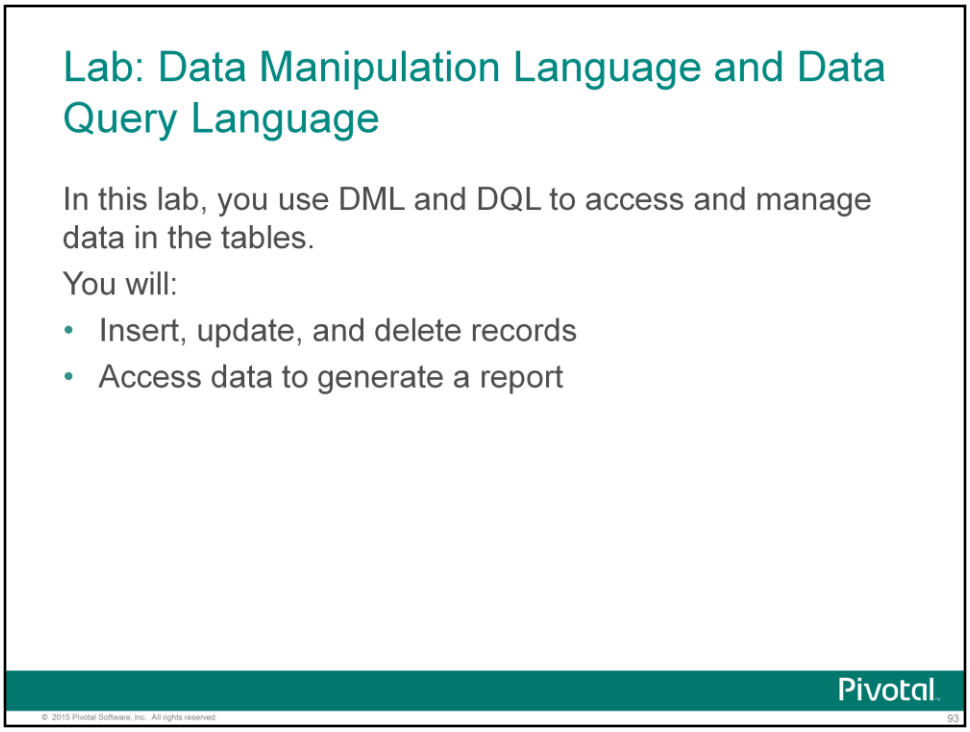
When an object is locked, subsequent concurrent queries that try to access the object need to wait for the lock to be freed. This can appear to the end user or application like the query is *hanging* or not being processed.

You should be aware of the following types of conflicts:

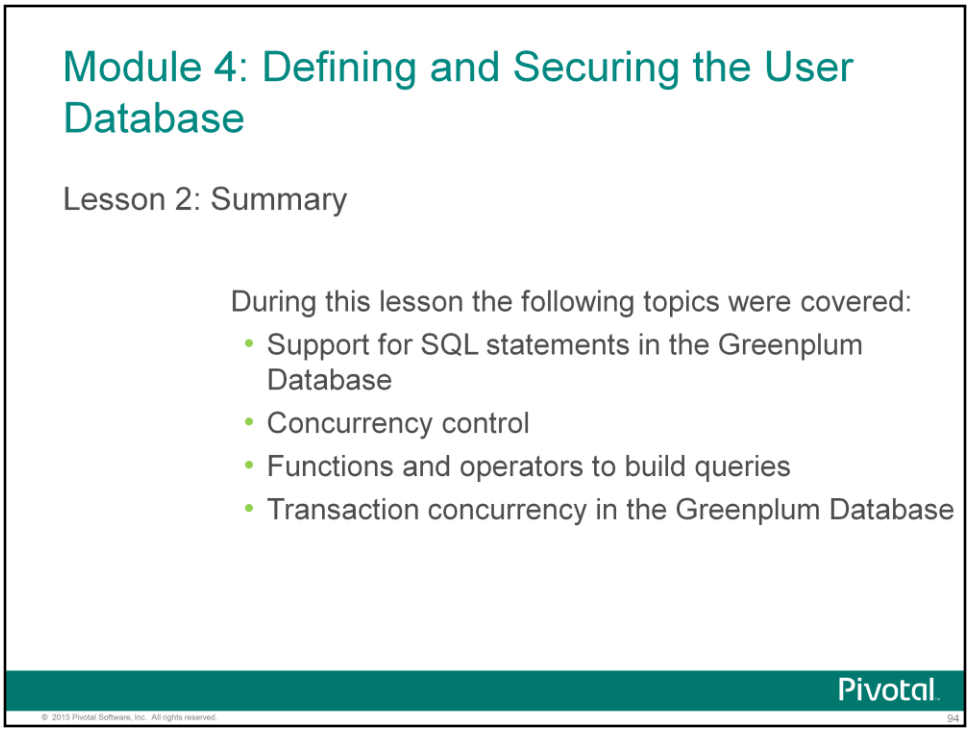
- **Lock Conflicts** – As long as a deadlock situation is not detected, a transaction seeking either a table-level or row-level lock will wait indefinitely for conflicting locks to be released. It is therefore a bad idea for applications to hold transactions open for long periods of time, such as while waiting for user input.
- **Queue Locks** – If a query is submitted through a resource queue that has exceeded it limits, the queue is locked until resources are freed. The `pg_locks` table will show a lock on the queue object for a particular query and session.
- **Deadlocks** – PostgreSQL has a built-in deadlock detector that catches most deadlock situations when the query is submitted. However, Greenplum does not have a global deadlock detector across all segments in the system. Deadlock detection is local at the segment level. There are rare cases where a transaction deadlock can occur because of a deadlock at the segment level.



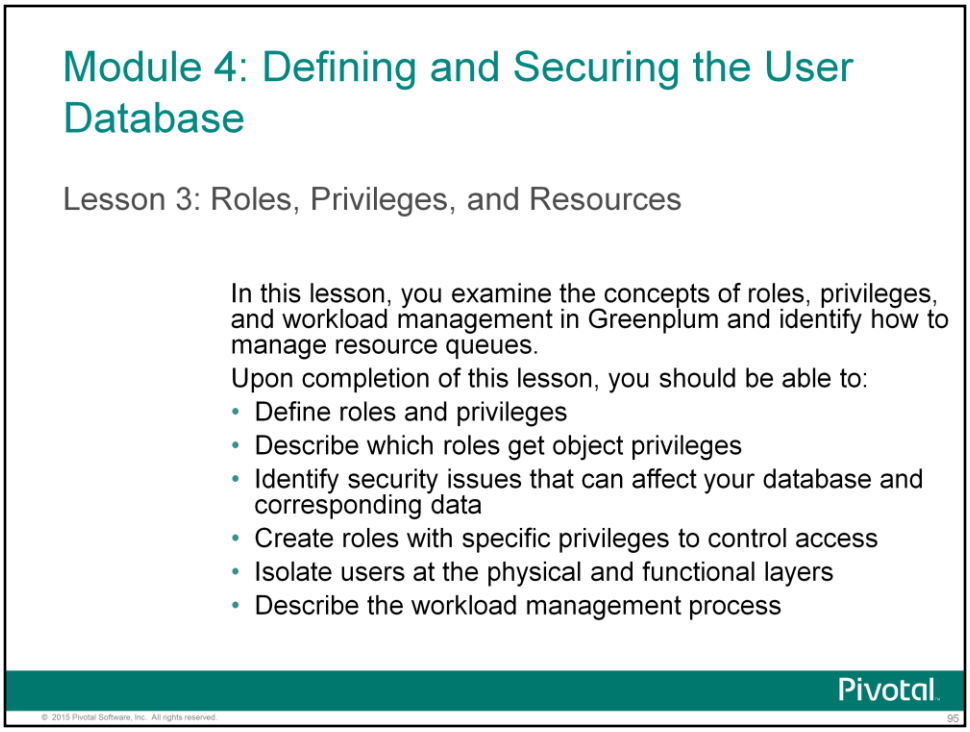
Checking the `pg_locks` view and joining it with `pg_class`, which lists the objects involved, and `pg_stat_activity`, which lists the users and sessions involved, can let an administrator know what is going on and identify lock conflicts. If a conflict is detected, killing the process id of a conflicting query on the master resolves the problem.



In this lab, you use DML and DQL to access and manage data in the tables.



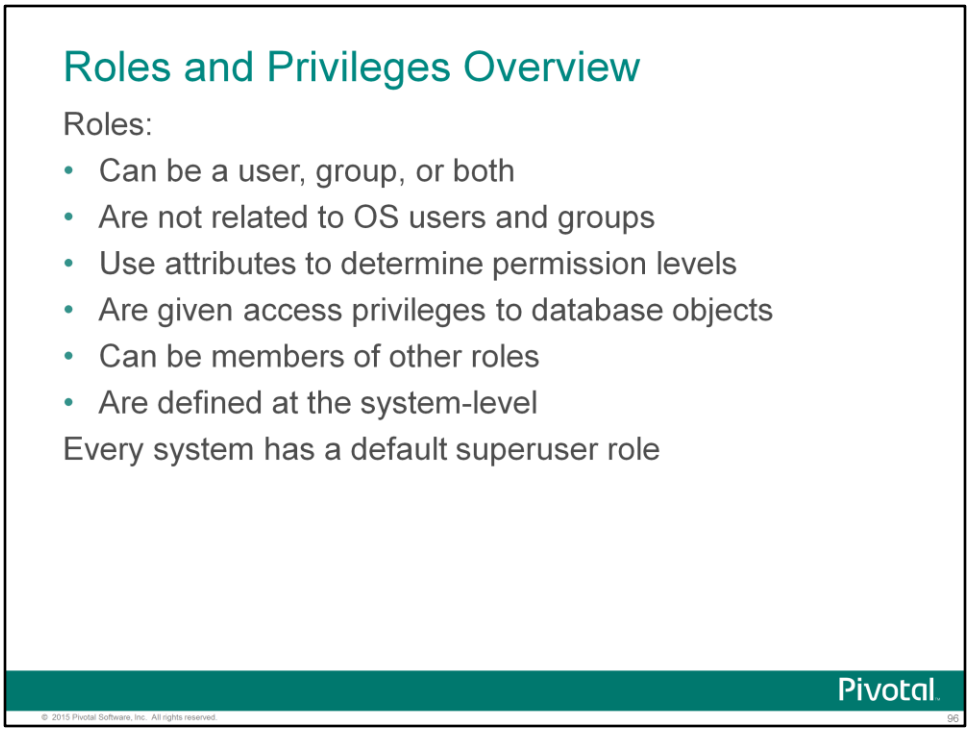
This lesson covered the DML and DDL syntax used to access and manipulate database objects, including how to insert, delete, and update data in tables, using functions, and various other statements for accessing database objects. Concurrency control allows both the system and users to lock database object to ensure that data retrieved is valid, despite changes made to the objects. Users create functions and use operators to repetitively access commonly used syntax.



To provide your users access to the database and to ensure each user's data is protected from unauthorized view, you must define roles for the database. You must also manage resources to ensure that users and their queries do not hog system resources.

In this lesson, you will:

- Define roles and privileges for the Greenplum Database system.
- Identify the privileges that can be assigned to specific roles.
- Identify security issues that can affect your database and corresponding data.
- Examine privileges that can be used to enforce access.
- Identify practices designed to isolate users.
- Examine the workload management processes and how to manage resources.

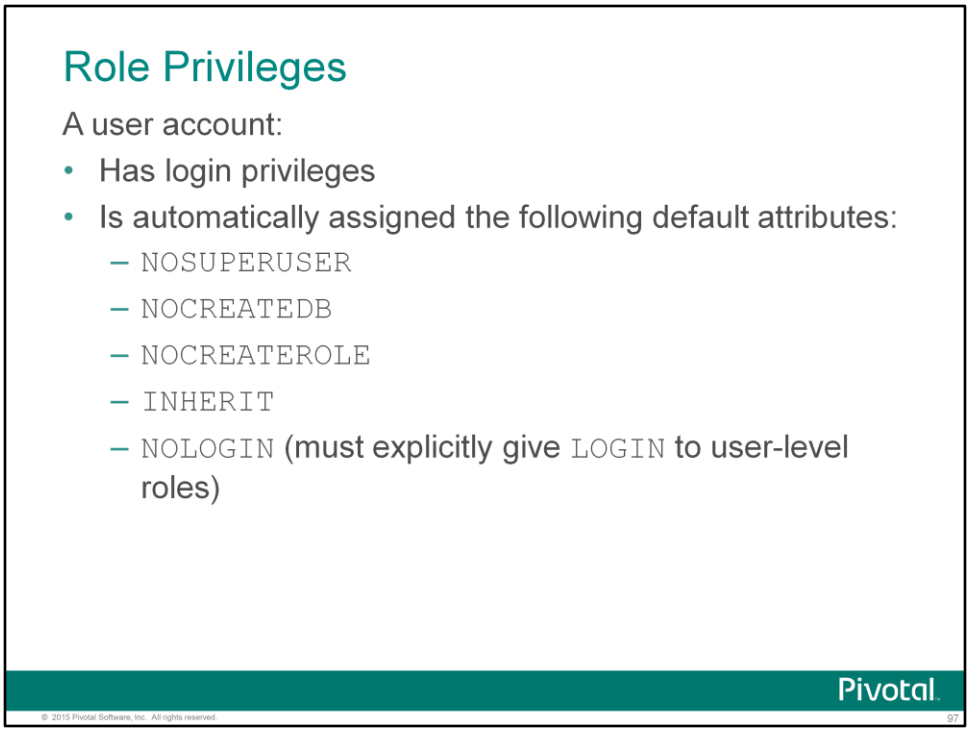


Greenplum Database manages database access permissions using the concept of roles. The concept of roles subsumes the concepts of users and groups. A role can be a database user, a group, or both.

Greenplum Database roles:

- Are separate from the users and groups managed by the operating system on which the server runs. For your convenience, you may want to maintain a relationship between operating system user names and Greenplum Database role names, as many of the client applications use the current operating system user names.
- Have attributes that determine what their permission levels are.
- Can own database objects, such as tables, and can assign privileges on those objects to other roles to control access to the objects. The role that creates an object becomes its owner by default.
- Can be members of other roles, thus a member role can inherit the attributes and privileges of its parent role.
- Are defined at the system level, meaning they are valid for all databases in the system. Managing roles and privileges in Greenplum Database is the same as in PostgreSQL.

Every freshly initialized Greenplum Database system contains one predefined role. This role is a superuser role. By default it has the same name as the operating system user that initialized the Greenplum Database system. In our example, the role is `gpadmin`. Before creating more roles, you first have to connect to Greenplum the superuser role.

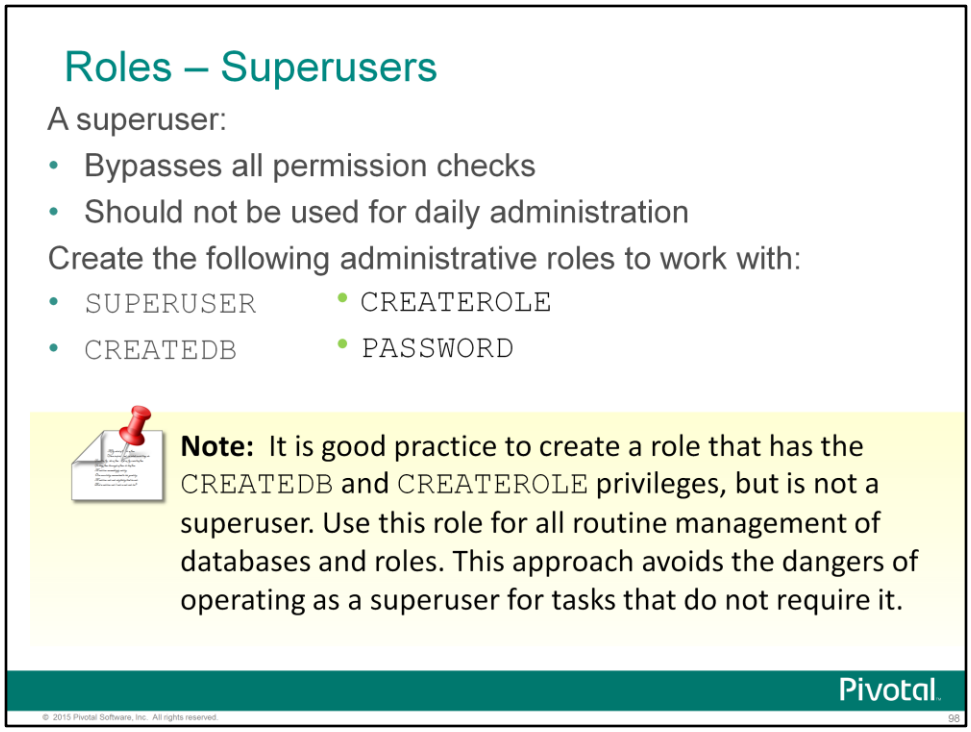


A user-level role is a role that can log in to the database and issue commands. By default, when you create a new role without declaring any attributes, it is assigned the following permissions:

- **NOSUPERUSER** – No superuser (a superuser is someone who have complete permissions and privileges to everything – should be given with caution)
- **NOCREATEDB** – Cannot create databases
- **NOCREATEROLE** – Cannot create other roles
- **INHERIT** – Does inherit the permissions and privileges of the roles it's a member of
- **NOLOGIN** – No login capabilities (must explicitly give this to user accounts)

While you can create roles with `LOGIN` privileges using the `CREATE USER` command, this command is deprecated in PostgreSQL 8.1. The recommended method is to use the `CREATE ROLE` command. You can also use the Greenplum client, `createuser`, to create a role.

You may create nested roles, where a role is a parent to another role. However, you should not have nested users, or users who are parents to other roles.



A database superuser bypasses all permission checks.

This is a dangerous privilege and should not be used carelessly. It is best to do most of your work as a role that is not a superuser.

To accomplish this, complete the following tasks:

- Create a database superuser called SUPERUSER – Use the command `CREATE ROLE` name `SUPERUSER` to create a SUPERUSER role. You must do this as a role that is already a superuser.
- Create a role that can create databases – The new role must be explicitly given permission to create databases. To create such a role, use the command, `CREATE ROLE name CREATEDB`.
- Create a role that can create roles – A role must be explicitly given permission to create more roles. To create such a role, use `CREATE ROLE name CREATEROLE`.
- A role with CREATEROLE privilege can alter and drop other roles, too, as well as grant or revoke membership in them. However, to create, alter, drop, or change membership of a superuser role, superuser status is required; `CREATEROLE` is not sufficient for that.
- Assign a password – A password is only significant if the client authentication method requires the user to supply a password when connecting to the database. The password, md5, and crypt authentication methods make use of passwords. Database passwords are separate from operating system passwords. Specify a password upon role creation with CREATE ROLE name PASSWORD 'string'.

Common Role Attributes

Role Attribute	Description
`SUPERUSER | NOSUPERUSER`	Determines if the role is a superuser. You must yourself be a superuser to create a new superuser. `NOSUPERUSER` is the default.
`CREATEDB | NOCREATEDB`	Determines if the role is allowed to create databases. `NOCREATEDB` is the default.
`CREATEROLE | NOCREATEROLE`	Determines if the role is allowed to create and manage other roles. `NOCREATEROLE` is the default.
`INHERIT | NOINHERIT`	Determines whether a role inherits the privileges of roles it is a member of. `INHERIT` is the default.
`LOGIN | NOLOGIN`	Determines whether a role is allowed to log in. `NOLOGIN` is the default.
`CONNECTION LIMIT` *`connlimit`*	If role can log in, this specifies how many concurrent connections the role can make. -1 (the default) means no limit.

Pivotal

© 2015 Pivotal Software, Inc. All rights reserved.

The table lists common user attributes you use to enforce security.

Common Role Attributes (Cont)

Role Attribute	Description
`PASSWORD 'password'`	Sets the role's password. A null password can optionally be written explicitly as `PASSWORD NULL`.
`ENCRYPTED | UNENCRYPTED`	Controls whether the password is stored encrypted in the system catalogs. The default behavior is determined by the configuration parameter password_encryption (currently set to MD5).
`VALID UNTIL 'timestamp'`	Sets a date and time after which the role's password is no longer valid. If omitted the password will be valid for all time.
`RESOURCE QUEUE queue_name`	Assigns the role to the named resource queue for workload management.

Pivotal

© 2015 Pivotal Software, Inc. All rights reserved.

Common Role Attributes (Continued)

SQL Commands for Roles

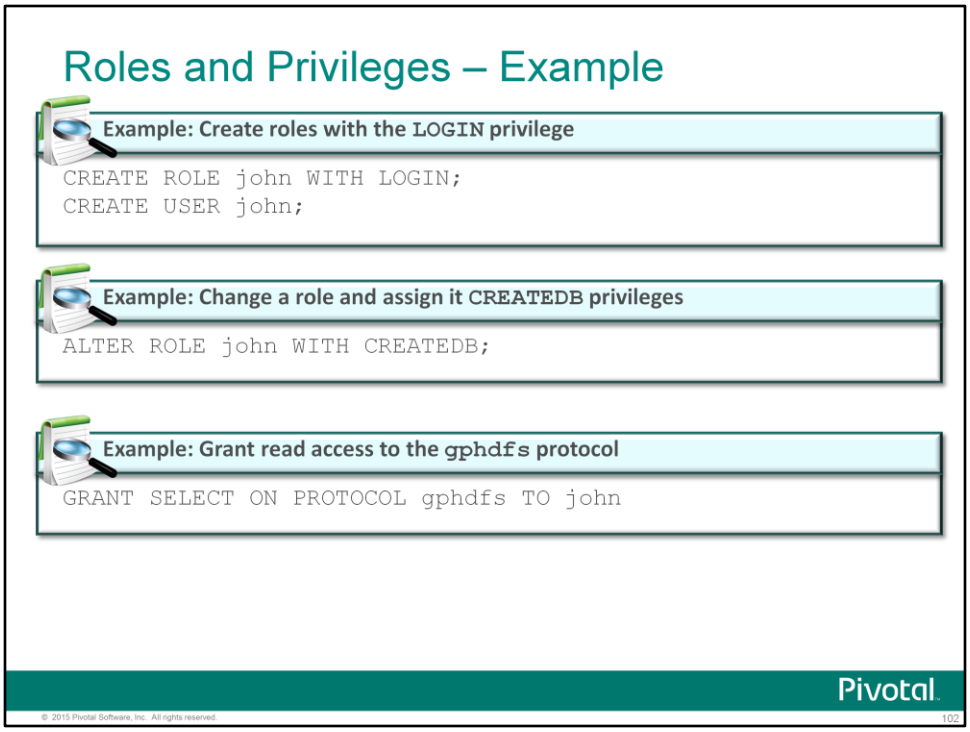
Use the following SQL commands and Greenplum applications to manage roles:

Action	SQL Syntax	Greenplum Application
Create a role	`CREATE ROLE`	`createuser`
Drop a role	`DROP ROLE`	`dropuser`
Alter a role	`ALTER ROLE`	

Pivotal

© 2015 Pivotal Software, Inc. All rights reserved.

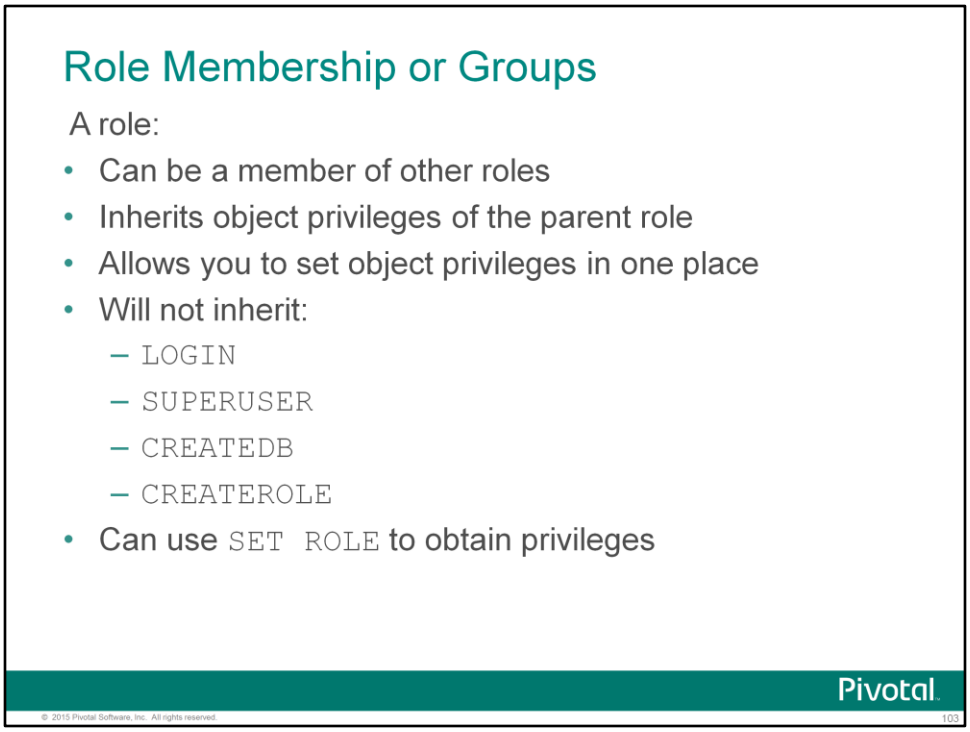
You can use the SQL commands, `CREATE ROLE`, `DROP ROLE`, and `ALTER ROLE` to manage users. The Greenplum applications, createuser and drop user can also be used to manage roles. Only superusers with CREATEROLE privileges can create roles, and so these SQL commands or Greenplum applications must be invoked by a user role with appropriate privileges, such as the superuser role.



The first example on the slide shows two roles being created, both with login privileges.

The second example shows the `ALTER ROLE` command being used to change permission attributes for a user.

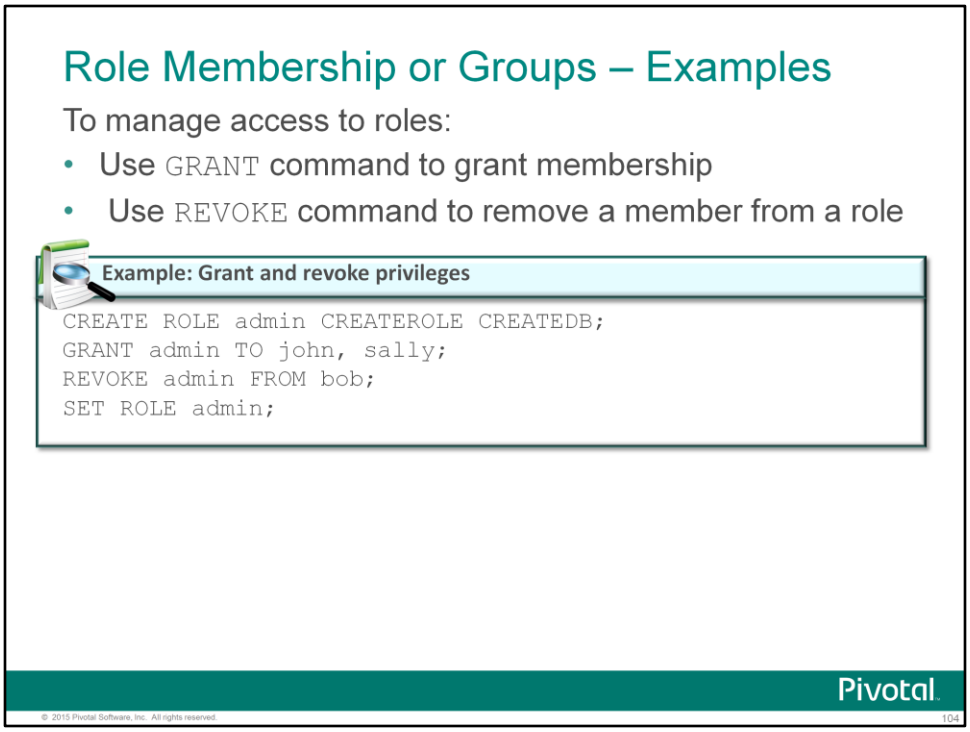
The last example has the `SELECT` access being granted to the role, `john`, so that the role can access it, such as when creating a read-only external table using the `gphdfs` protocol.



It is frequently convenient to group users together to ease management of permissions and privileges. This allows you to grant or revoke privileges for a group of users. In PostgreSQL this is done by creating a role that represents the group and granting membership in the group role to individual user roles.

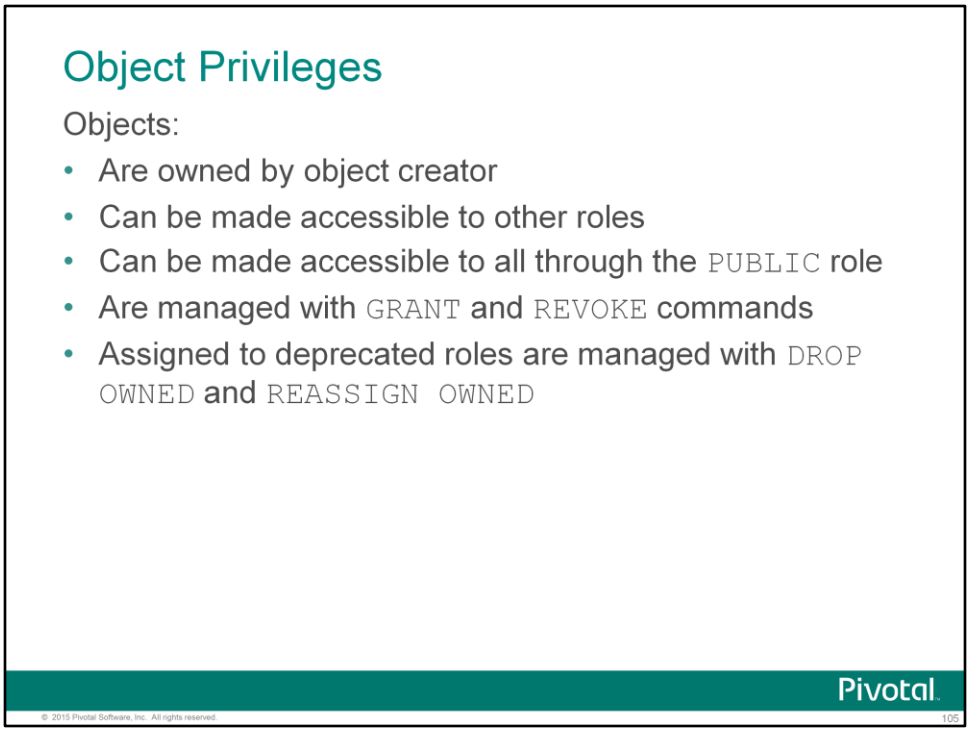
The role attributes `LOGIN`, `SUPERUSER`, `CREATEDB`, and `CREATEROLE` are never inherited as ordinary privileges on database objects. Using the `SET ROLE` command, you must assign a role that already has these attributes defined.

In the example on the slide, the `admin` role is granted `CREATEDB` and `CREATEROLE` attributes. If the user sally is a member of the admin role, she can issue the following command to assume the role attributes of the parent role: `SET ROLE admin;`



Once the group role exists, you can add and remove members (user roles) using the `GRANT` and `REVOKE` commands. Those user roles then inherit the attributes and privileges of the parent role.

In the example shown, the `admin` role is assigned to two roles, `john` and `sally`. Both roles now inherit the `CREATEROLE` and `CREATEDB` privileges. In the last line of the example, the current session user has been changed to `admin`.



When an object, such as a table, is created, it is assigned to an owner. The owner is normally the role that created the object. This allows the role completely manage this object.

The owner role is given all privileges on that object and can assign privileges to other roles. The owner, or a superuser, are the only roles that can drop the object.

`PUBLIC` is a special predefined role that includes all roles, even ones that are created later.

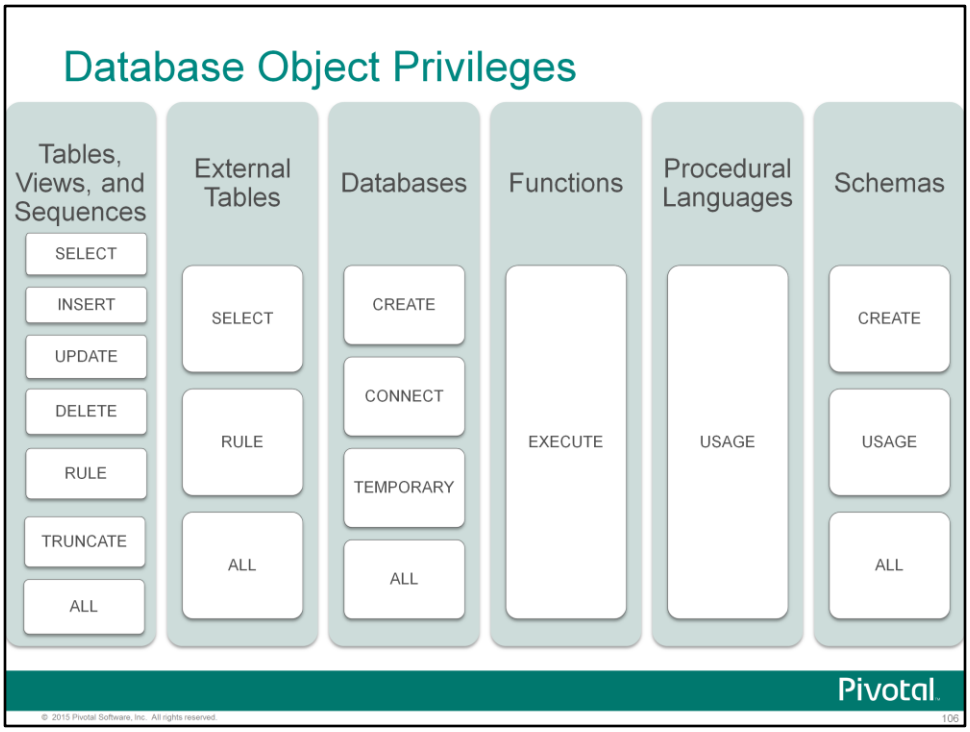
Use the following commands to manage objects owned by deprecated roles:

- **`REASSIGN OWNED`** reassigns all the objects in the current database that are owned by *old_row* to *new_role*. Note that it does not change the ownership of the database itself.
- **`DROP OWNED`** drops all the objects in the current database that are owned by one of the specified roles. Any privileges granted to the given roles on objects in the current database will also be revoked.

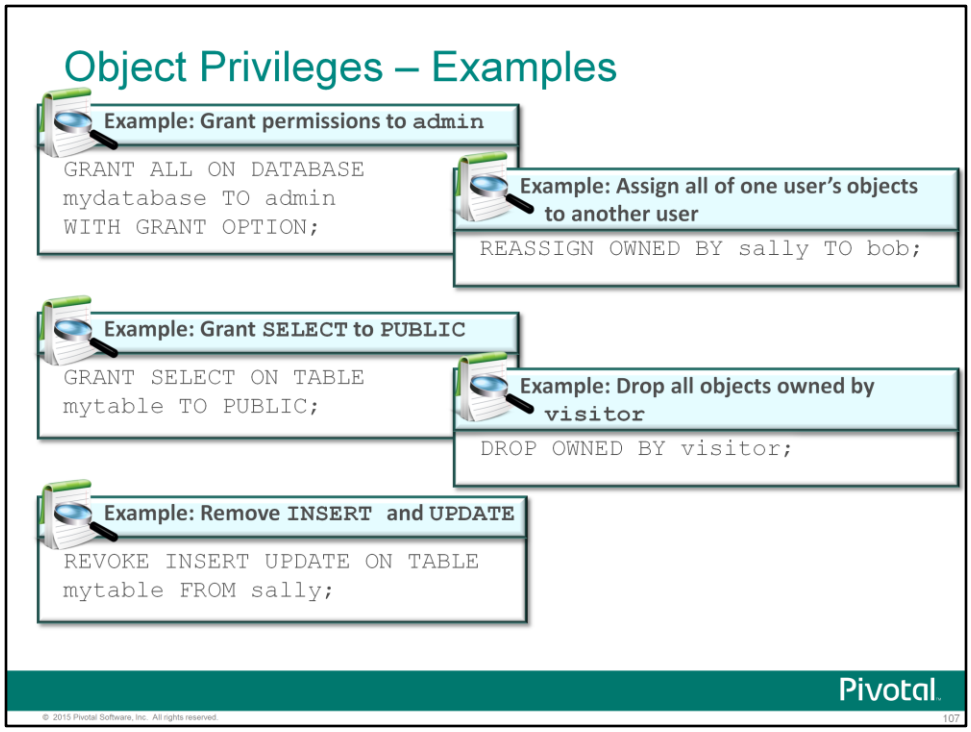
Note: Privileges are not passed down in inherited table structures from parent to child tables. Must be explicitly set on all tables.

You must grant access to an object if other users wish to use it. For managing object privileges, you grant the appropriate permissions to the group-level role only.

The member user roles then inherit the object privileges of the group role.

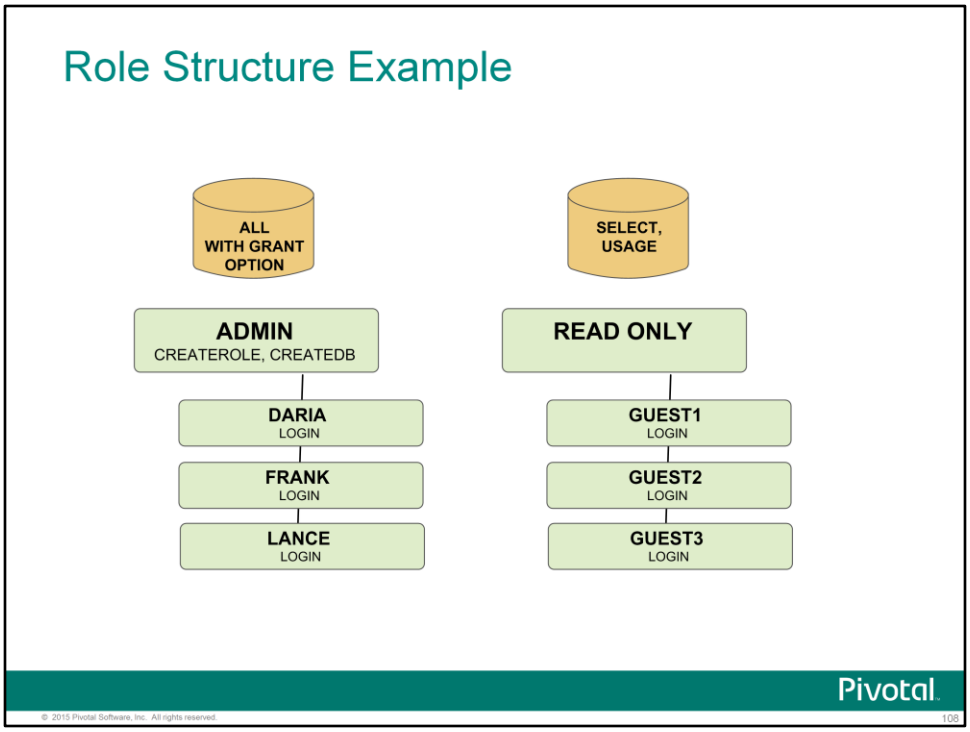


These are the database objects and associated privileges that are relevant in Greenplum Database. Privileges may vary based on the object type. Privileges provide access rights to the named objects, allowing roles to read, access, or modify the object. Privileges are not hierarchical. For example granting `ALL` on a database does not include `ALL` on the tables in that database. Those privileges must be assigned individually.



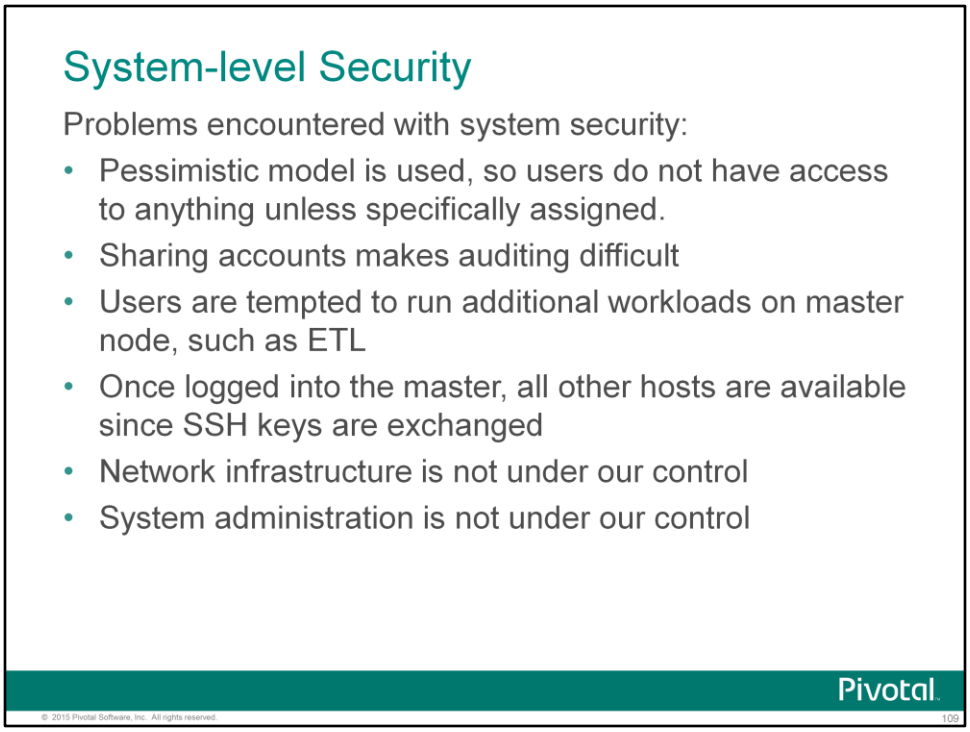
Here are some examples of how to alter privileges for users and objects:

- In the first example, all privileges are granted to the user role, `admin`, for the database `mydatabase`. The `admin` role is also allowed to grant permissions to other roles for the specified database.
- The second example on the left side shows only `SELECT` privileges on the table, `mytable`, are granted to the role, `public`.
- The third example on the left side of the slide shows both `INSERT` and `UPDATE` permissions on the table, `mytable`, being revoked for the role, `sally`.
- The first example on the right side of the slide shows that all objects owned by one role, `sally`, are being reassigned to another role, `bob`.
- In the last example on the slide, all objects owned by the role, `visitor`, are removed from the database.



On this slide is an example of two role hierarchies you can define using the following specifications:

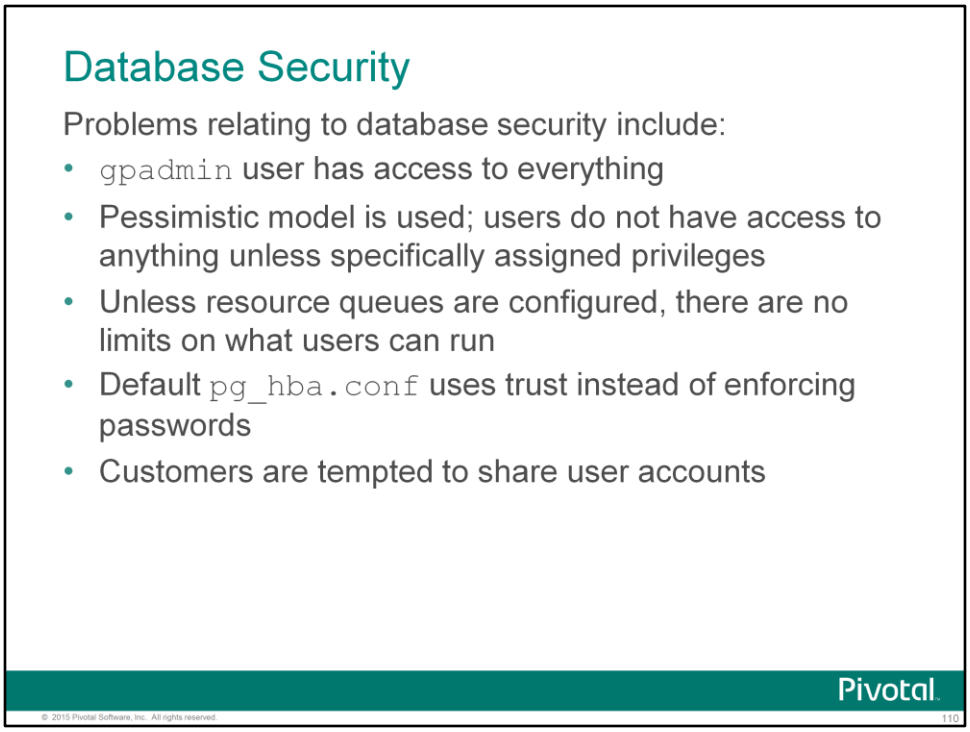
- The first is a top level role named `ADMIN` with permissions to create roles and databases.
- The second is a role called read-only with no create permissions.
- Each Group level role has login accounts associated with it. Note that the permission attributes are assigned at the parent level.
- Privileges are assigned at the parent-level role. For the admin role, you can assign all privileges to all database objects and also give the grant option so members of that role could grant privileges to other roles.
- For the read-only role, assign the `SELECT` privileges on tables. You can also assign `USAGE` on schemas.



In system level security, the concern is about defining privileges for users who access the environment. Unauthorized users with access to the system increase their chances of gaining access to data.

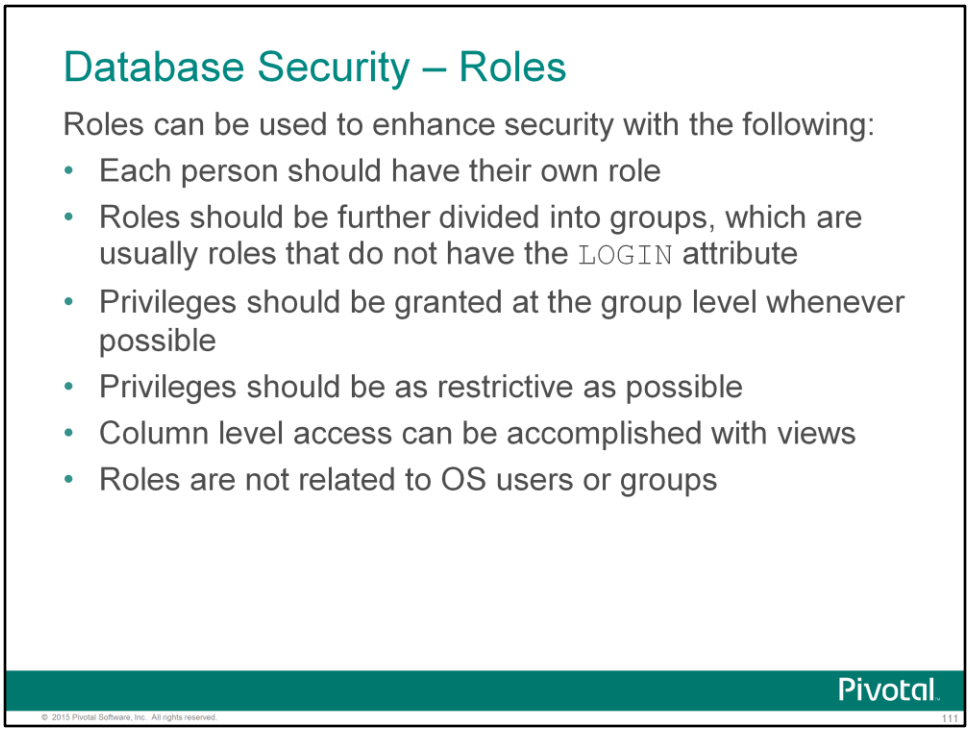
Problems that are oftentimes encountered with securing a system include:

- Pessimistic model prevents users from having access to specific items unless specifically assigned. While there is a benefit in that, as roles are implemented at the system level, it can that a user is prevented from performing tasks to which they have been assigned.
- Account sharing can make auditing a difficult task. It is not easy to determine who did what, when, and who approved the action if one or two major accounts are used by multiple individuals.
- Users may be tempted to hog resources and run additional workloads on the master, including data loads through ETL. This can freeze other users out of resources until that action is completed.
- Once logged into the master, all other hosts are available to the user on the master node since SSH keys have been exchanged.
- Network security issues outside of the control of the database can have an adverse impact.
- Other system administration outside of the database can also have an adverse impact.



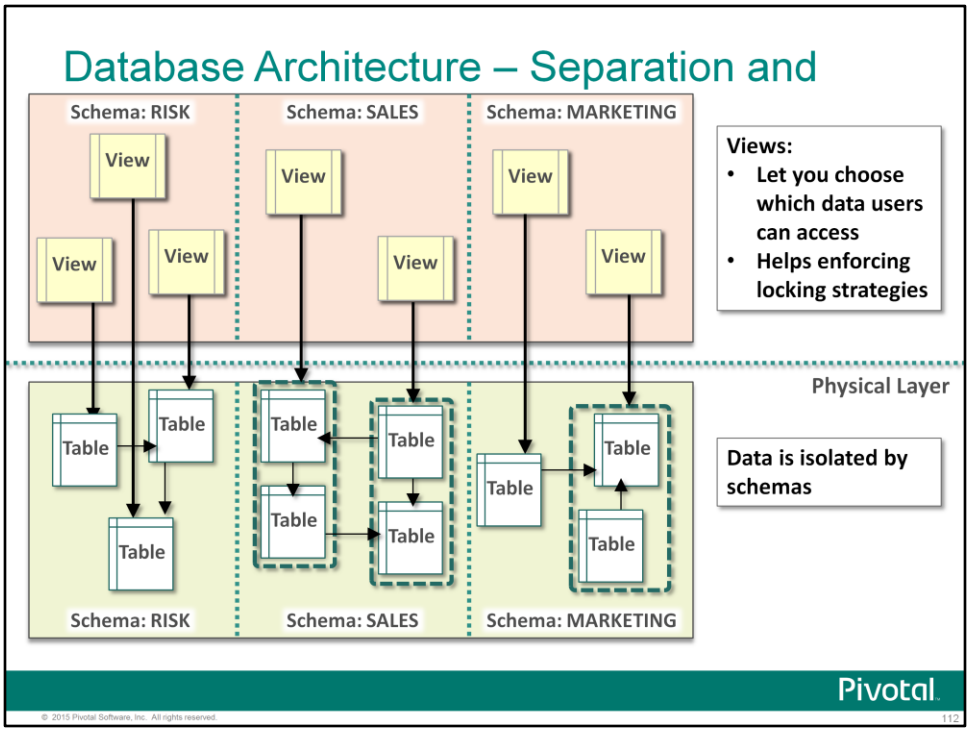
Problems that are specifically associated with the database include the following:

- The superuser role created on database initialization has access to every aspect of the system. This can be dangerous particularly if the role is used for daily administration and is shared by multiple users.
- Pessimistic model usage in the database has benefits and detractors. By employing a pessimistic model, users do not have access until specifically assigned. However, if the correct privileges are not assigned, this can cause frustration and delay for that user.
- There are no limits on resources unless specifically configured. Once again, a user can hog resources and effectively, shut other users out of the database.
- The default configurations in the `pg_hba.conf` are loosely configured to use `trust`. This bypasses security measures that should be in place to require authentication from the connecting user.
- As with regular system-level security, users may be tempted to share user accounts. Users, or roles, can be defined for each user to protect their data. If multiple users share the same role, this reduces the chances of finding who had access to the system and to data.



You can implement the following rules to tighten security using roles:

- Roles are implemented at the system-level and are separate from operating system users. Each person that connects to the database should have their own role assigned to them. Users are roles with login attributes associated with the role. By assigning a role to each individual user, you increase the chances of properly auditing the system to ensure that the appropriate users have access to appropriate data. Application services should also be given a distinct role in the environment.
- Roles should be divided into groups to ease administration tasks. Groups represent role membership, where role is created and membership to that role is granted. The role can be assigned privileges which are inherited by members of the role. This makes it easier to assign privileges to users.
- By assigning privileges at the group level, you reduce the chances that an individual user may have a privilege that should have been revoked. You can also more quickly assign a group of privileges to users by adding them to the appropriate group.
- While the pessimistic model is in use, assigned privileges should be as restrictive as possible. Do not assign superuser privileges to a user who only needs to create roles.
- You can restrict access to data by implementing views instead of providing full access to tables.
- You can make it harder for someone to *guess* what a valid account is by removing the relationship between operating system users and groups to Greenplum roles. If you use the same names, you increase the chances that someone can guess role names.



Users are isolated at the physical layer through the use of views. By isolating users, you:

- Insulate users from changes made to tables
- Control access to columns and rows
- Enforce locking strategies

Data is isolated by separating functional areas into schemas. By doing this, you:

- Simplify security
- Simplify backups and restores to data

Security Example

The following is an example of user and role creation and assignment and how to use the roles to limit or grant privileges to user roles:

Example: Inherit privileges through nested roles

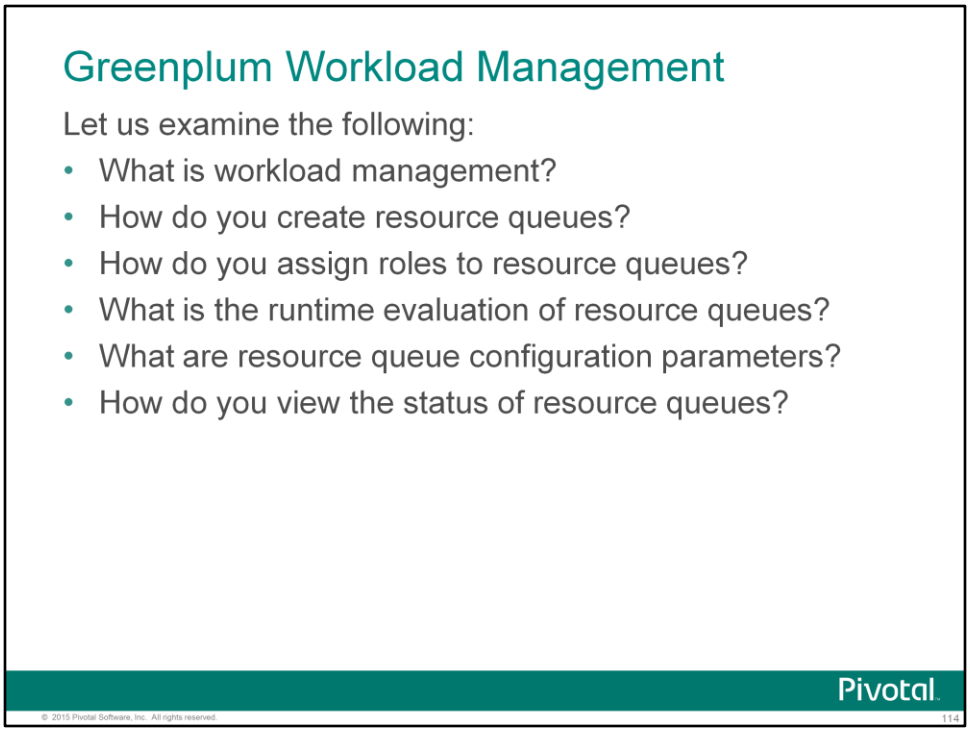
```
CREATE ROLE batch;
GRANT select, insert, update, delete
    ON dimensions.customer TO batch;
CREATE ROLE batchuser LOGIN;
GRANT batch TO batchuser;
```

Pivotal

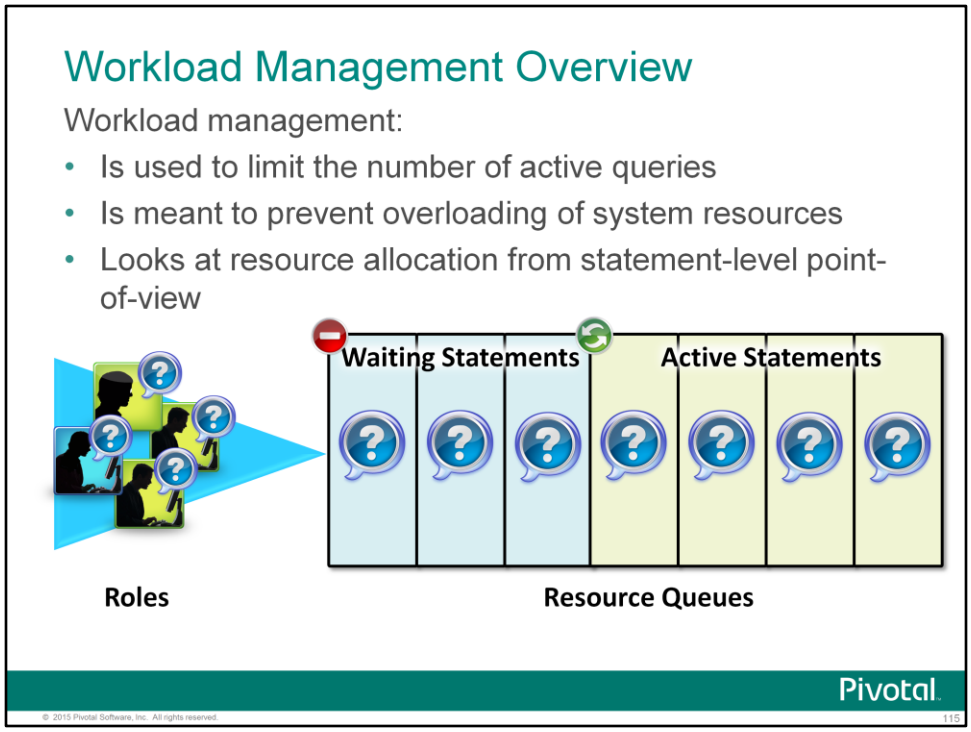
© 2015 Pivotal Software, Inc. All rights reserved.

In this example:

- The newly created role, `batch` is granted `SELECT`, `INSERT`, `UPDATE`, and `DELETE` privileges on the table `dimensions.customer`.
- The role is not a user as it was not assigned the `LOGIN` attribute.
- The `batchuser` role is then created with the `LOGIN` attribute and granted access to the `batch` role.
- The user, `batchuser`, can now has the privileges assigned to the role, `batch`.
- The object that the role is given access to is fully qualified, indicating the role has access only to the `customer` table in the `dimensions` schema.



In this next section, you examine workload and resource management, a feature that gives you the ability to more closely limit the number of active queries in the system at any given time.



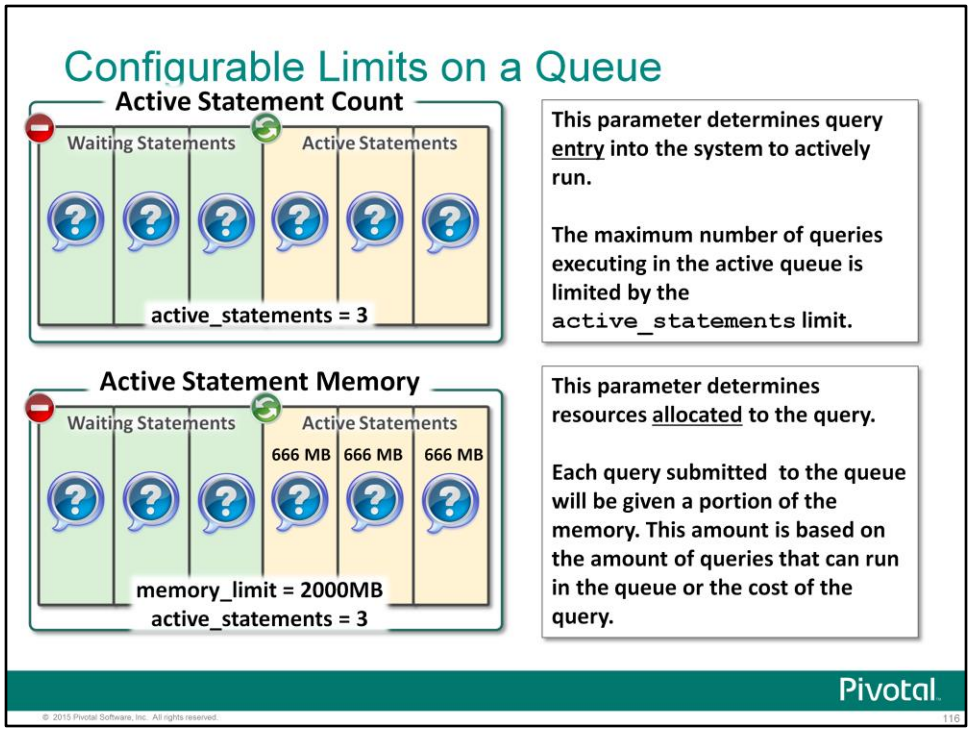
The purpose of Greenplum Database workload management is to limit the number of active queries in the system at any given time in order to avoid exhausting system resources such as memory, CPU, and disk I/O.

This is accomplished by creating role-based *resource queues*. A resource queue has attributes that limit the size and/or total number of queries that can be executed by the users (or roles) in that queue.

By assigning all of your database roles to the appropriate resource queue, administrators can control concurrent user queries and prevent the system from being overloaded.

Administrators create resource queues for the various types of workloads in their organization. For example, you may have a resource queue for power users, web users, and management reports. The administrator would then set limits on the resource queue based on his estimate of how resource-intensive the queries associated with that workload are likely to be. Database roles, which include users and groups, are then assigned to the appropriate resource queue. A resource queue can have multiple roles, but a role can only be assigned to a single resource queue.

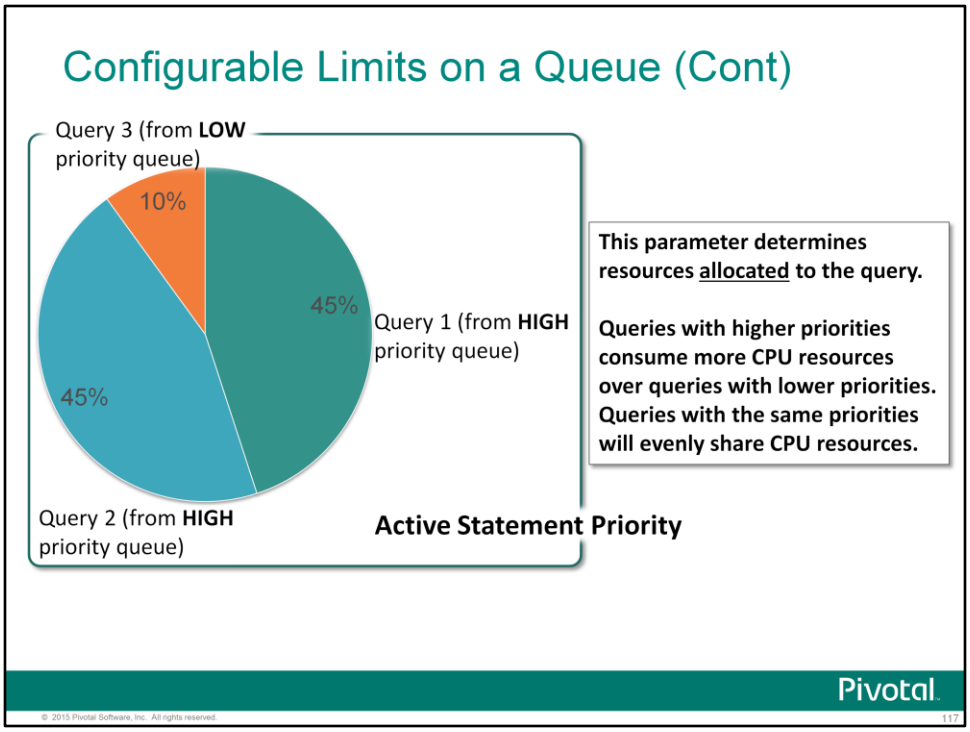
Starting with Greenplum 4.1, the Greenplum memory management system looks at allocating memory and CPU resources not from an operator or user point of view, but instead from the statement level point of view.



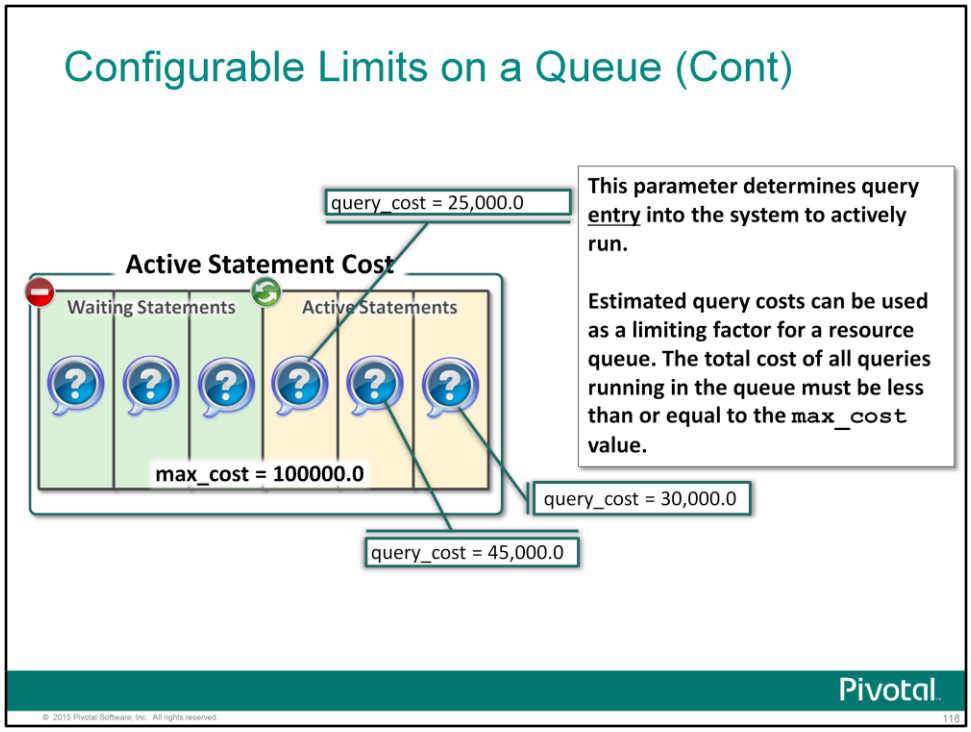
Resource queues can be created to address different types of workloads, such as workloads for power users, web users, and management reports. A queue for a web user may be fast and require very little resources, whereas the workload for management reports may be more intensive, requiring more resources, such as memory.

You can set limits on a queue using the following configurable limits:

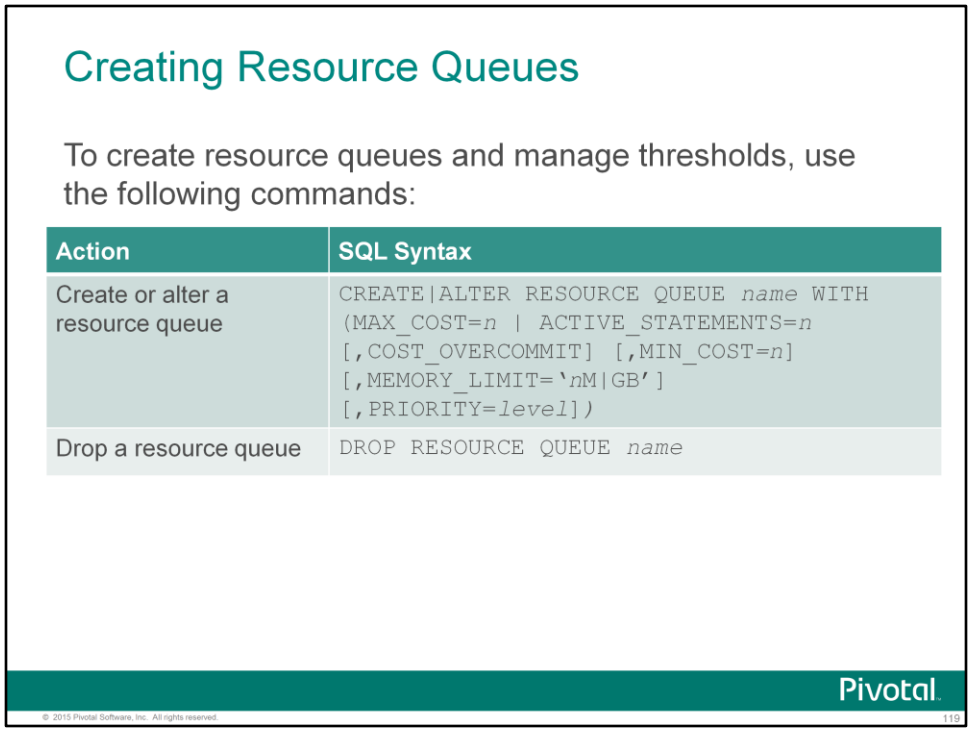
- **Active statement count –** This limit allows you to set the maximum number of statements that can run concurrently. The total number of queries running in a resource queue with an active statement limitation must be less than or equal to the defined ceiling. If the maximum number of queries are running, an additional query that enters the queue must wait for one of the active queries to complete before being allowed to execute on the system.
- **Active statement memory** – The active statement memory limit lets you set the total amount of memory that all queries assigned to the resource queue can consume. No additional memory will be allocated to the queue once the maximum has been reached and each query is allocated the same amount of memory, which is the total available memory divided by the number of active queries. It is recommended you use this limit in conjunction with the active statement count limit.



- **Active statement priority –** The active statement priority queue lets you define the relative priority of the queue to other queues in terms of available CPU resources. All queries entering the queue are given the same priority. If the priority of one resource queue is higher than the priority on another resource queue, the higher priority queue will consume more CPU resources.



- **Active statement cost –** The query planner estimates a cost for each query submitted. You can set the maximum query planner cost for a resource queue, limiting the number of queries that can enter the queue by the total cost of all queries in the queue. The cost is measured in units of disk page fetches, where 1.0 is equal to one sequential disk page read.



Greenplum has added the SQL commands `CREATE RESOURCE QUEUE`, `ALTER RESOURCE QUEUE`, `DROP RESOURCE QUEUE` which you will use to create and manage the limits of your resource queues. Queries gain entry into the system so that they can actively run with the following resource queue limits:

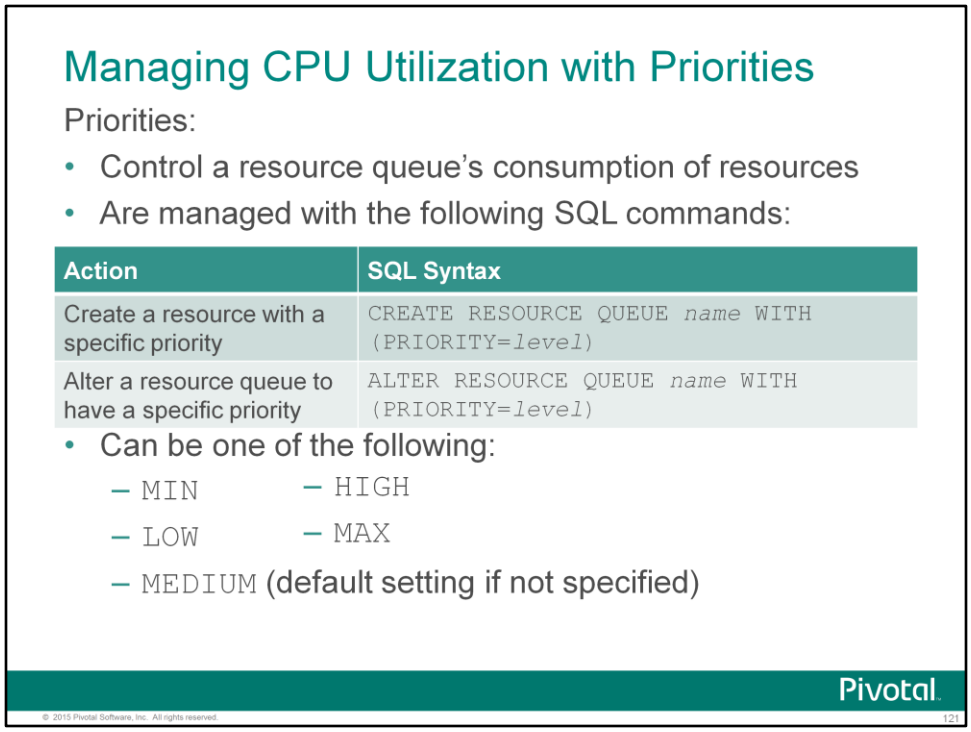
- **`ACTIVE_STATEMENTS`** – Limits the number of queries that can be executed by roles assigned to that queue.
- **`MAX_COST`** – Limit the total size of queries that can be executed by roles assigned to that queue based on query cost. Cost is measured in the estimated total cost for the query as determined by the Greenplum query planner. Therefore, an administrator must be familiar with the queries typically executed on the system in order to set an appropriate cost threshold for a queue. Cost is measured in units of disk page fetches; 1.0 equals one sequential disk page read.

The following additional settings can be used in conjunction with `ACTIVE_STATEMENTS` and `MAX_COST` to determine how much resources queries receive once they are actively running:

- **`MEMORY_LIMIT`** – Setting this limit sets the total memory quota for all queries within the resource queue. Each query is allocated a pre-determined amount of memory.

Creating Resource Queues (Continued)

- **`COST_OVERCOMMIT`** – Queries that exceed the cost threshold can continue executing, provided there are no other queries in the system at the time the query is submitted.
- **`MIN_COST`** – The query cost limit of what is considered a small query. Queries with a cost under this limit will not be queued but will instead run immediately.
- **PRIORITY** – The priority does not determine which queries enter the active queue, but instead decides how much CPU resources the query receives. Queries with higher priority receive a larger share of available CPU resources, decreasing the CPU resources available to any in-flight queries with lower priorities.



To control a resource queue's consumption of available CPU resources, you can assign an appropriate priority level. When high concurrency causes contention for CPU resources, queries and statements associated with a high-priority resource queue will claim a larger share of available CPU than lower priority queries and statements.

Priority settings are created or altered using the WITH parameter of the commands `CREATE RESOURCE QUEUE` and `ALTER RESOURCE QUEUE`.

You can assign priorities with the following levels:

- **`MIN`** – This is the lowest possible priority setting for a resource queue.
- **`LOW`** – At one time, the lowest possible available setting, resource queues with this level consume great CPU resources than those with a level of `MIN`.
- **`MEDIUM`** – This is the default level and is automatically assigned when a PRIORITY is not defined for the resource queue.
- **`HIGH`** – Queries with this setting take precedence over those assigned to MEDIUM or LOW resource queues.
- **`MAX`** – This is the highest level that can be assigned to a resource queue. Queries with this setting take precedence over all others.

Managing Memory Allocation with Memory Limits

Query memory allocation is controlled with:

- `MEMORY_LIMIT` parameter for the resource queue:

Action	SQL Syntax	Memory Allocation per Query
Create or alter a resource with a specific memory limit and maximum number of statements	`CREATE|ALTER RESOURCE QUEUE name WITH (ACTIVE_STATEMENTS = x, MEMORY_LIMIT='xxxM|GB')`	`MEMORY_LIMIT/ ACTIVE_STATEMENTS`
Create or alter a resource queue to have a specific memory limit with `MAX_COST` defined	`CREATE|ALTER RESOURCE QUEUE name WITH (MAX_COST = x.x, MEMORY_LIMIT='xxxM|GB')`	`MEMORY_LIMIT * (query_cost / MAX_COST)`

- At the database level with `statement_mem` and `max_statement_mem`

Pivotal

© 2015 Pivotal Software, Inc. All rights reserved.

122

Memory can be controlled both within a resource queue as well as at the database level.

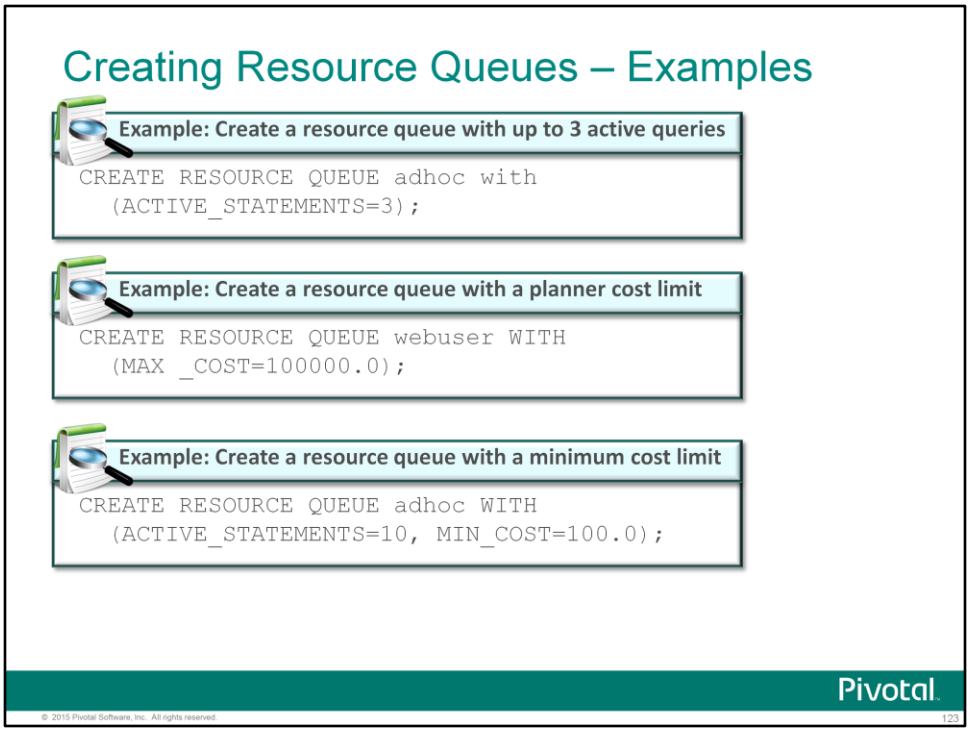
For a resource queue, a maximum amount of memory is allocated to the queue with the MEMORY_LIMIT parameter and value as part of the CREATE RESOURCE QUEUE command. The value specified can be in Megabytes or Gigabytes.

Each query is allocated a certain amount of memory depending on how the queue is configured:

- If the resource queue is created with memory limits and ACTIVE_STATEMENTS set to a specified value, each query will be allocated an equal amount of memory. The memory allocated is determined by the `MEMORY_LIMIT` value divided by the `ACTIVE_STATEMENTS` value.
- If the resource queue is created with memory limits and MAX_COST set to a specified floating value, the query is allocated memory based on the formula, `MEMORY_LIMIT` * (`query_cost` / `MAX_COST`).

The resource queue allocation can be overridden by the use of the `statement_mem` server configuration parameter. A query submitted to the environment after `statement_mem` is set will be allocated the amount of memory defined in `statement_mem`, as long as the amount allocated is less than the value defined in `max_statement_mem`.

One thing to note, the memory limit defined across all resource queues must be less than the total physical memory available to a segment host.



In the first example, for all roles assigned to the `adhoc` resource queue, only three active queries can be running on the system at any given time. If this queue has three queries running, and a fourth query is submitted by a role in that queue, that query must wait until a slot is free before it can run.

In the second example, a resource queue named `webuser` is created with a query cost limit of 100000.0 (1e+5). For all roles assigned to the `webuser` resource queue, it will only allow queries into the system until the planner cost limit of 100000.0 is reached. If this queue has 200 queries with a 500.0 cost all running at the same time, and query 201 with a 1000.0 cost is submitted by a role in that queue, that query must wait until space is free before it can run.

In the third example, any queries that fall below the defined `MIN_COST` limit will run immediately. The restriction of ten active queries still applies however.

Assigning Roles to Resource Queries

Resource queues:

- Must be assigned at the user-level (group-level ignored)
- Do not affect superuser roles
- Are managed with the following SQL commands:

Action	SQL Syntax
Add a resource queue to a role	`ALTER ROLE role_name RESOURCE QUEUE queue_name;`
Create a role and assign it to a resource queue	`CREATE ROLE role_name WITH LOGIN RESOURCE QUEUE queue_name;`

Note: All users are assigned to a resource queue, even if one is not specified. The default resource queue is `pg_default`.

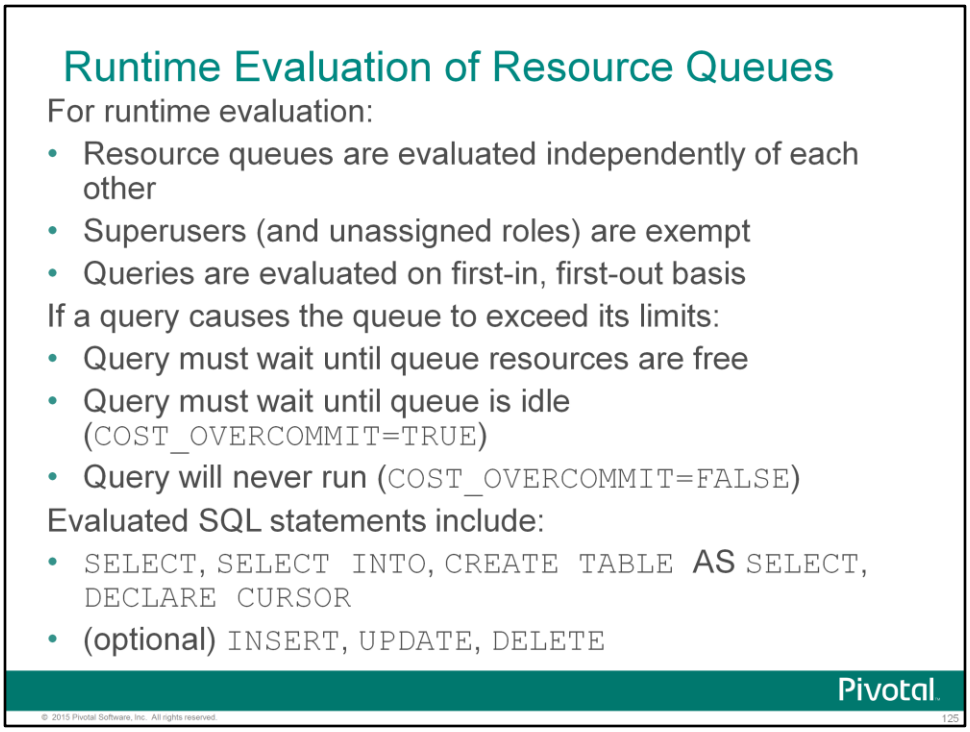
Pivotal

© 2015 Pivotal Software, Inc. All rights reserved.

Currently resource queues must be assigned on a user-by-user basis. If you have a role hierarchy such as a group-level role, then assigning a resource queue to the group does not propagate down to the users.

All superuser roles are exempt from resource queue limits. Even if you assign a superuser role to a resource queue, their queries will always run regardless of restrictions defined for the resource queue.

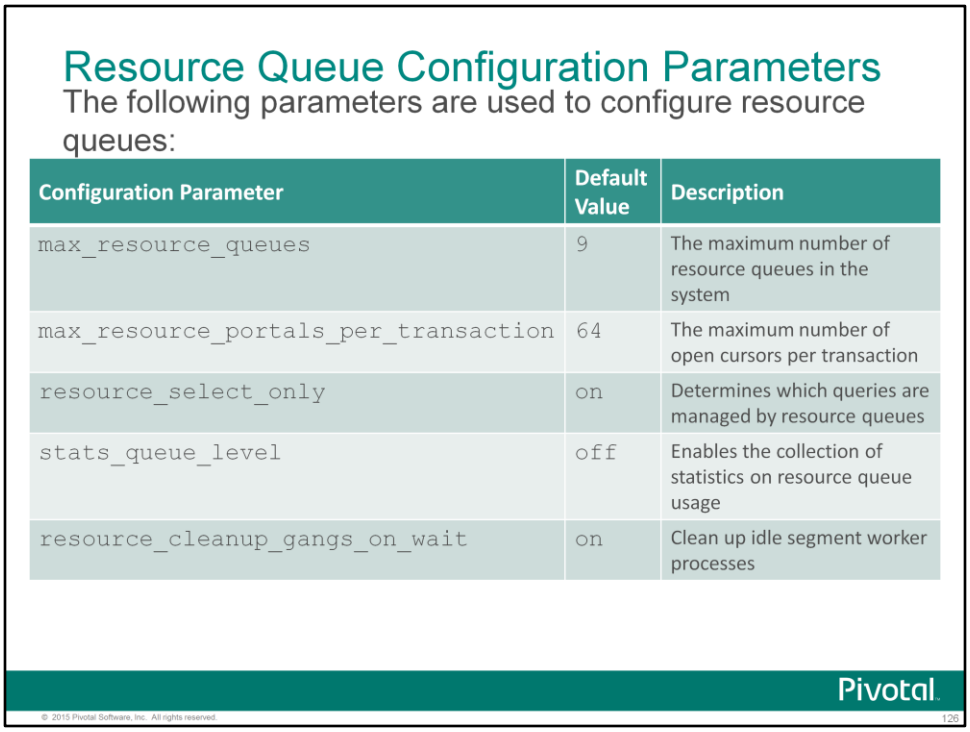
To assign a user to a resource queue use the `CREATE ROLE` and `ALTER ROLE` commands. A user role can only be assigned to one resource queue.



At runtime, when the user submits a query for execution, that query is evaluated against the resource queue's limits. If the query does not cause the queue to exceed its resource limits, then that query will run immediately. If the query causes the queue to exceed its limits, such as if the maximum number of active statement slots are currently in use, then the query must wait until queue resources are free before it can execute. If the queue is idle however, and `COST_OVERCOMMIT` is set to true, a query that exceeds the MAX_COST will be allowed to run. If `COST_OVERCOMMIT` is set to false, the query will never execute.

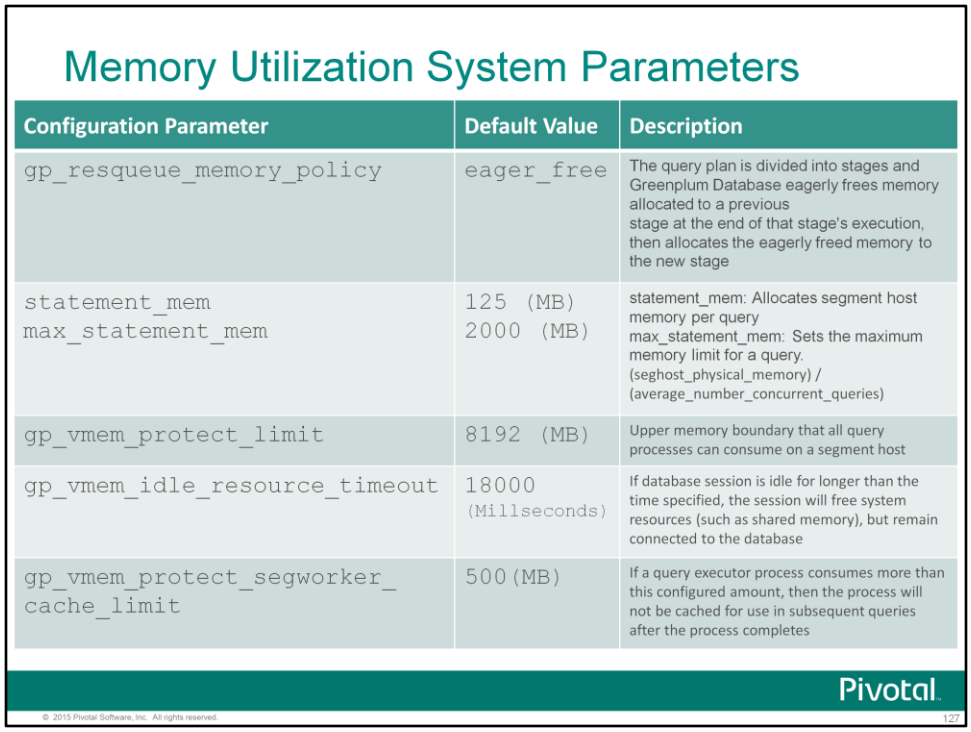
Queries submitted through a queue are evaluated and executed on a first in, first out basis.

Not all SQL statements submitted through a resource queue are evaluated against the queue limits. By default only `SELECT`, `SELECT INTO`, `CREATE TABLE AS SELECT`, and `DECLARE CURSOR` statements are evaluated. If the server configuration parameter `resource_select_only` is set to `off`, then `INSERT`, `UPDATE`, and `DELETE` statements will be evaluated as well.



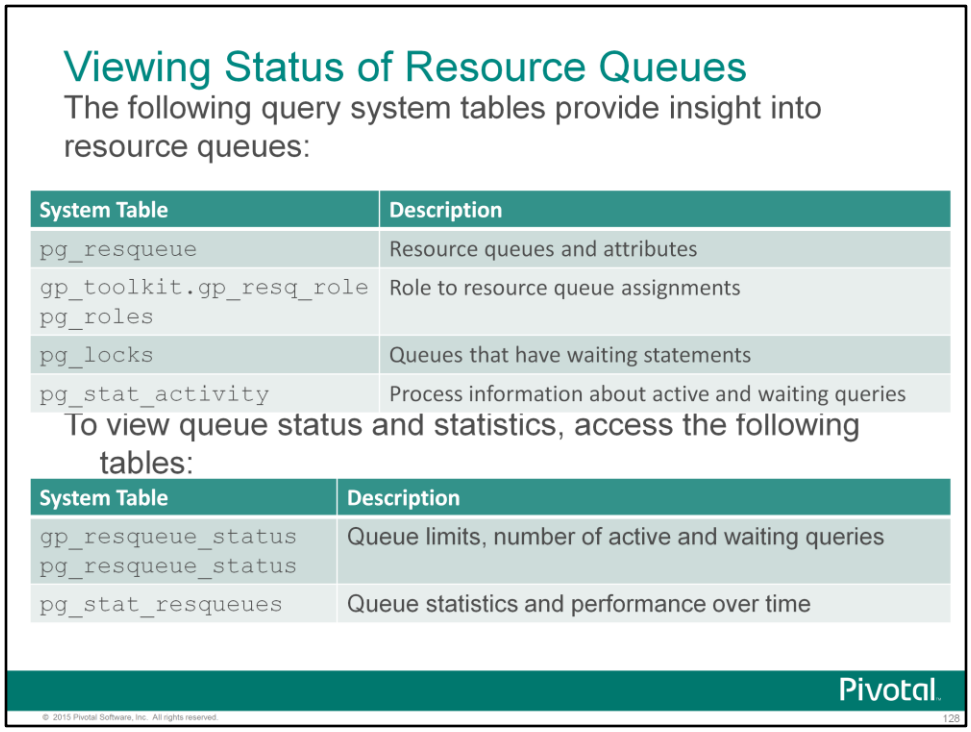
Here are the parameters for resource queues shown with their defaults. All of these must be set in the master postgresql.conf and require server restart to take effect:

- **`max_resource_queues`** — This parameter sets the maximum number of resource queues that can be created in a Greenplum Database system. Note that resource queues are system-wide (as are roles) so they apply to all databases in the system.
- **`max_resource_portals_per_transaction`** — This parameter sets the maximum number of simultaneously open cursors allowed per transaction. Note that an open cursor will hold an active query slot in a resource queue.
- **`resource_select_only`** — This parameter sets the types of queries managed by resource queues. If set to on, then `SELECT`, `SELECT INTO`, `CREATE TABLE AS SELECT`, and `DECLARE CURSOR` commands are evaluated. If set to off `INSERT`, `UPDATE`, and `DELETE` commands will be evaluated as well.
- **`stats_queue_level`** – Setting this parameter to on enables the collection of statistics on resource queue usage, which can then be viewed by querying the `pg_stats_resqueue` system view. This is set to `off` by default.
- **`resource_cleanup_gangs_on_wait`** – Setting this parameter cleans up idle segment worker processes before taking a slot in the resource queue.



The following parameters determine how memory is utilized at the database level:

- **`gp_resqueue_memory_policy`** — The parameter enables Greenplum memory management features, where memory management is done at the statement level instead of at the operator level. The parameter can be set to `auto`, `off`, or `eager_free`. If set to auto or `eager_free`, the new memory management system will be used. Otherwise, if set to `none`, the system prior to Greenplum Database 4.1 will be used.
- **`statement_mem` and `max_statement_mem`** – These parameters are used to both allocate memory to a query and set the maximum amount of memory that can be allocated. Once the `statement_mem` parameter has been set, any queries executed after will ignore the resource queue settings for memory allocation and use this set value.
- **`gp_vmem_protect_limit`** – The parameter sets the upper boundary that all query processes can consume on a segment host. Queries that cause this limit to be exceeded will be cancelled. This parameter must be set at the local level for each segment instance.
- **`gp_vmem_idle_resource_timeout` and `gp_vmem_protect_segworkder_cache_limit`** – These two parameters are used to free memory on segments held by idle database processes. The first parameter sets the resource timeout value in milliseconds while the second sets the cache size in megabytes. If the second parameter is exceeded, the process that exceeded the cache will not be cached for subsequent queries.



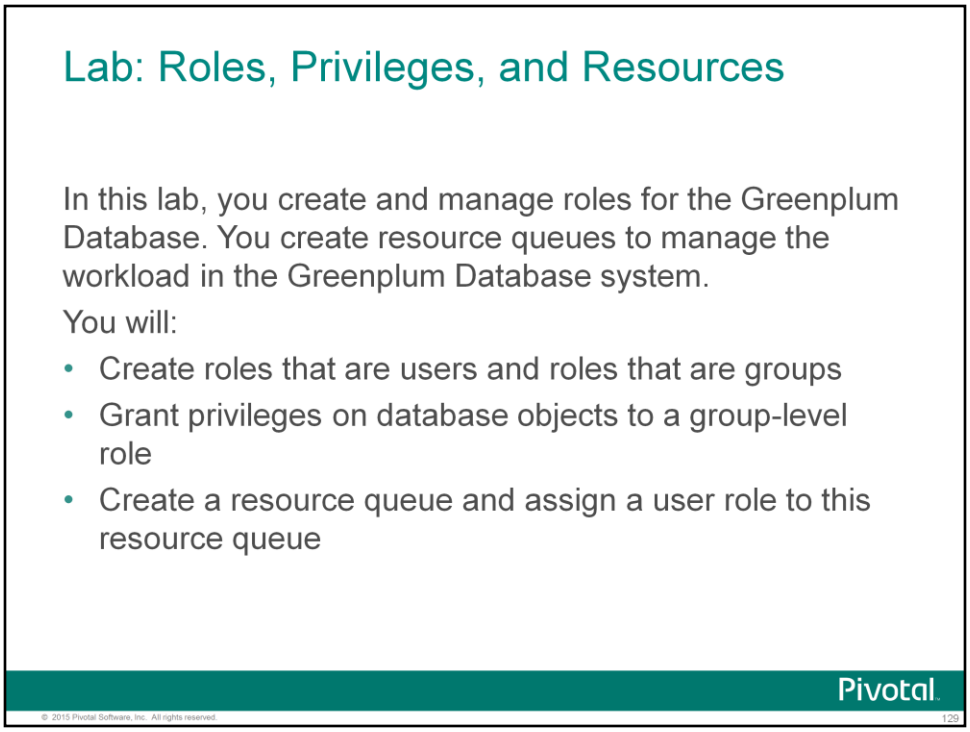
Currently there is no management interface for workload management and resource queues. To see status information about resource queues, you'll have to query and join these system tables. Your training guides have sample queries for finding out certain information about resource queues. You may want to save these queries as VIEWS in your production Greenplum Database system. Several system tables from the gp_toolkit or pg_catalog schemas are available to track resource queue behavior and activity.

The `pg_resqueue_status` system view allows administrators to see status and activity for a workload management resource queue. It shows how many queries are waiting to run and how many queries are currently active in the system from a particular resource queue.

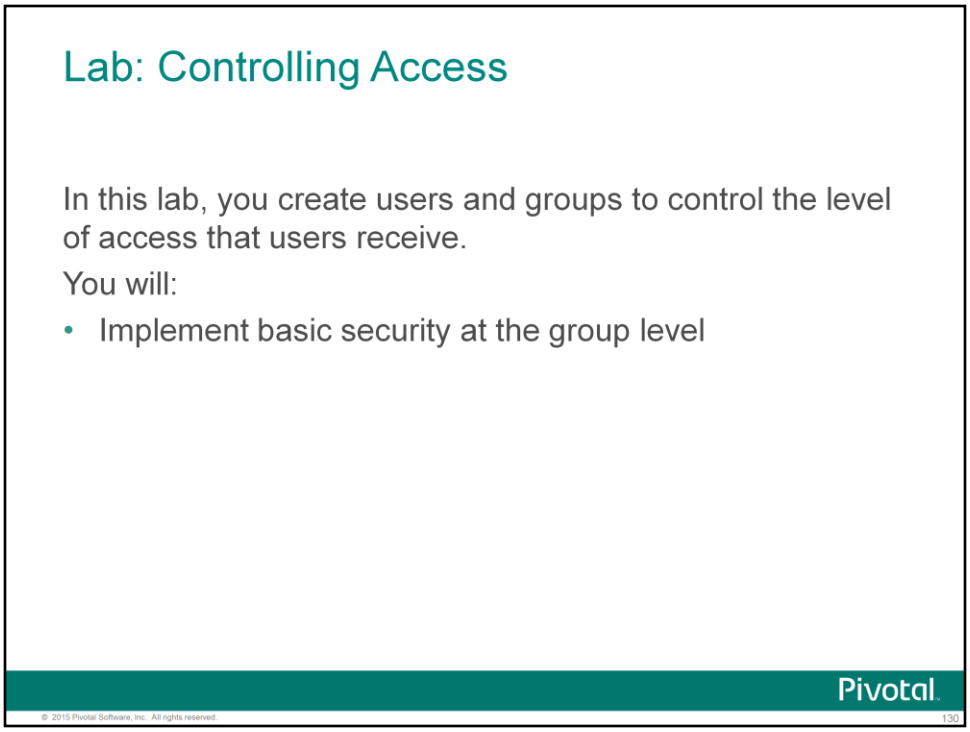
If you want to track statistics and performance of resource queues over time, you can enable statistics collecting for resource queues. This is done by setting the following server configuration parameter in your master `postgresql.conf` file:

```
stats_queue_level = on
```

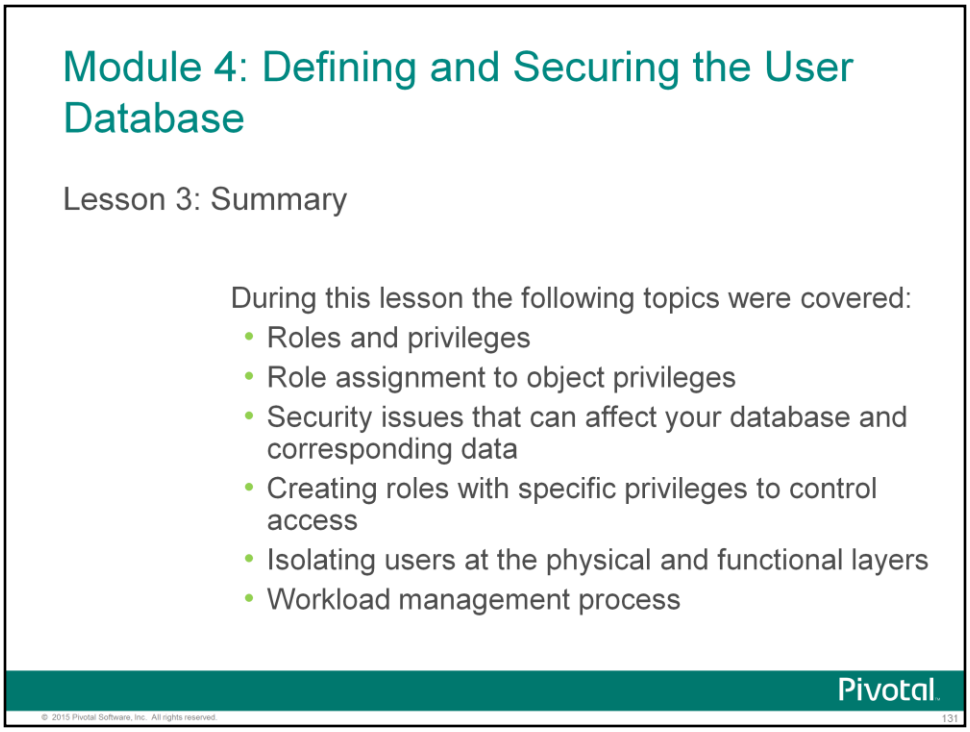
Once this is enabled, you can use the `pg_stats_resqueue` system view to see the statistics collected on resource queue usage. Note that enabling this feature does incur slight performance overhead, as each query submitted through a resource queue must be tracked. It may be useful to enable statistics collecting on resource queues for initial diagnostics and administrative planning, and then disable the feature for continued use.



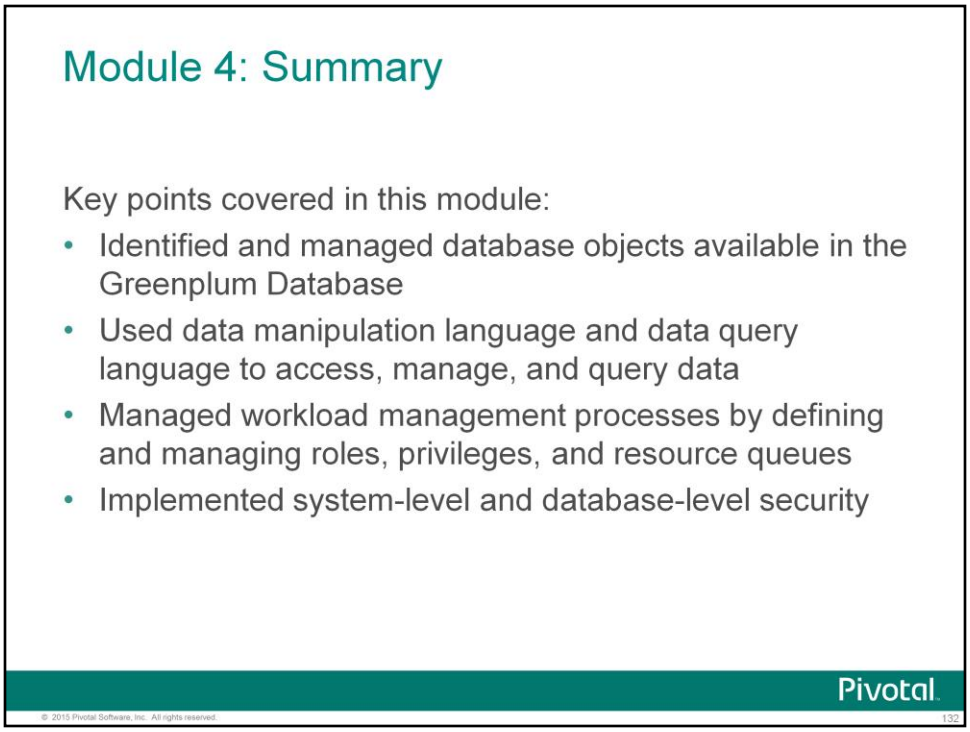
In this lab, you create and manage roles for the Greenplum Database. You also create resource queues to manage the workload in the Greenplum Database system. You also implement security measures to tighten access privileges in the Greenplum Database.



In this lab, you create and manage roles for the Greenplum Database. You also create resource queues to manage the workload in the Greenplum Database system. You also implement security measures to tighten access privileges in the Greenplum Database.



This lesson covered the roles and privileges assigned to roles to protect sensitive data from unauthorized roles and groups. Administrators can assign specific privileges to individual roles or group roles, where the privileges are inherited by users within those group roles. Additionally, workload management lets you assign resource profiles to roles. This allows users to share resources, reducing the chances that queries will be starved out of resources.



Listed are the key points covered in this module. You should have learned to:

- Identify and manage database objects available in the Greenplum Database.
- Use data manipulation language and data query language to access, manage, and query data.
- Manage workload management processes by defining and managing roles, privileges, and resource queues.
- Implement system-level and database-level security in the Greenplum Database.