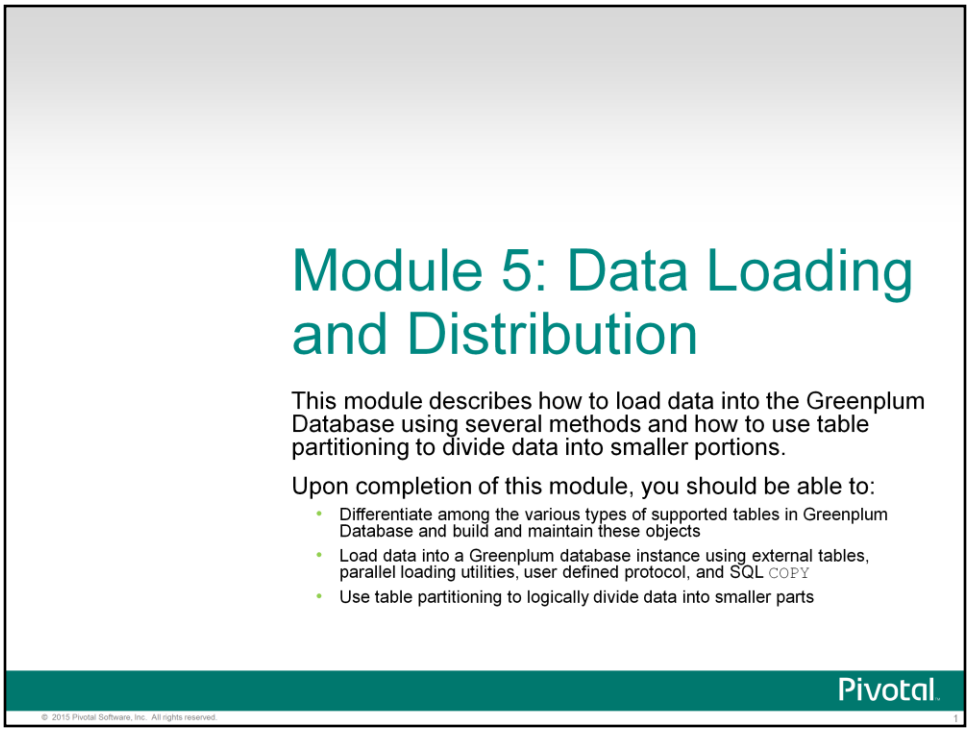
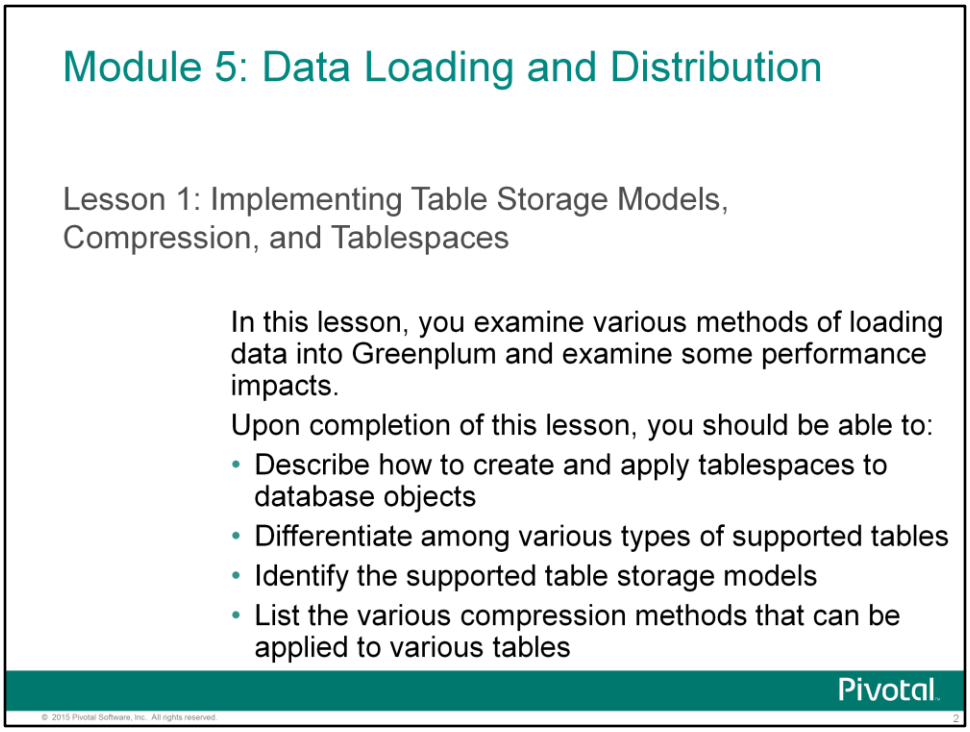


Greenplum Database provides support for several different types of tables, all with specific purposes to support database administrators meet the needs of their consumers. You should be able to differentiate among these tables and determine when to use each of them.

When getting ready to perform data loads, you must evaluate if and how the data should be partitioned and what commands, applications, or tools to use to load the data. Part of the design strategy for organizing your data will be to optimize for query. Table partitioning can help by allowing you to divide very large tables into more manageable portions. You must also understand the impact that the different methods of data loading can have on performance and ease of administration.

In this module, you will:

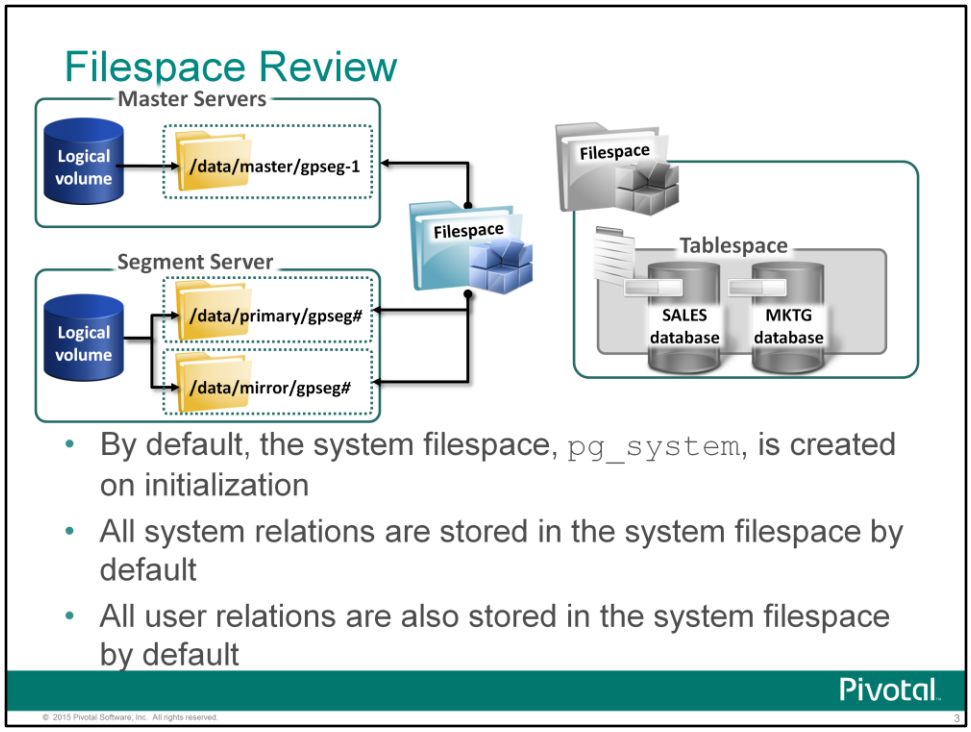
- Differentiate among the various types of supported tables in Greenplum Database and build and maintain these objects
- Load data into a Greenplum database instance using external tables, parallel loading utilities, and SQL `COPY`.
- Use table partitioning to divide tables into smaller parts.



Greenplum offers several ways in which you can further manage user data within their own tablespaces and build appropriate tables to accommodate different types of data needs.

By the end of this lesson, you should be able to:

- Describe how to create and apply tablespaces to your database objects
- Differentiate among the various types of tables supported within Greenplum Database
- Identify the supported table storage models available to Greenplum Database tables
- List the various compression methods that can be applied to tables



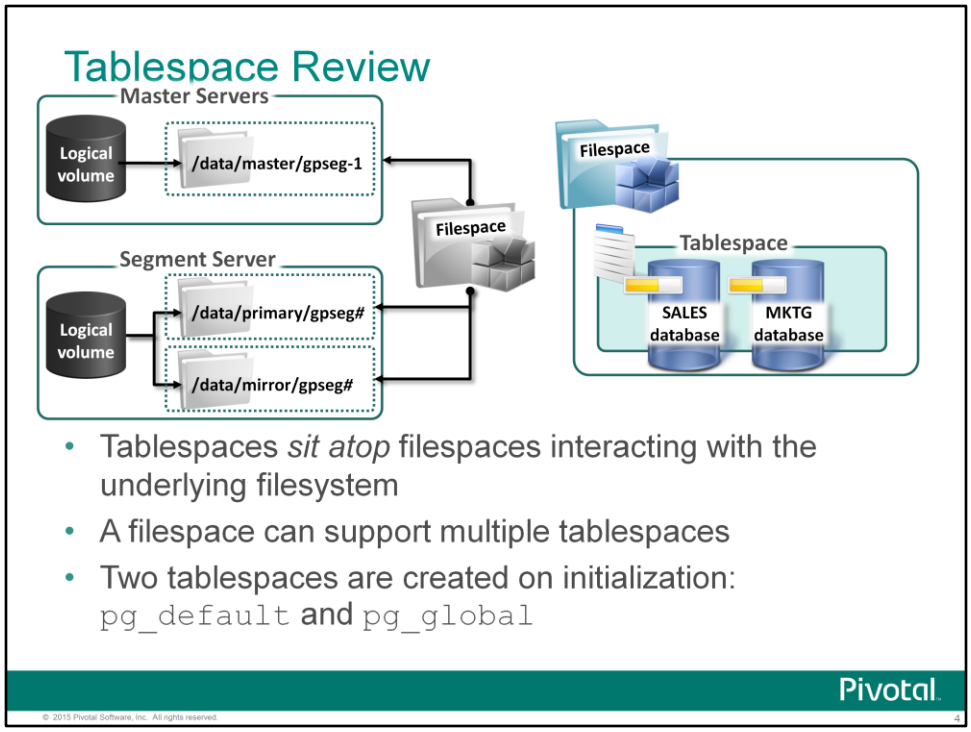
A filespace is a pointer to specific storage defined on your filesystem. The filespace maps to a set of locations used by the master, standby, and segment hosts.

The system filespace, `pg_system`, is automatically created on initialization and used to store system and user relations. In the example shown, the filesystem that supports the filespace is associated with the `/data` directory. Data is stored as follows:

- Master and standby master store files in the `/data/master` directory.
- Primary segments store their files in `/data/primary` while mirror segments store their files in `/data/mirror`.

All tables and other database objects you subsequently create are stored within the system filespace, unless otherwise specified.

Note that the filespace is tied to a location on the filesystem and not necessarily to the filesystem root. You can have multiple filespaces share the same filesystem by pointing to different directories that exist on the filesystem.

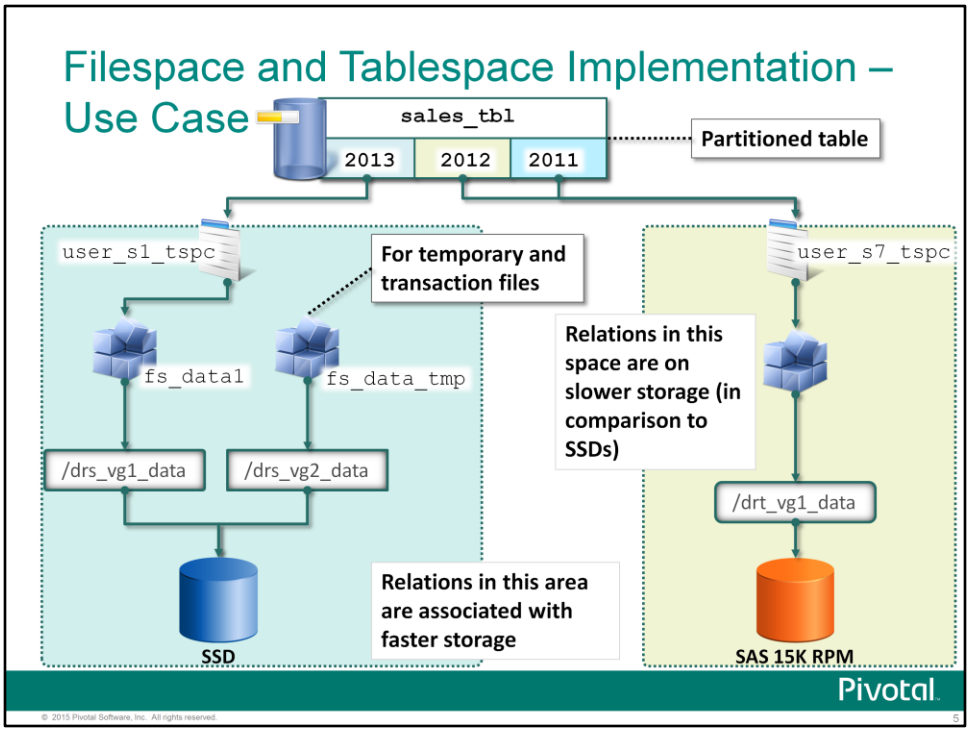


A tablespace interacts with the underlying filesystem by being associated with a specific filespace. A filespace can host multiple tablespaces, but a tablespace can only belong to a system filespace. Tablespaces allow database administrators to effectively segregate data and take advantage of different storage profiles.

Within Greenplum, you can create and assign database objects, such as databases, tables, and indexes, to a specific tablespace.

Two tablespaces are defined by default:

- `pg_default` – This tablespace is the default tablespace used to store the default databases created for the system, template0 and template1, as well as any other user-related database objects.
- `pg_global` – This tablespace is used to store shared system catalog for the environment.

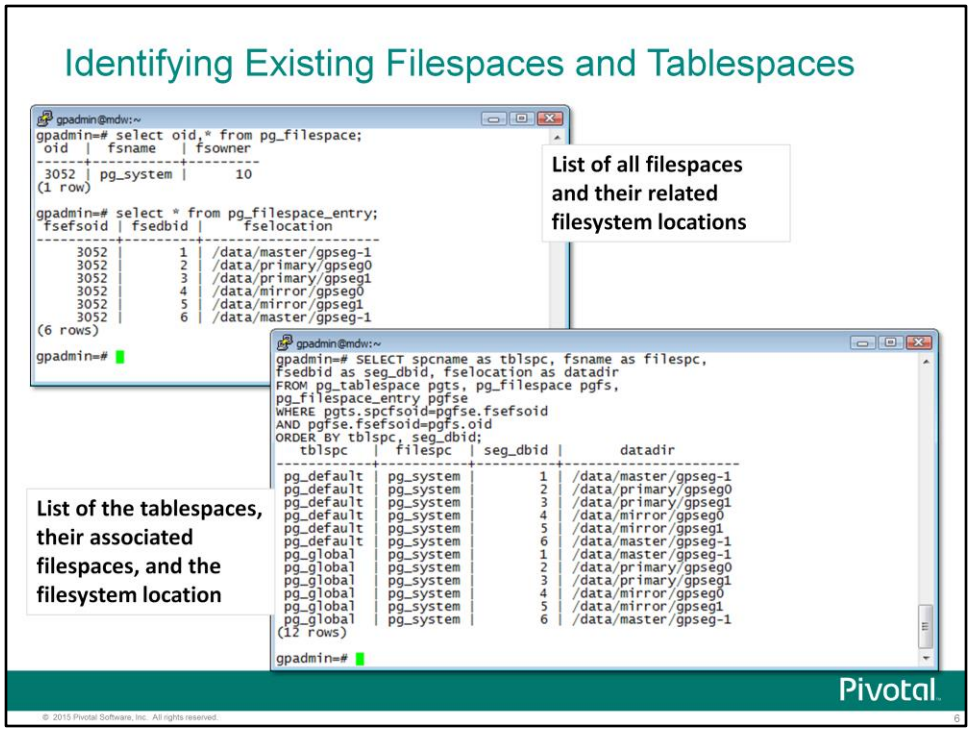


Greenplum is not aware of, nor does it control the underlying filesystem. Instead, it relies on you creating the filesystem and associating the filespace with the filesystem.

You can derive large benefits by maintaining multiple filespaces to meet the needs of customers within the environment. Your requirement may be to provide faster performance on data that is more often used, while relegating older, less-often used data to slower disks. By defining a filesystem on faster SSD disks for example, while defining another filesystem on SAS or SATA drives with lower speeds, you can help to meet the requirements of the service level agreement associated with the requirement.

In this example, partitions of a partitioned table, `sales_tbl`, are placed on separate tablespaces. The table is partitioned on the year field, with the latest year associated with the tablespace, `user_s1_tspc`, which is on the `fs_data1` filespace. This filespace is tied to a specific filesystem on faster SSD drives. The remaining two partitions are on one tablespace, which is associated with a slower disk device. As the partitions for previous years are less often used, they can be moved to tablespaces on slower devices, leaving the partition with newer data on the faster device.

A third filespace in this slide, `fs_data_tmp`, is used to store temporary and transaction files. These files are created when queries are executed, during backup, and when data is stored sequentially. Moving these temporary and transaction files to a filespace on faster storage can help to improve the performance of the database overall. This filespace can be used to store user data. However, you cannot store temporary or transaction data across multiple filesystems – you are restricted to only one.



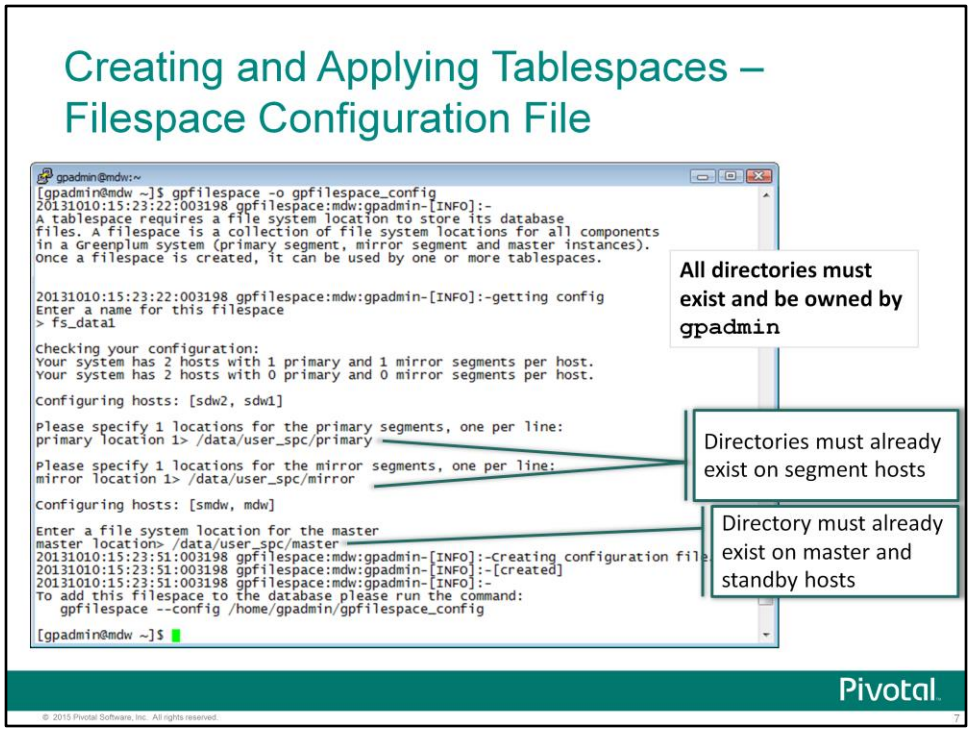
You can determine the filespaces that exist within Greenplum by querying two system tables:

- `pg_filespace` lists the filespaces that exist within the environment.
- `pg_filespace_entry` lists the directories and the filespace database IDs associated with each filespace in the environment.

The OID listed for the `pg_filespace` table can be used to join the `pg_filespace` table to the `pg_filespace_entry` table on the `fsefsoid` column to list all directories associated with a specific filespace.

The second sample shows all of the tablespaces defined in the system, along with their associated filespace and filesystem. The tablespaces are listed within the `pg_tablespace` table. The syntax in the second sample is as follows:

```
SELECT    spcname as tblspc, fsname as filespc,
     fsedbid as seg_dbid, fselocation as datadir
FROM      pg_tablespace pgts, pg_filespace pgfs,
     pg_filespace_entry pgfse
WHERE     pgts.spcfsoid=pgfse.fsefsoid
     AND pgfse.fsefsoid=pgfs.oid
ORDER BY tblspc, seg_dbid;
```



To create a tablespace, you must first create the underlying filesystem and related filespace.

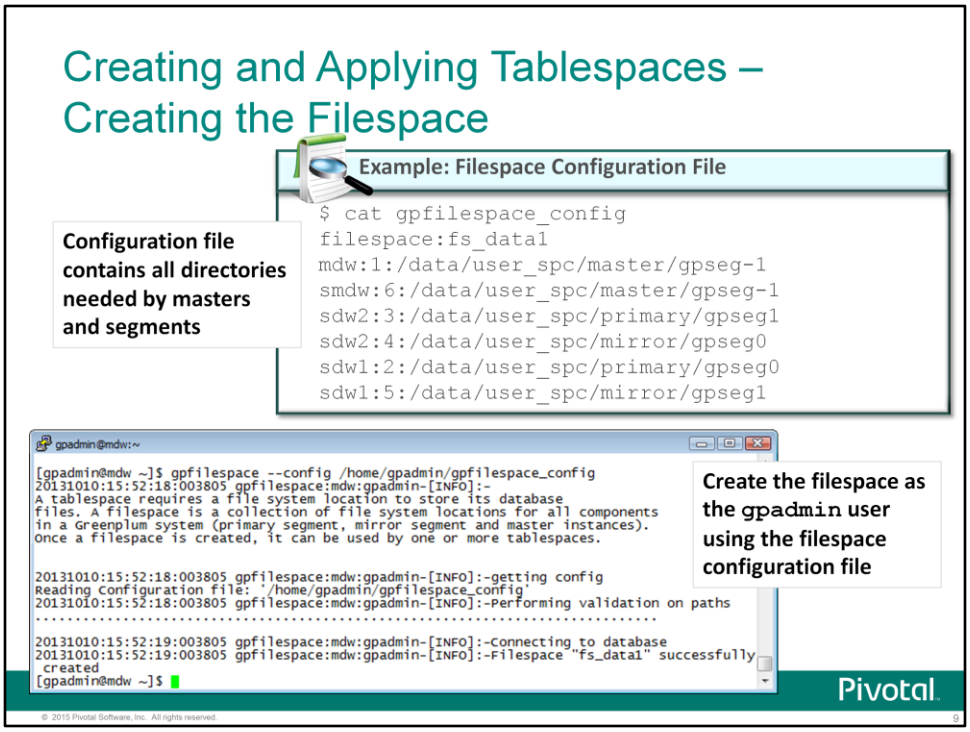
Before creating the filespace:

- Ensure the directory that the filespace will be associated with already exists on the master, standby, and segment hosts. You can use the `gpssh` command to create the directory on all hosts in the cluster. For example, the directory on the master and standby servers would be `/data/user_spc/master`. The directory for the primary segments could be `/data/user_spc/primary`, while the mirror segments would use `/data/user_spc/mirror`. If you have multiple primary and mirror segments per segment host, you will need to create separate directories for each of these segments.
- The directories you create must be owned and be writable by the `gpadmin` user.

Creating and Applying Tablespaces – Filespace Configuration File (Continued)

To create the filespace:

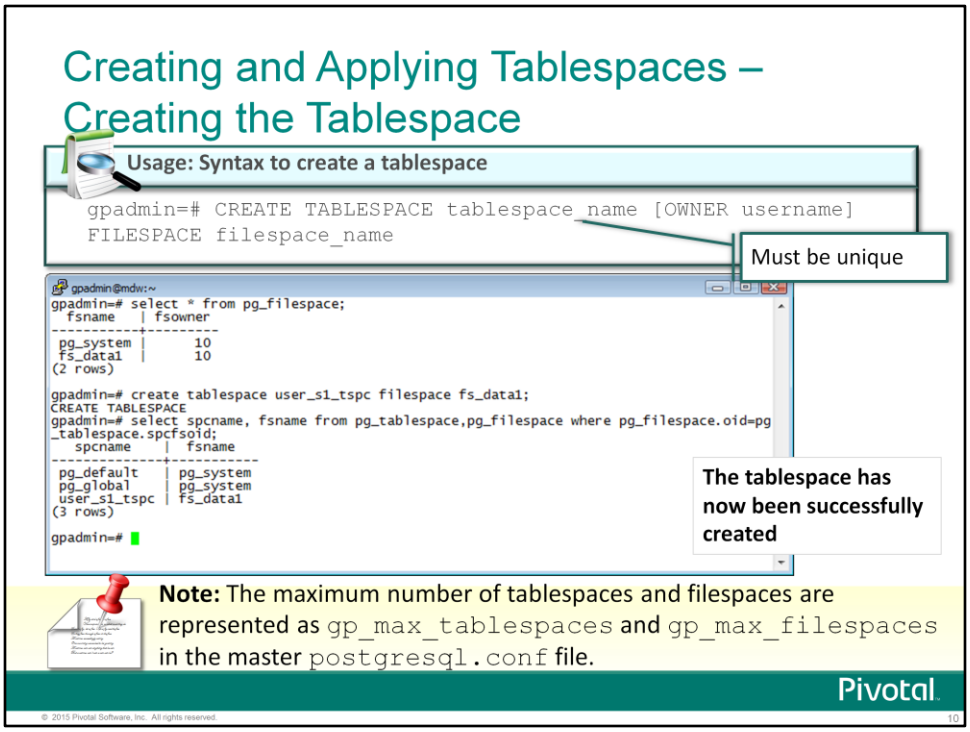
1. As a database superuser, use the command, `gpfilespace -o` *`config_file`*, to create the configuration file that will eventually be used to create the filespace, where *`config_file`* is the file to be created.
2. You will be prompted for each primary segment directory and mirror segment directory on all segment hosts. If your cluster hosts multiple primary segments per host, you will need to define a separate directory for each of these primary and their corresponding mirror segments.
3. You will be prompted for the master and standby master directory. This directory is the same on both systems, so this is entered only once.



Once the filespace configuration file has been created, you can use it to create the filespace. The configuration file contains the information you entered during the file configuration step. The directories for the master, standby, primary, and mirror segments are all documented in the file.

To create the filespace, as the gpadmin user, issue the command, `gpfilespace --config` *`config_file`*,

where *`config_file`* represents the file that contains the filespace configuration.



Creating the tablespace defines an in-database relationship between the tablespace and the filespace you created. The tablespace name must be distinct from other tablespaces in the database.

To create the tablespace, enter the following SQL command as a database superuser:

`CREATE TABLESPACE` *`tablespace_name`* `FILESPACE` *`filespace_name`*

As the database superuser, you can assign the tablespace to a non-superuser during the creation process or afterwards with the `ALTER TABLESPACE` command. Note that the tablespace name cannot begin with `gp_` or `pg_` as these prefixes are reserved for system tablespaces.

Note that by default, the maximum number of tablespaces you can define in the Greenplum Database system is 16, while the maximum number of filespaces is 8. These values can be adjusted in the `postgresql.conf` file by changing the following values:

- `gp_max_tablespaces` – Sets the maximum number of tablespaces in the system
- `gp_max_filespaces` – Sets the maximum number of filespaces in the system

The values for these parameters are included in the calculation of space reserved for internal structures for the database. Exceeding these values makes your environment vulnerable to out-of-shared-memory issues. Increasing these values assumes you have enough memory to accommodate the new values.

Creating and Applying Tablespaces – Applying the Tablespace

Object	Example
Database	`CREATE DATABASE tt_db TABLESPACE user_s1_tspc;`
Table	`CREATE TABLE tt_rt (id int) TABLESPACE user_s1_tspc;`
Partitioned Table	`CREATE TABLE ttct2_part_rt (id int, id2 int) PARTITION BY LIST (id) ( PARTITION one VALUES (1), PARTITION two VALUES (2) TABLESPACE user_s1_tspc, PARTITION three VALUES(3) );`
Index	`CREATE INDEX tt_idx ON tt_rt (id) TABLESPACE user_s1_tspc;`

Note: The default tablespace for the environment can be set by modifying the `default_tablespace` parameter in the master server's `postgresql.conf` file.

Pivotal

11

When creating a database, table, or index, you can specify the tablespace to create the database object in. When you specify the tablespace for a database, all objects created in that database will be created in the same tablespace.

When you do not specify the tablespace on creating an object, the default tablespace for the database is used. You can change the default tablespace for users in Greenplum by modifying the `default_tablespace` parameter in the `postgresql.conf` file on the master server.

Altering and Removing Tablespaces and Filespaces

Action	Syntax
Changing the tablespace	`ALTER TABLESPACE name RENAME TO newname` `ALTER TABLESPACE name OWNER TO newowner`
Removing a tablespace	`DROP TABLESPACE [IF EXISTS] tablespacename`
Changing a filespace	`ALTER FILESPACE name RENAME TO newname` `ALTER FILESPACE name OWNER TO newowner`
Removing a filespace	`DROP FILESPACE [IF EXISTS] filespacename`

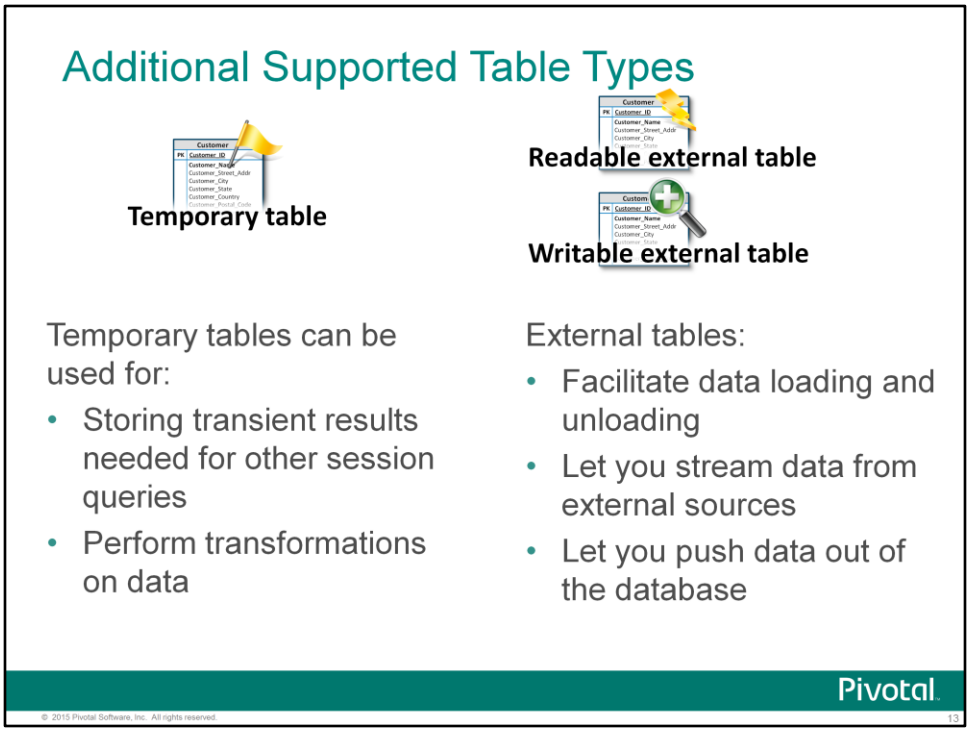
Note: You cannot drop a tablespace or filespace until all associated objects are removed from the tablespace and all tablespaces using the specified filespace are removed.

Pivotal

You can change and remove tablespaces if you are the owner, a member of a role that owns the tablespace, or are a database superuser. You can change the name of the tablespace or reassign it to a new owner using the `ALTER TABLESPACE` SQL command. You can remove the tablespace using the `DROP TABLESPACE` SQL command.

The same logic applies to filespaces, where you can change and remove a filespace if you are the owner, a member of a role that owns the filespace, or a database superuser. You can change the name or owner of the filespace using the `ALTER FILESPACE` command and drop the filespace using the `DROP FILESPACE` command.

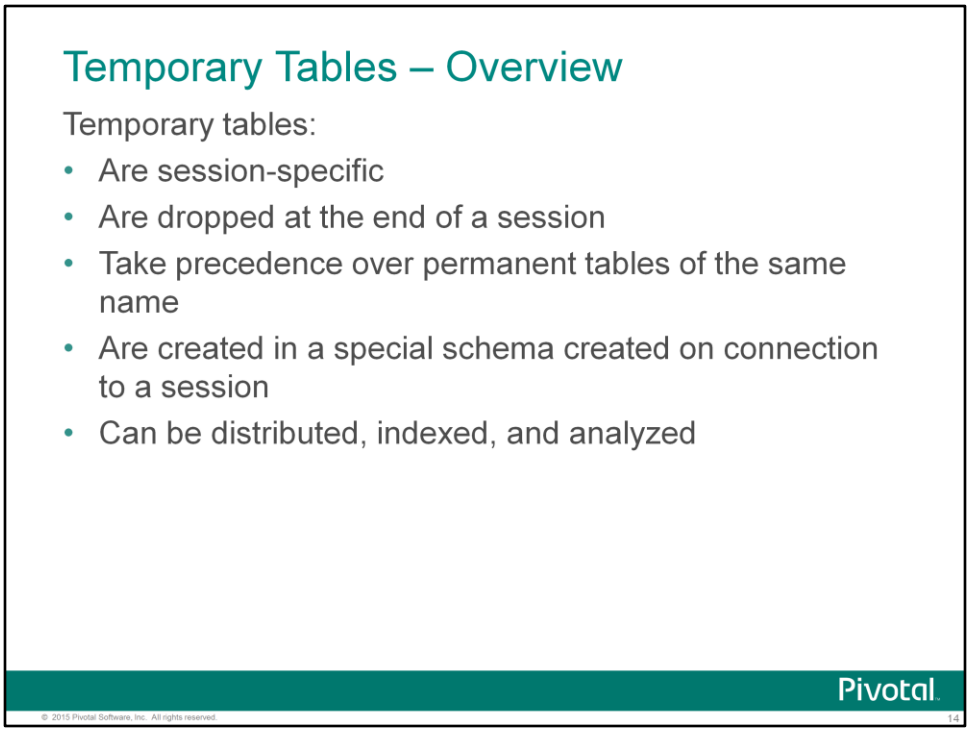
Before you can drop a filespace, you must remove all tablespaces associated with the filespace. Additionally, if the filespace is being used for temporary or transaction files, it cannot be dropped.



Greenplum Database provides support for a variety of tables to facilitate data processing and data loading or unloading. In addition to regular tables, the following table types are supported:

- Temporary tables – These are often used to store the results of queries that are required for a short period of time, during a single connection session, or for transforming data during an ELT process.
- External tables – External tables are used to facilitate data loading and unloading to and from the Greenplum Database. It can be useful in a variety of applications, including to retrieve a stream of data from an external source, such as from a website of operating system command. It can also be used to push data between databases.

External tables have previously been discussed in this course. More details on external table usage and definition will be provided later in the course.



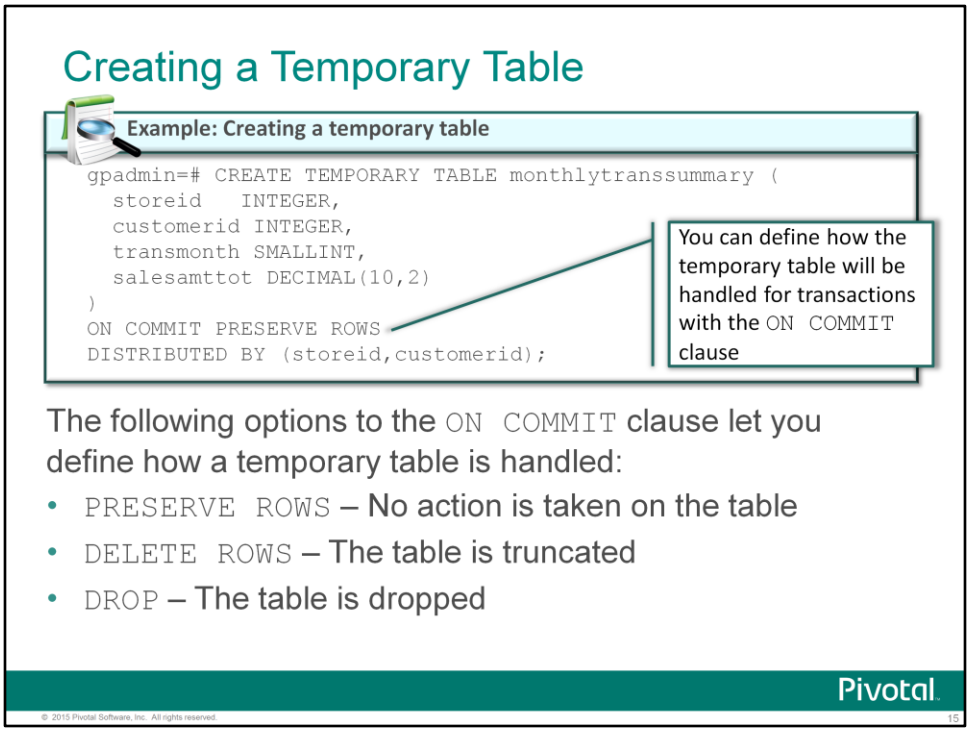
Temporary tables in Greenplum Database are session specific, in that each session must create its own temporary table. A temporary table is not shared across sessions and is automatically dropped at the end of the session in which it is created. Any indexes created against the temporary table are also considered temporary and are dropped at the end of the session. As temporary tables are session-specific, you cannot share data across sessions.

If you are not referencing tables by their schema-qualified names, a temporary table takes precedence over existing permanent tables using the same name. Temporary tables are automatically created in a special schema and cannot be assigned to a schema of your choosing. These schemas are automatically created when you connect to a session.

Temporary tables behave like permanent tables in that:

- You can specify a distribution key for a temporary table.
- You can create indexes against columns of a temporary table. These indexes are themselves temporary.
- Temporary tables can be analyzed, allowing the optimizer to generate an execution plan for the table.

They differ from permanent tables in that they are not added to the system catalog and therefore do not impact the size and performance of the system catalog.

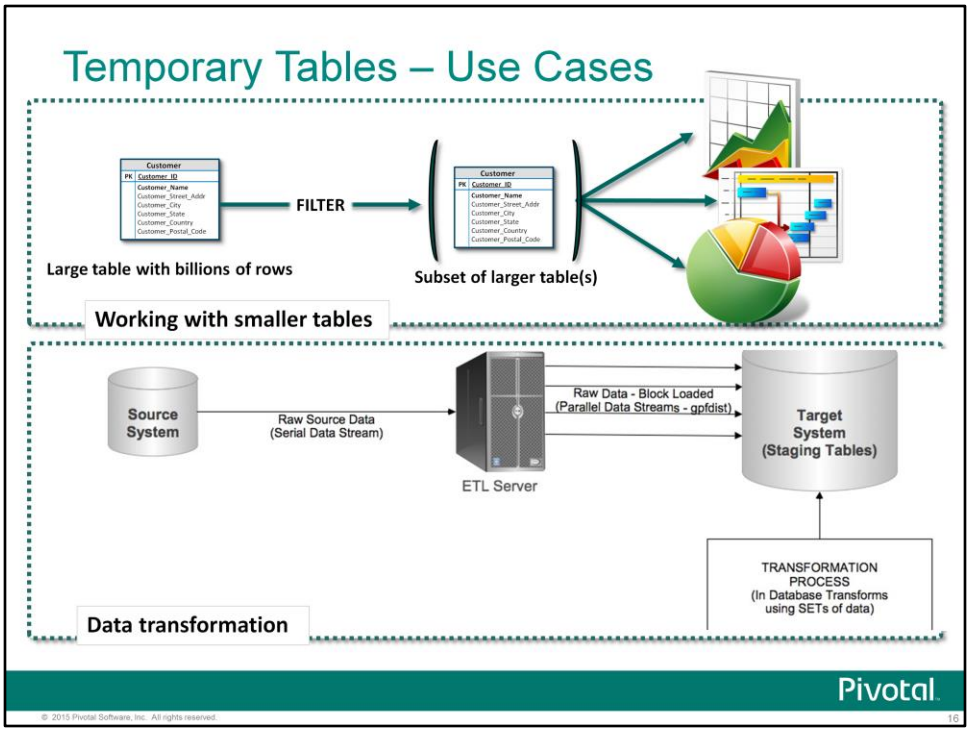


The syntax to create a temporary table is `CREATE TEMPORARY TABLE` or `CREATE TEMP TABLE`.

In this example, the temporary table, `monthlytranssumarry` is created with the distribution key on two columns: `storeid` and `customerid`. The rows of the temporary table will be preserved, or saved, at the end of the transaction, ensuring that the rows are not removed until the session has ended.

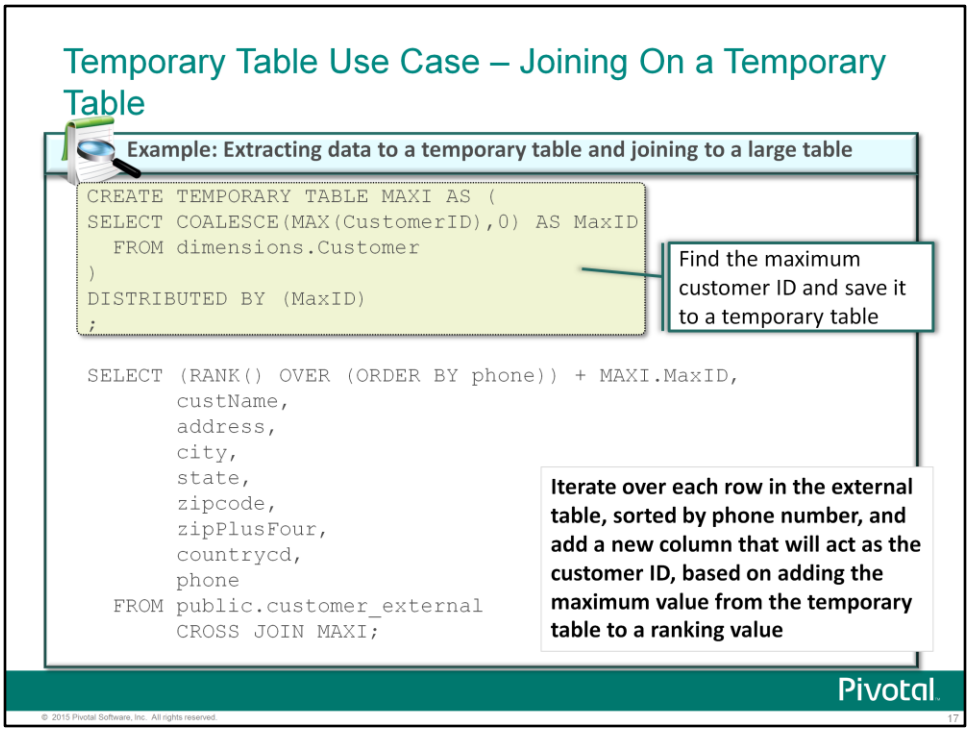
There are three `ON COMMIT` states that you can pass to the temporary table are:

- `PRESERVE ROWS` – The rows within the table are saved from removal at the end of a transaction block. This translates into no action being taken on the temporary table or its data.
- `DELETE ROWS` – The temporary table is truncated at the end of the transaction block, so all rows have been removed.
- `DROP` – The temporary table is dropped at the end of the transaction block.

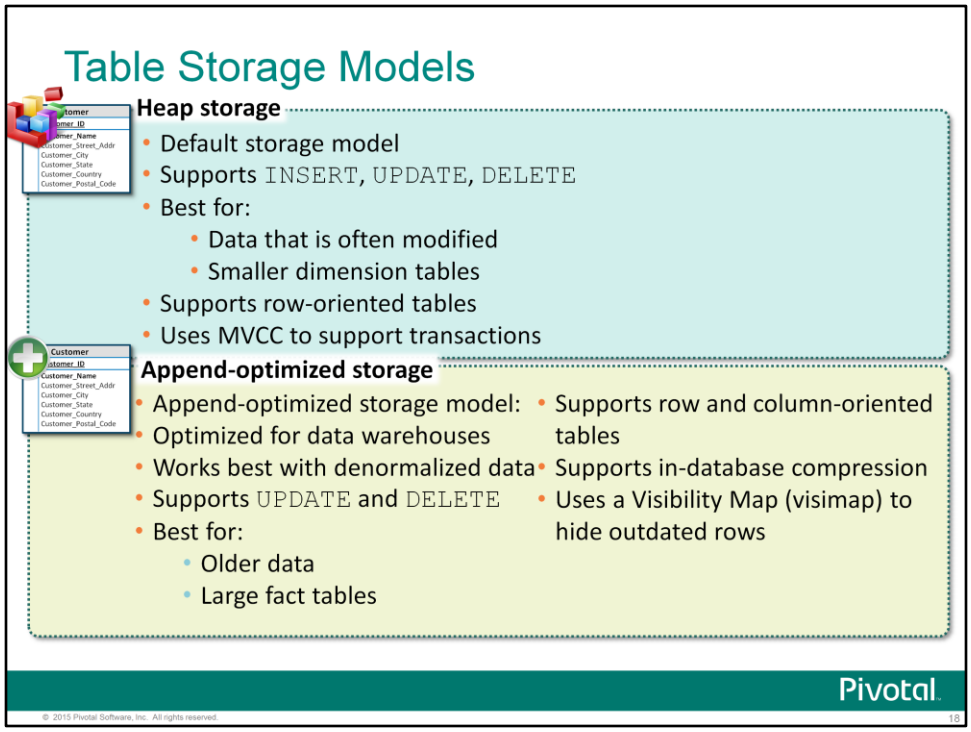


The most common use of a temporary table is to take a subset of data from a larger table, reorganize it, and use the results for reporting. The intention is to limit the number of rows you are working with to facilitate reporting. This improves overall performance as you are now working on a subset of data instead of a very large dataset. When working with large tables, it is best to avoid performing a union on a large number of queries against large tables. Instead, if you can reduce the size of your table to just what is needed, you can gain significant performance in generating results. You also reduce the chance of running out of memory when attempting to work with a large amount of rows in memory because the query optimizer will attempt to launch each query in parallel.

Another use case for working with temporary tables is for data transformation. With the extract, load, and transform method, you load data directly into the database and then perform your transformation on the data. By loading the data into the database and performing the transform, you take advantage of the Greenplum MPP architecture to improve performance in an area that is often the slowest part of data migration.



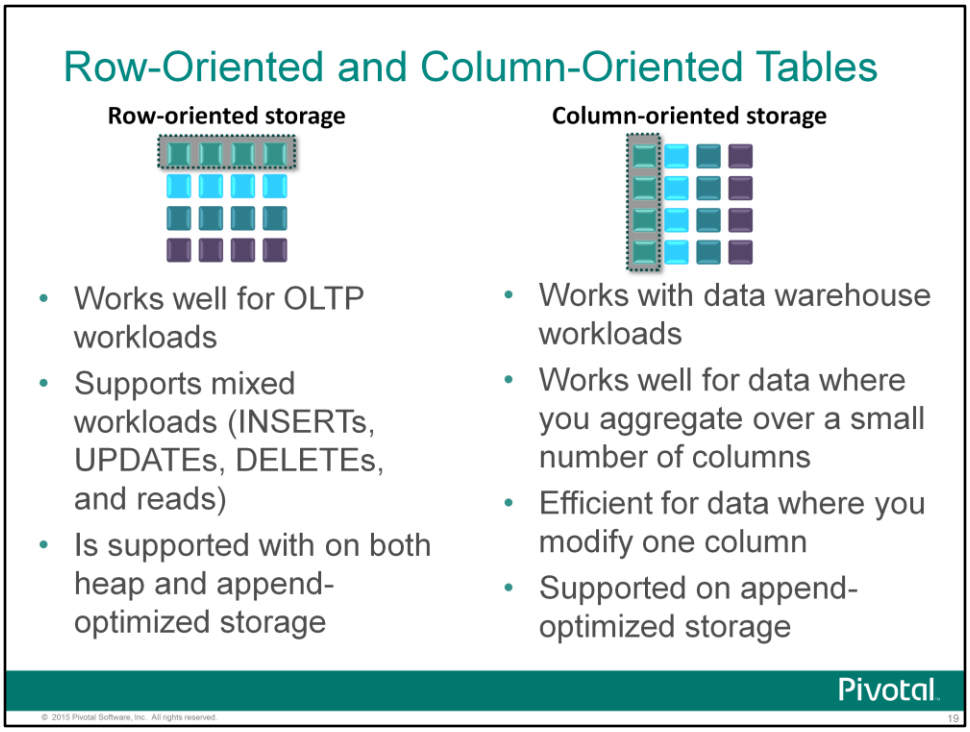
In the example shown, the maximum non-null customer ID is extracted from the table containing customer information. The results are saved to a temporary table, `maxi`. In the next query statement, data is extracted from an external table which contains additional customer information. The information is ordered by the customer's phone number and a rank is assigned to each customer. The rank will increment with each row line. This value is added to the results of the temporary table. The final result is that each customer can be assigned a customer ID starting with the maximum customer ID that is in the `dimensions.customer` table.



Greenplum provides support for two table storage models with various available options. These storage models used are.

- Heap storage model – Unless otherwise specified, Greenplum tables are created using the heap storage model. Heap tables allow you to add new rows, update existing rows, and remove rows from the table. These types of transactions are seen in OLTP databases where data is often modified after it has been loaded.
- Append-optimized storage model – An append-optimized table is optimized for data warehouses in that it works best with denormalized data. If you follow the strictest definition of the data warehouse, data is not be normally modified once it has been loaded to its permanent table. Append-optimized tables afford some flexibility in that the data can be refreshed, allowing updates and deletes. Append-optimized tables work well for large batch uploads instead of single row inserts as would be seen in OLTP-type transactions. Large tables, where data does not normally require refreshing, such as older data, or large fact tables, are ideal use cases for append-optimized tables.
  Append-optimized storage does not work with transactions that use serializable isolation levels, as would be seen with row-level inserts, and updatable cursors. Append-only storage uses a visibility map to mark the rows which should be visible and those that should not.

Both storage models support row-oriented tables, whereas column-oriented tables are only supported with append-optimized storage. Append-optimized storage also supports in-database compression at the table and columnar levels.



In addition to heap and append-optimized storage models, you can also define your table to be row-oriented or column-oriented.

A row-oriented table meets the needs of OLTP workloads, where you are accessing row and columnar data. It supports mixed workloads where you may perform inserts, updates, deletes, and reads quite frequently.

A column-oriented table, as with append-optimized storage, performs well for data warehouse workloads, where you may oftentimes be computing aggregations of data over a small number of columns. This type of storage also works well for cases where you need to modify very few columns without modifying other columnar data.

Creating Heap and Append-Optimized Tables

Action	Example
Creating a heap, row-oriented table	`CREATE TABLE tc_heap (id int, descr text) DISTRIBUTED BY (id);`
Creating an append-optimized, row-oriented table	`CREATE TABLE tc_ao (id int, sales float) WITH (appendonly=true) DISTRIBUTED BY (id);`
Creating an append-optimized, column-oriented table	`CREATE TABLE tc_ao_c (id int, sales float) WITH (appendonly=true, orientation=column) DISTRIBUTED BY (id);`

Note: You cannot modify the storage or orientation of a table once defined. You can create a new table with the desired options and migrate your data.

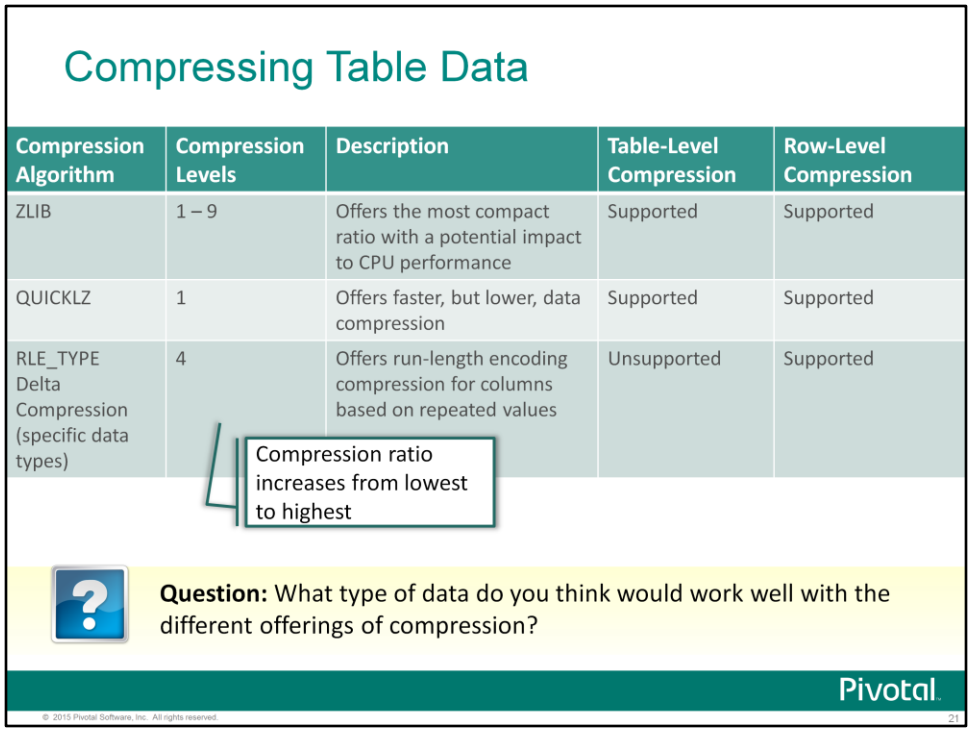
Pivotal

Unless otherwise specified, the default behavior when creating a table is to create a heap storage table with row orientation.

To create an append-optimized table, you must include the clause `WITH (appendonly=true)` in your table definition.

To specify column orientation, use the `WITH (appendonly=true, orientation=column)` clause in your table definition.

You cannot change the storage model or orientation of an existing table. You can instead create a new table with the desired storage model and orientation and migrate your data into that table.



Append-optimized tables support compression at two levels:

- Table-level compression is applied to the entire table with support for ZLIB and QuickLZ compression algorithms.
- Column-level compression is applied to columns within a table. Each column can utilize a different compression algorithm. Supported algorithms at the column-level include RLE_TYPE, ZLIB, and QuickLZ.

Note that there is an order of precedence. Compression at the subpartition level of a table overrides all other compression defined for the table. The order of precedence from lowest to highest is table, column, partition, and subpartition.

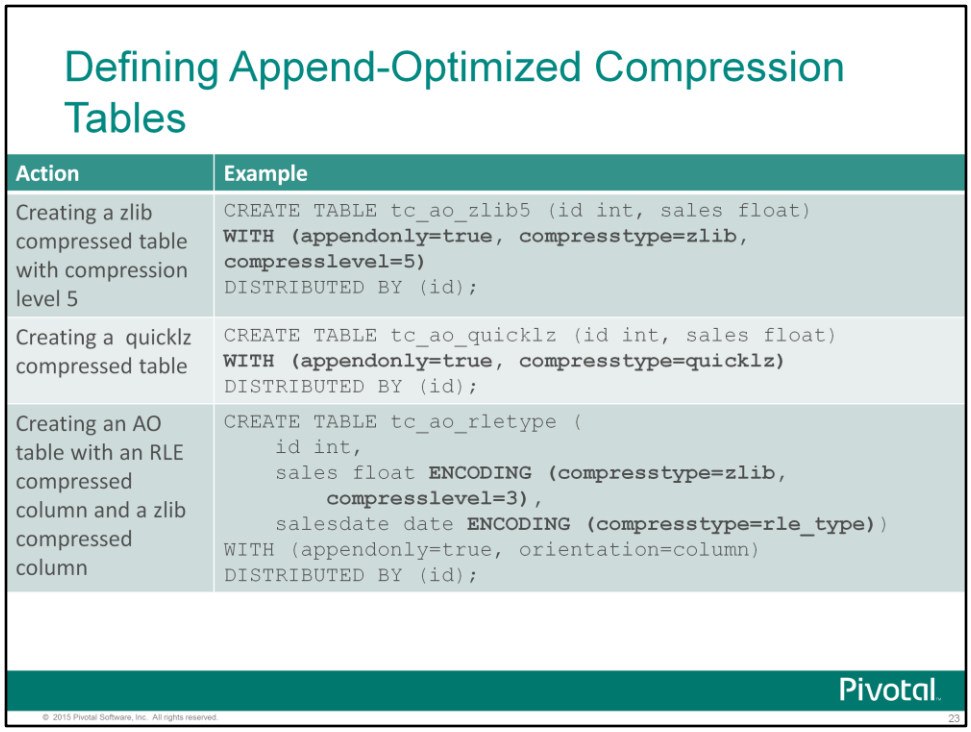
The various algorithms offer different performance and compression levels that may lend itself to some environments and data more than others.

- ZLIB is the default algorithm that offers greater compression, but at lower speeds. At compression level 1, zlib offers the fastest compression with the lowest ration, while at 9, it offers the most compact compression at the slowest speed.
- QuickLZ uses less CPU cycles and compresses data faster at a lower compression ratio than zlib, but offers only one compression level.
- RLE_TYPE offers run-length encoding (RLE) for column-level compression. With RLE, repeated data is stored as a single value along with a count of the number of times that value is found. As of GPDB 4.3.3, in addition to RLE_TYPE, Delta compression is also applied for columns that are defined as `BITINT`, `INTEGER`, `DATE`, `TIME`, or `TIMESTAMP`.

Compressing Table Data

Compression can be applied to data to preserve space within the cluster. Choosing the appropriate compression algorithm and ratio can impact overall performance in the following ways:

- Data compression can help improve I/O performance by saving disk space. As your disk drives reach capacity, you could potentially see a decrease in performance.
- You can also negatively affect performance on some data. You must weigh the compression ratio against storage savings as compressed data may require more CPU cycles to unpack and retrieve.



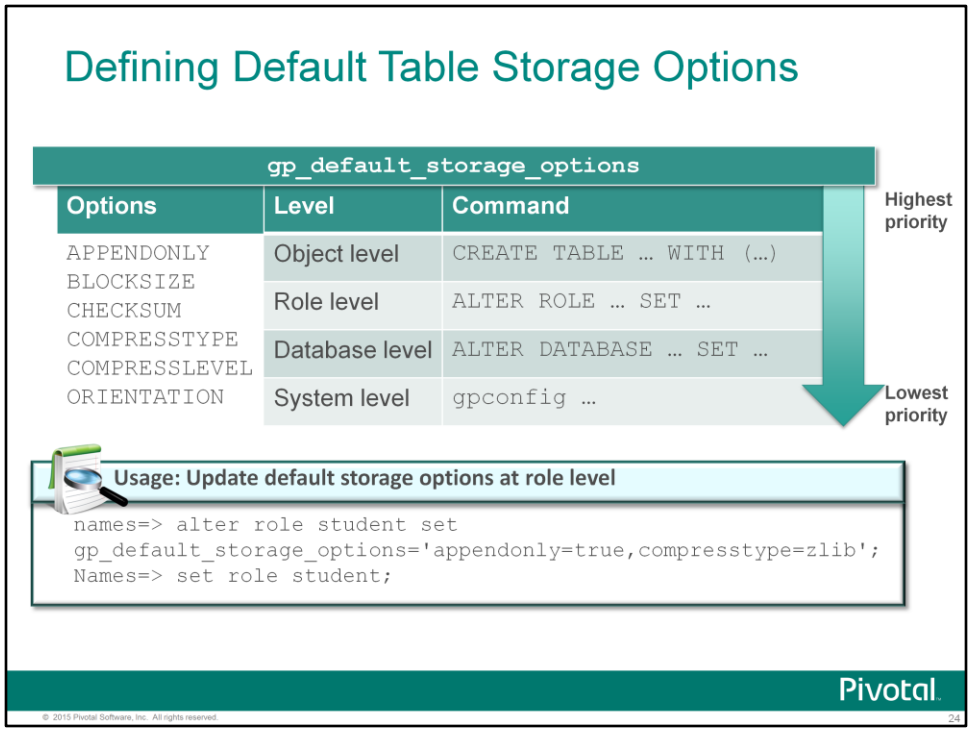
When applying compression to the table, you include the compression type and level in the `WITH` clause of your table definition.

In the first example, the zlib compression algorithm is applied to the table with the clause, `WITH (appendonly=true, compresstype=zlib, compresslevel=5)`. The default compression level is `1` if the `compresslevel` option is not included in the clause.

The second example shows how to define a QuickLZ compressed table by including the `compresstype=quicklz` option in the `WITH` clause.

In the third example, compression algorithms are applied to two columns. Column compression requires the `ENCODING` clause as part of the column definition before you can define the storage specification. The `sales` column will use `zlib` compression at compression level `3`, while the `salesdate` column will use `rle_type` compression.

Note that when you apply compression, it is best to apply it to the lowest level, where the data resides. If only one column requires compression, apply the compression to the column.



The server configuration parameter, `gp_default_storage_options`, lets you define the storage options to automatically apply when creating a table with the `CREATE TABLE` command. Options specified as part of the `CREATE TABLE` command automatically override any settings called for in the parameter.

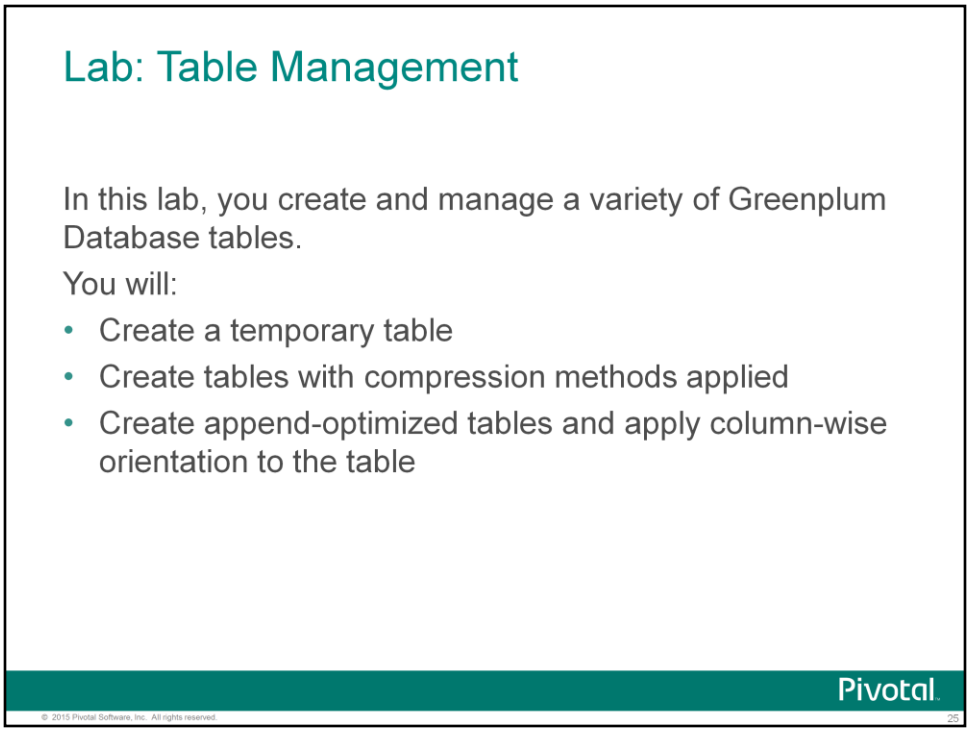
The options are a comma separated list, consisting of the storage options, `APPENDONLY`, `BLOCKSIZE`, `CHECKSUM`, `COMPRESSTYPE`, `COMPRESSLEVEL`, and `ORIENTATION`.

The default behavior can be set at various levels of the database:

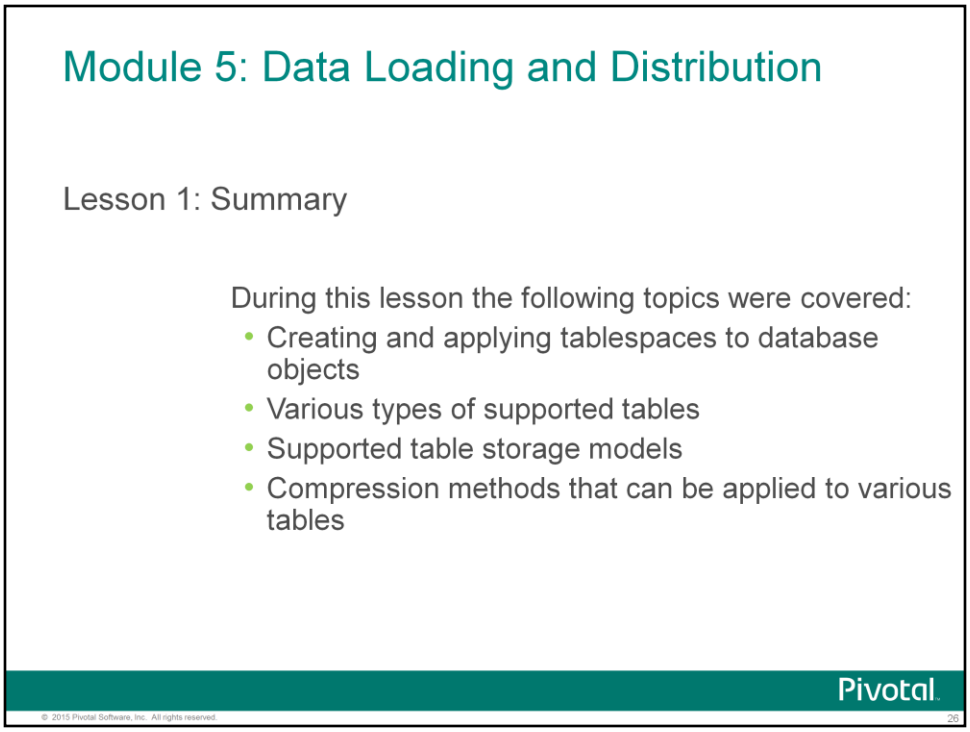
- Role level using the `ALTER ROLE SET …` command
- Database level with the `ALTER DATABASE SET …` command.
- Global level using the `gpconfig` command.

Specifying options using the `CREATE TABLE` takes precedence over all other forms. This is done on a one-on-one basis. Next order of precedence is the role level, followed by the database level, and then the system level. The options are not cumulative across the various levels. Only the options specified at each level will be used if not specified at a higher level.

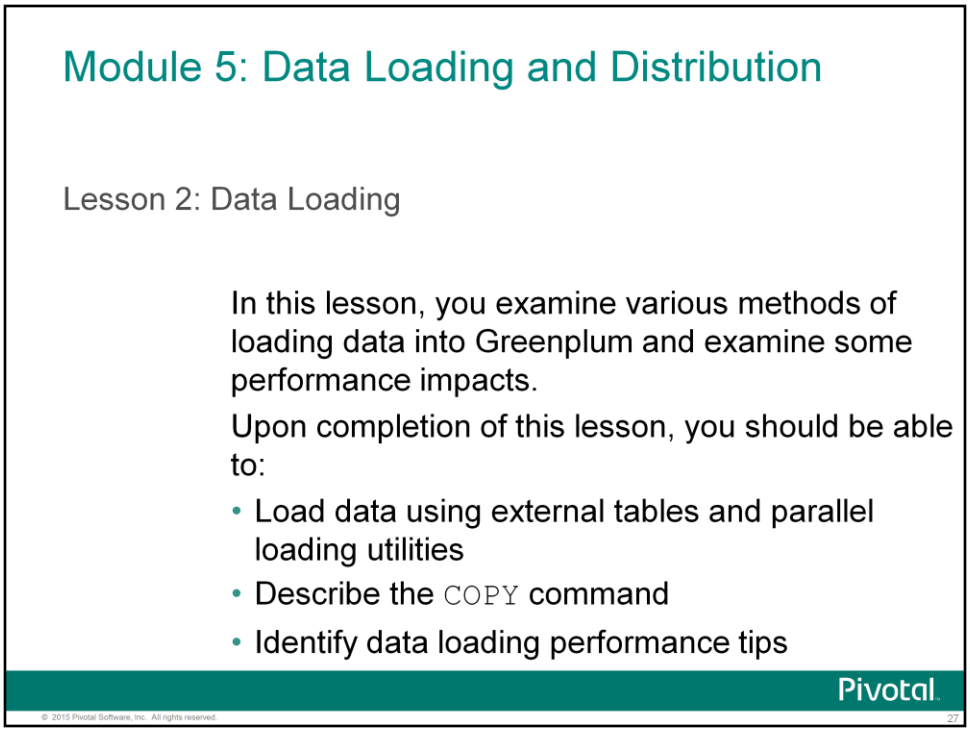
In the example shown, the student account alters the `gp_default_storage_options` parameter with the options, `appendonly=true` and `compresstype=zlib`. After reconnecting to the database as the same account, any tables created by that user will adhere to the default options specified.



In this lab, you create a variety of tables supported by Greenplum Database, including temporary, column-wise store, and append-optimized tables. You also apply compression methods to tables you create.



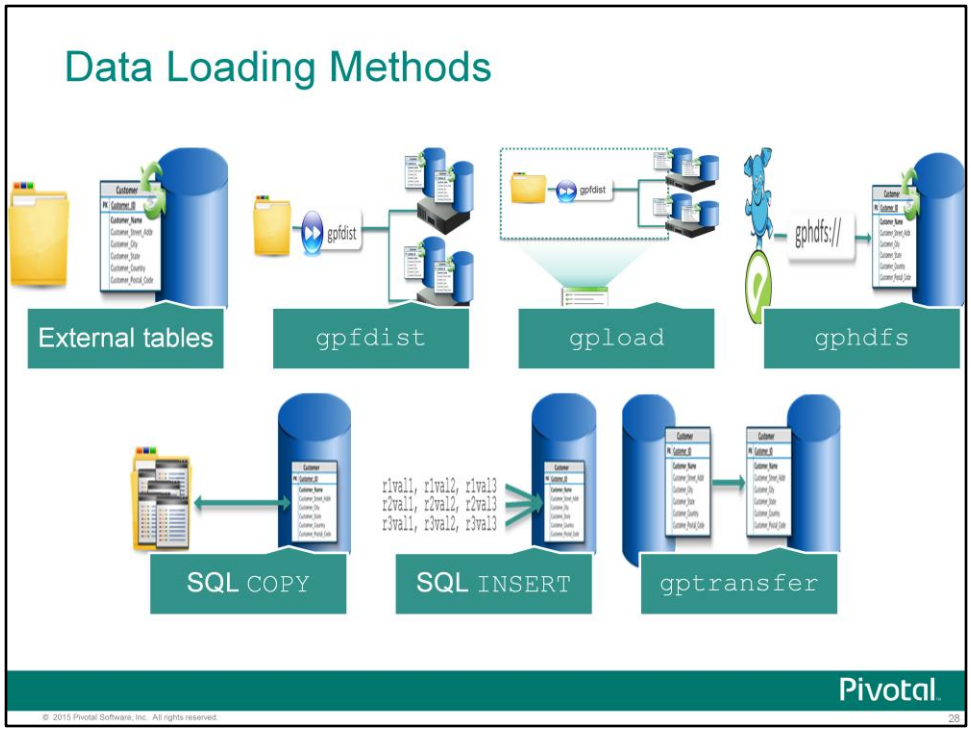
This lesson covered the storage table models supported in Greenplum Database. The various storage models, supported tables, and compression methods are discussed in this model, including how to create the various table types and apply the different compression methods.



There are various methods of loading data into Greenplum Database. Greenplum supports fast, parallel data loading for large amounts of data. This method can be used for ETL processing.

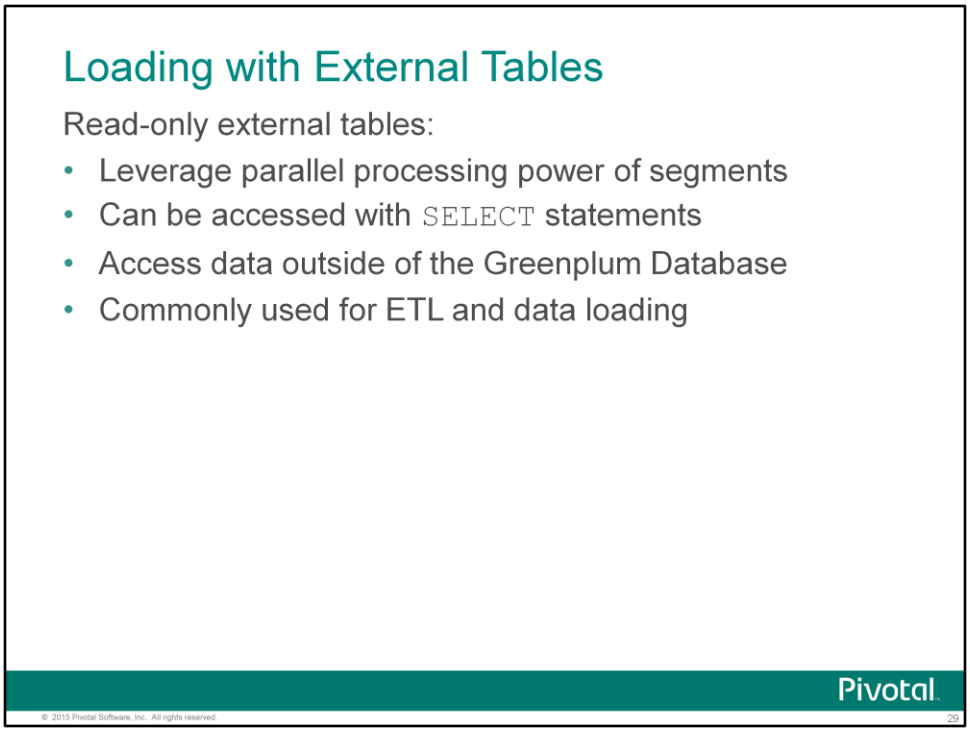
Upon completion of this lesson, you should be able to:

- Load data into Greenplum using external tables and parallel loading utilities.
- Describe the `COPY` command and how it can be used to load data.
- Identify data loading performance tips that can impact you during and after data loads.

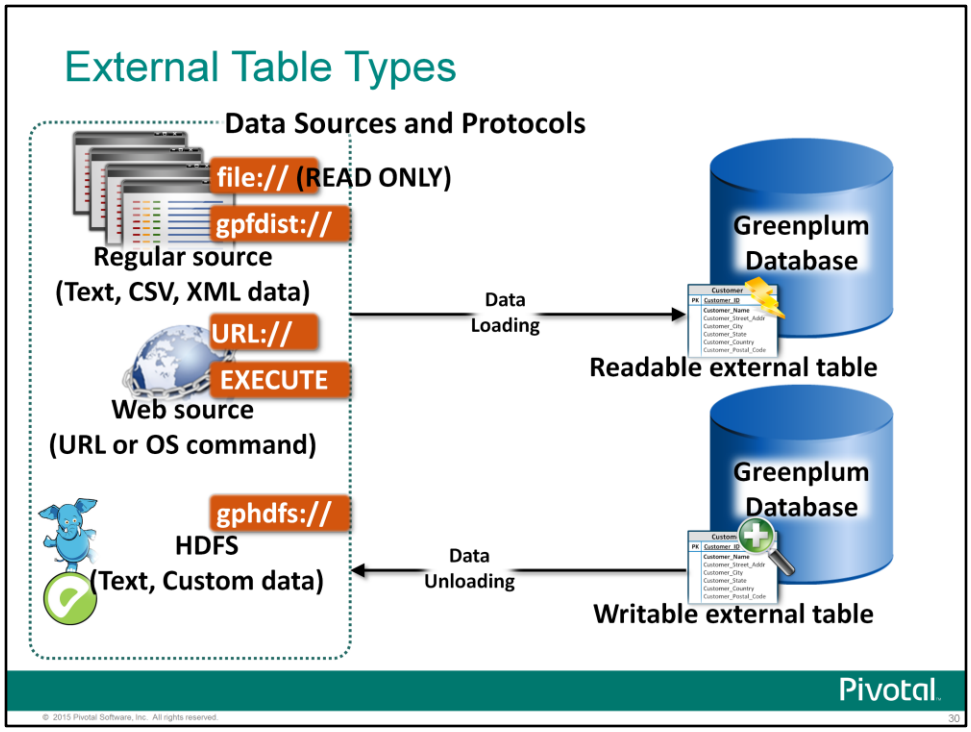


Greenplum supports various methods for loading data into Greenplum Database. These include:

- External tables for data loading and unloading.
- The `gpfdist` utility that is used in conjunction with external tables to provide parallel loads and unloads.
- The `gpload` data loading utility which acts as an interface to the external table parallel loading feature.
- The `gphdfs` protocol acts as an interface between Greenplum Database and the Hadoop filesystem.
- `COPY` command to load or unload data in a non-parallel manner.
- The use of the SQL `INSERT` command to insert data into tables. This method should not be considered for bulk loads of large data volumes. It can be used with external tables to load data into Greenplum Database tables. This option will be discussed later in the course.
- Migrating data from one Greenplum Database system or database to another system or database using the `gptransfer` utility.



External tables leverage the parallel processing power of the segments for data loading. Unlike other loading mechanisms, you can access multiple data sources with one `SELECT` of an external table. The data resides outside of the Greenplum Database. There is a lot of flexibility when defining an external table as to where, how, and what data will be used. External tables are commonly used for ETL and data loading into the Greenplum Database.



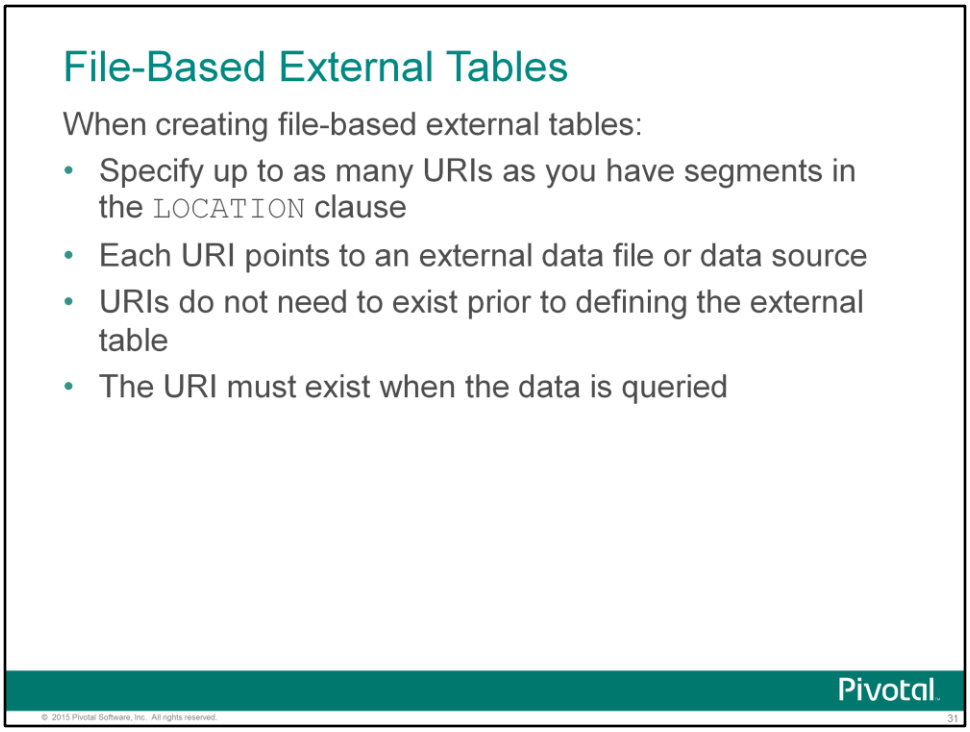
Regular external tables:

- Access static data and is rescannable.
- Uses the `file://` or `gpfdist://` protocols. `gpfdist` is a file server program that serves files in parallel.

Web tables use the `http://` protocol or `EXECUTE` clause to execute an operating system command or script. Data is assumed to be dynamic, meaning that query plans involving web tables do not allow rescanning as the data could change during the course of query execution. This can yield slower query plans as the data must be materialized to the I/O if cannot fit in memory.

Hadoop-based data access data over Hadoop file systems using the `gphdfs` protocol. Data can be text or a user-defined format.

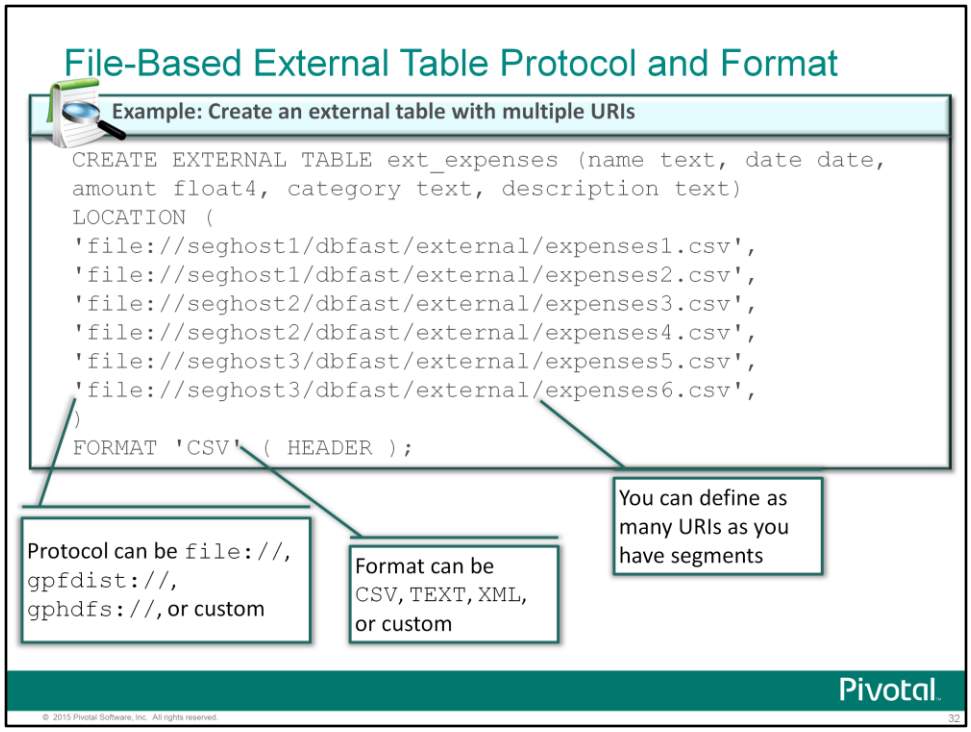
There is one exception to data unloading: you cannot use the `file://` protocol to perform this action. Any of the others are acceptable.



File-based external tables allow you to access flat files as if they were database tables.

When defining file-based external tables:

- You can specify multiple external data sources or URIs (uniform resource identifiers) with the `LOCATION` clause, up to the number of primary segment instances in your Greenplum Database array.
- Each URI points to an external data file or data source.
- The URIs do not need to exist prior to defining an external table.
- The `CREATE EXTERNAL TABLE` command does not validate the URIs specified. You will get an error if the URI cannot be found when querying the external table.



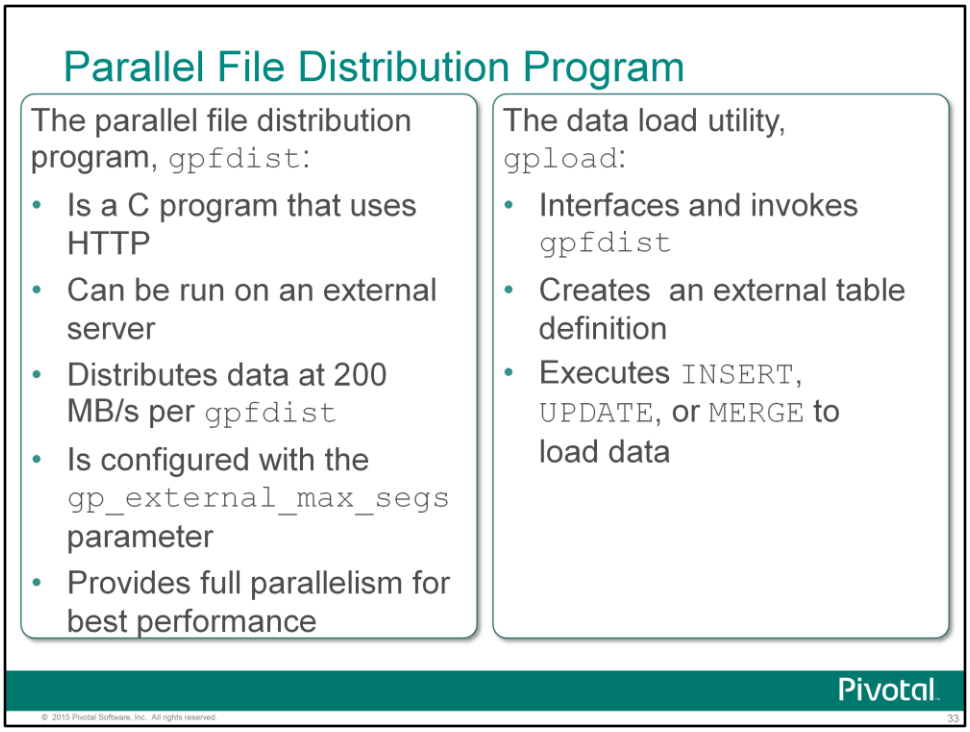
You may use only one of the following protocols for each external table you define:

- **`file://`** — If using the `file://` protocol, the external data file or files must reside on a segment host in a location accessible by the Greenplum super user. You can specify up to the number of available segments in your Greenplum Database system. If you have a Greenplum Database system with 8 primary segments and you specify 2 external files, only 2 of the 8 segments will access the external table in parallel at query time. The number of external files per segment host cannot exceed the number of primary segment instances on that host. The host name used in the URI must match the segment host name as registered in the `gp_configuration` system catalog table.
- **`gpfdist://`** — This protocol retrieves the files using the Greenplum file distribution program.
  NOTE: If you use CSV formatted files with the `gpfdist` protocol and your external data files have a header row, do not declare `HEADER` when creating the external table definition. Instead, use the `-h` option when you start the `gpfdist` program.

The `FORMAT` clause:

- Is used to describe how the external table files are formatted.
- Accepts plain text (`TEXT`) or comma separated values (`CSV`) format.

If the data in the file does not use the default column delimiter, such as the escape character or null string, you must specify the additional formatting options. Formatting options are the same as the `COPY` command.

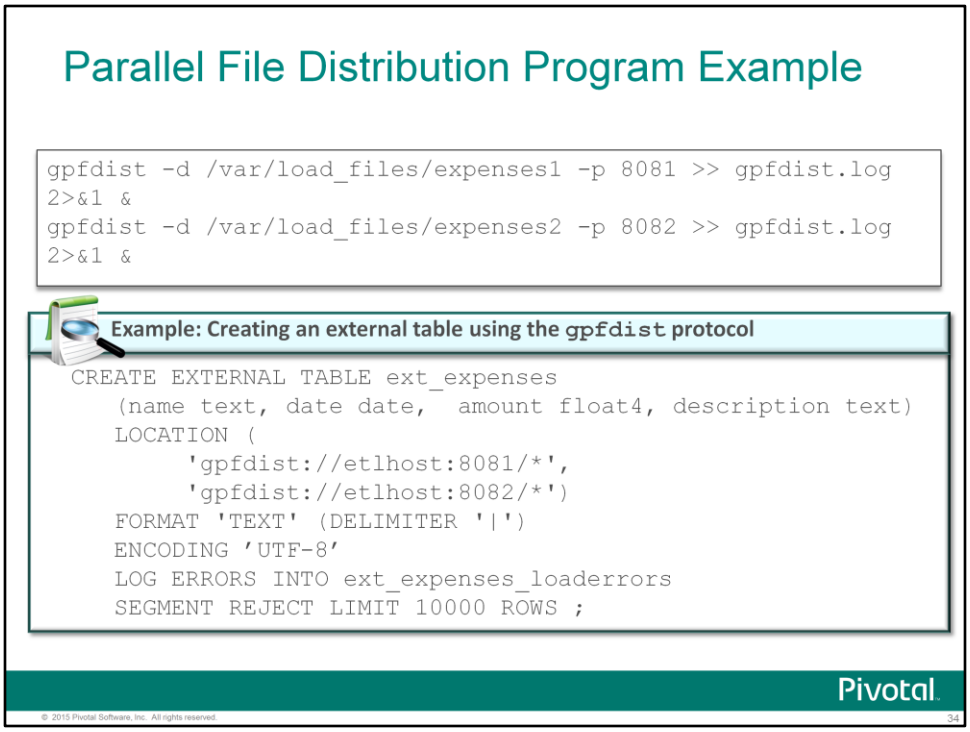


The parallel file distribution program, `gpfdist`:

- Is a C program which uses the `HTTP` protocol.
- Can be run on a server outside the Greenplum array.
- Distributes data at a rate of 200 MB/s per `gpfdist` executing on the system.
- Can be configured with the parameter: `gp_external_max_segs`.
- Provides full parallelism for the best performance.

`gpload` is a data loading utility that acts as an interface to Greenplum's external table parallel loading feature. Using a load specification defined in a YAML formatted control file, `gpload` executes a load by:

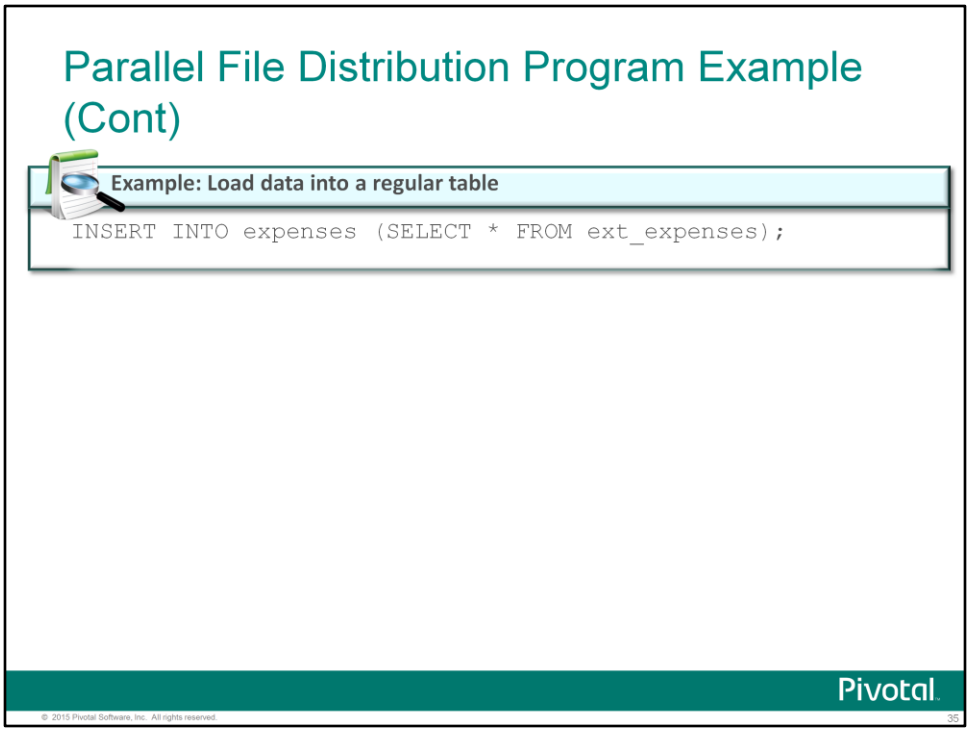
- Invoking the Greenplum parallel file server program.
- Creating an external table definition based on the source data defined.
- Executing an `INSERT`, `UPDATE` or `MERGE` operation to load the source data into the target table in the database.



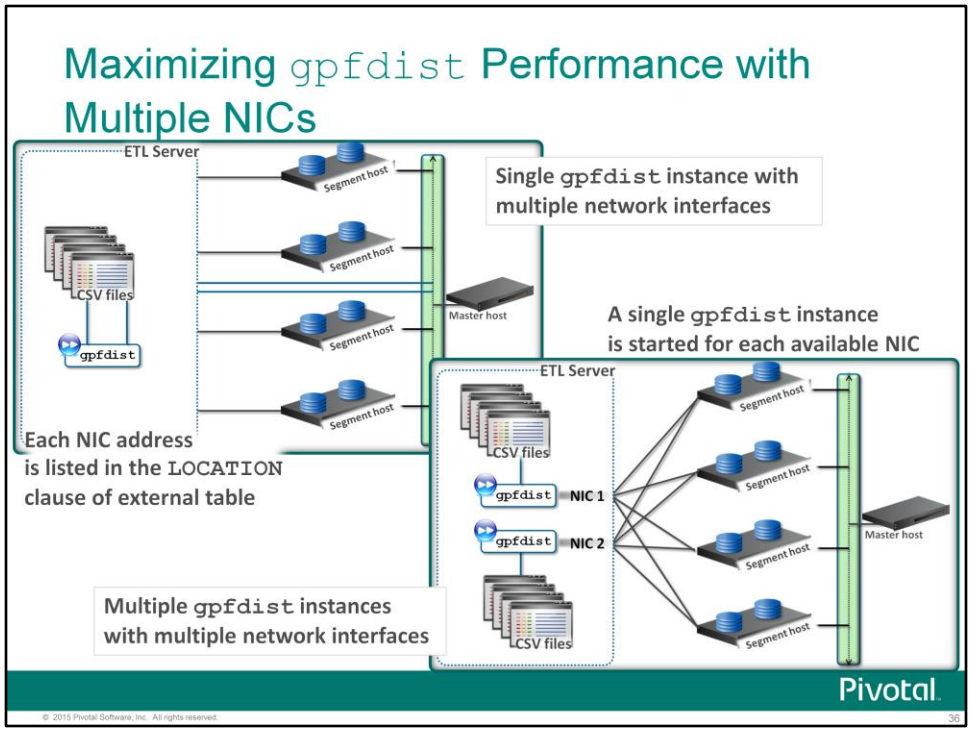
In the example shown, the parallel file distribution program is started and points to the directory, `/var/load_files/expenses`. Each instance opens a unique port number: port 8081 in the first example and port 8082 in the second.

The external table that is created connects to the parallel file distribution programs started on ports 8081 and 8082 of the server.

Issuing a `SELECT` statement on the external table allows you to view the data pulled in through the parallel file distribution program into the external table.



Once the external table has been created, data can now be loaded into a regular table. The table has the same columnar definitions as the external table. Once the data has been loaded, it returns how many rows have returned an error and the final number of rows that have been loaded into the table, `expenses`.



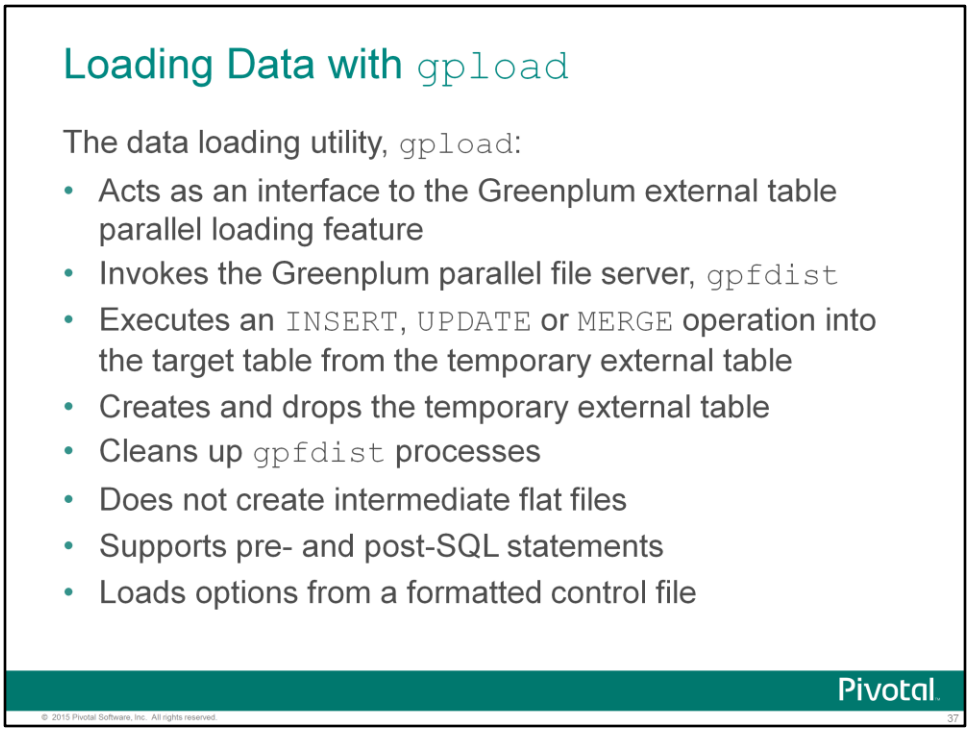
The parallel file distribution program, `gpfdist`, can take advantage of multiple NICs, maximizing network bandwidth between the systems serving the source files, ETL servers, and the Greenplum Database.

In the first example, a single `gpfdist` process is executed on the ETL server. The external table defines two host locations, one for each network interface card in the system. The source files can exist in a single directory on the ETL server. All segment hosts will use all NICs on the ETL simultaneously, thereby increasing bandwidth and helping to improve ingest performance.

The second example shows multiple `gpfdist` instances with multiple network interfaces. To improve the load performance, you divide the data source files evenly among the multiple `gpfdist` programs.

In general, the following steps occur during runtime:

- All segments access all `gpfdist` programs in parallel according to what is defined in the external table definition.
- The segment hosts must be able to connect to the `gpfdist` host using the HTTP protocol to access the files.
- `gpfdist` parses out the data in the files to the segment hosts evenly.



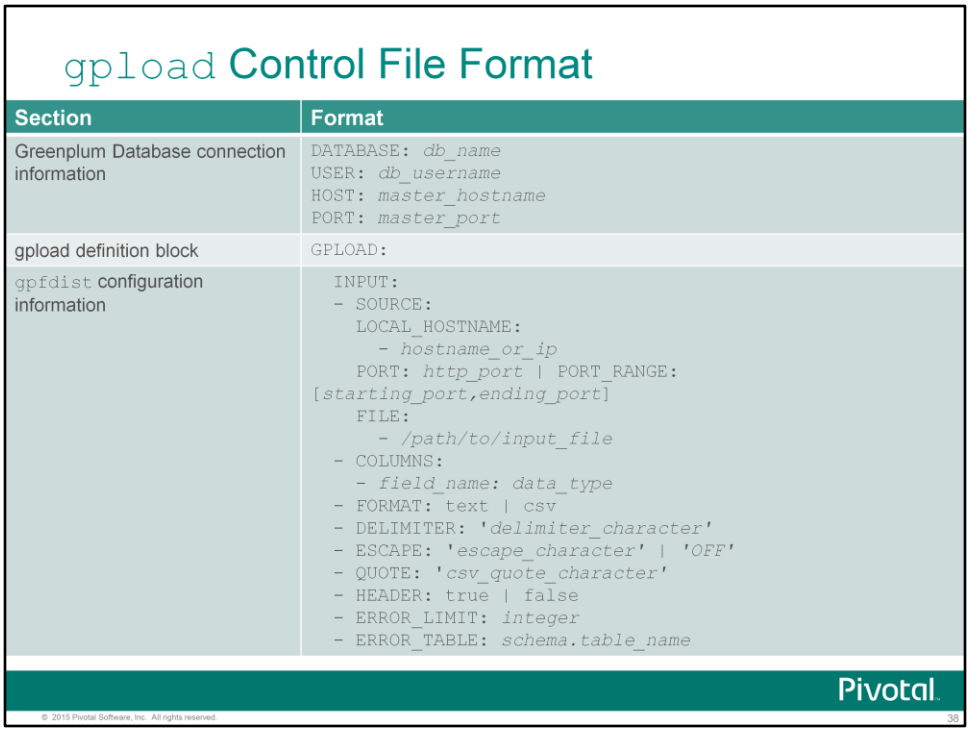
The `gpload` utility:

- Acts as an interface to the Greenplum external table parallel loading feature, automatically defining the external table in the process.
- Invokes the parallel file server, `gpfdist` on remote server.
- Executes SQL against the external table, including `INSERT`, `UPDATE`, or `MERGE` to load the source data into the target table in the database.
- Drops the temporary external table once it has completed the load.
- Terminates the `gpfdist` processes on the remote server.

The `gpload` utility is a wrapper utility that uses information in a control file to eliminate the manual steps related to bulk loading in parallel. It does not create any intermediate files while processing the data load.

Additional load functionality can be realized by the execution of SQL prior and subsequent to the actual load process as the `gpload` utility supports the use of pre and post SQL statements.

The control file is in a human readable data serialization format, YAML, Yet Another Multicolumn Layout. This borrows concepts form C, Perl and Python and XML.



The `gpload` control file:

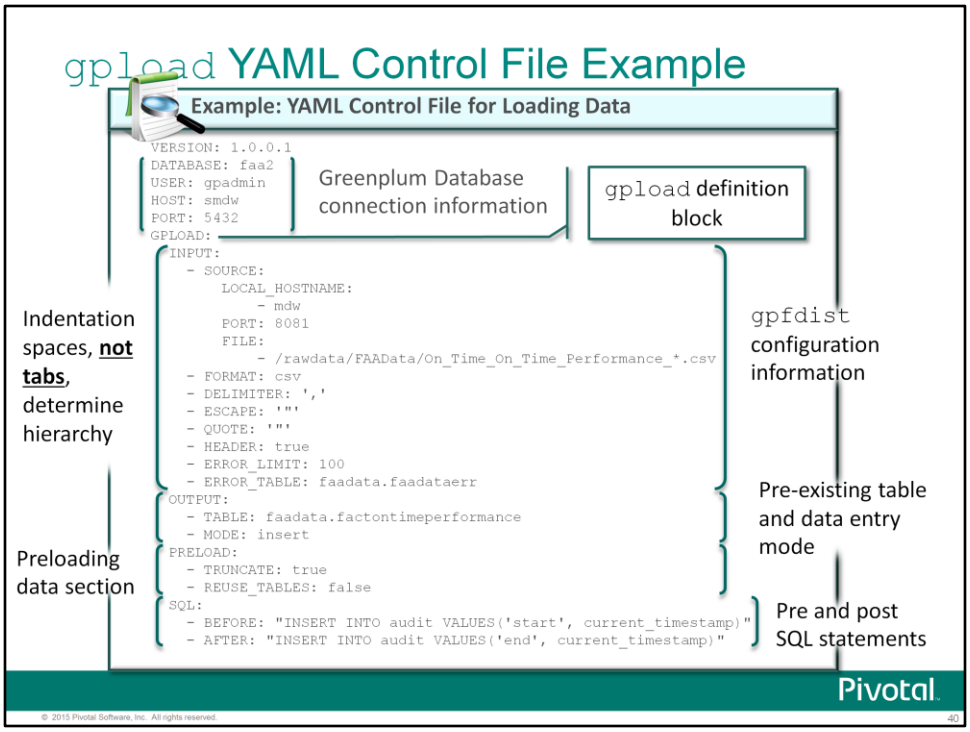
- Is based on YAML 1.1
- Implements its own schema for defining the various steps of a Greenplum load operation.
- Specifies the:
    - Greenplum Database connection information.
    - `gpfdist` configuration information which contains information about the location of the data, the column names and types of the external data table to be created, the expected format, any delimiters and quotes, whether or not a header exists for each data file loaded, the error limit, and the error table to be generated for each error.
    - Destination table that the data will be loaded into. With gpload, the data is automatically loaded after the external table is created. The destination table must therefore exist prior to performing the gpload.
    - Preloading information determines what to do with the table before the data is loaded and whether or not to reuse the external table created as a part of this transaction.
    - Pre and post-SQL statements allow you to specify SQL statements to be executed before and after the data load has occurred.

gpload Control File Format (Cont)

Section	Format
Destination table information	OUTPUT: - TABLE: *schema.table_name* - MODE: insert | update | merge - MATCH_COLUMNS: - *target_column_name* - UPDATE_COLUMNS: - *target_column_name* - UPDATE_CONDITION: '*boolean_condition* - MAPPING: *target_column_name: source_column_name | 'expression'*
Preloading information	PRELOAD: - TRUNCATE: true | false - REUSE_TABLES: true | false
Pre and Post SQL Statements	SQL: - BEFORE: "*sql_command*" - AFTER: "*sql_command*"

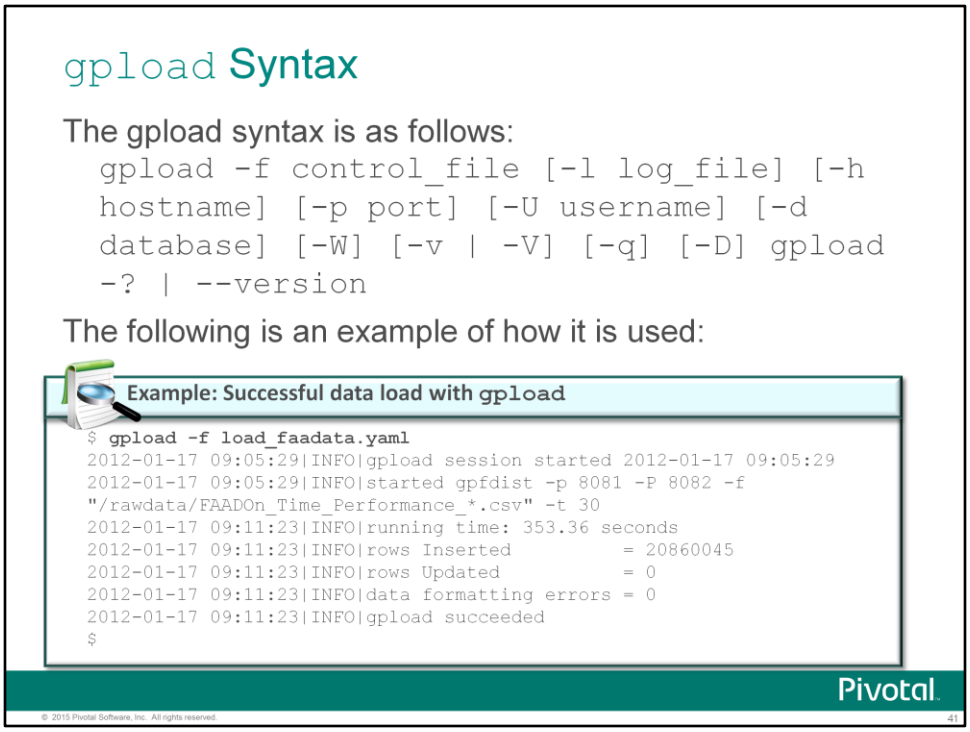
Pivotal

The `gpload` program processes the control file document in order and uses indentation (spaces) to determine the document hierarchy and the relationships of the sections to one another. The use of white space is significant. White space should not be used simply for formatting purposes, and tabs should not be used at all.



In this example, the YAML control file, load_faadata.yaml, defines how data from several source files are to be loaded into the `faa2` database. It is assumed that the `faadata.factontimeperformance` and `public.audit` tables already exist within the database. A reject limit of 100 errors is defined for this load, with any errors redirected to the `faadata.faadataerr` table.

Before the load occurs, the table is truncated. The external table created as a result of this load will not be reused. Reusing a table is useful if you are performing trickle loads or need to reduce system catalog bloat, which can occur when creating and dropping multiple temporary tables. By reusing the table, the table is not destroyed and recreated.



The `gpload` utility is executed by the superuser with the following syntax:

`gpload -f` *`control_name`*

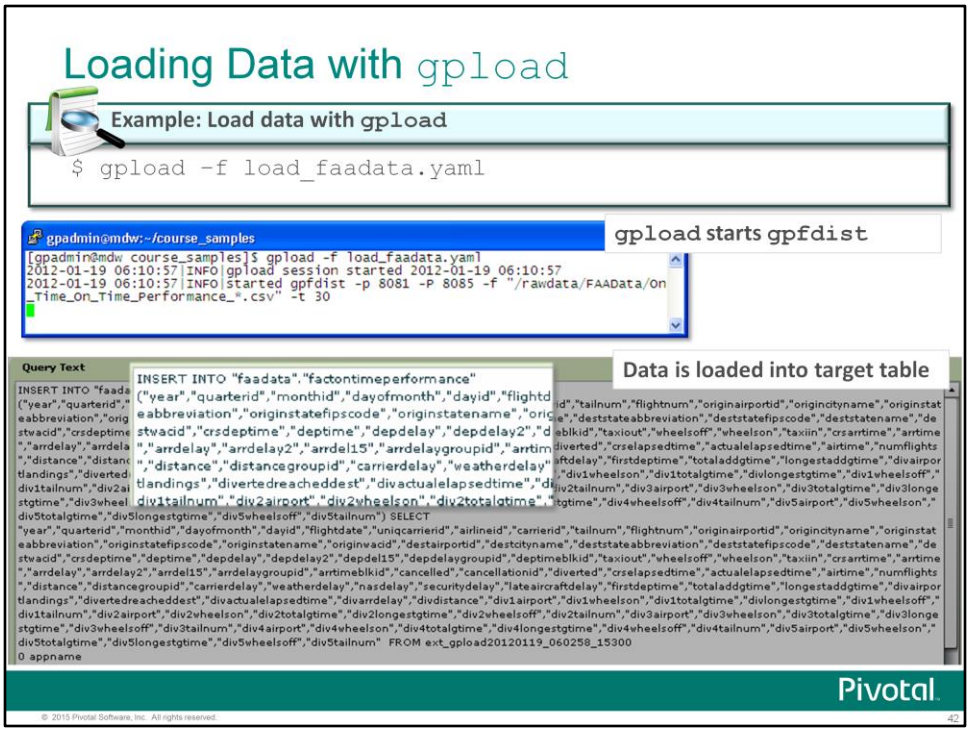
The command will display the `gpfdist` command that was issued while performing the data load.

Once the external table has been created, data is automatically loaded into the target database table specified in the control file. `gpload` will return a list of the number of rows inserted, updated, as well as any errors detected.

The `gpload` utility will connect to the database with the user specified with the `-U` option and the password specified with the `-W` option.

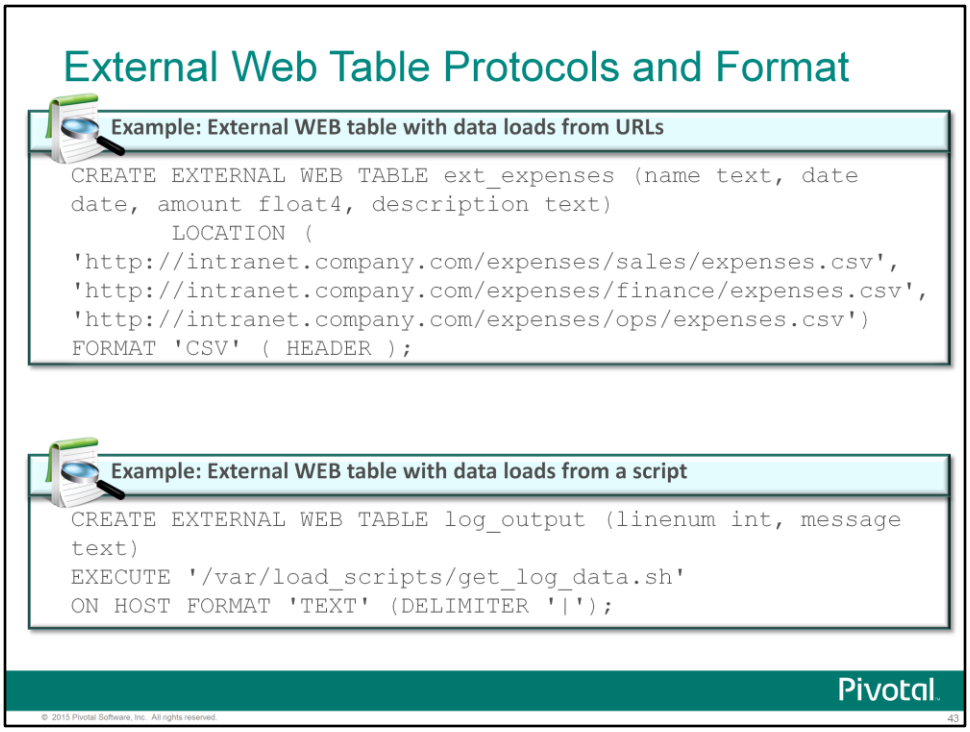
If the user is not specified on the command line, `gpload` will extract the user from one of the following in the order shown: the load control file, the environment variable `$PGUSER`, or lastly, the current system user name.

As with the user, if the password is not specified on the command line, `gpload` will extract the password from the following in the order specified: `$PGPASSWORD` variable, the password file specified by `$PGPASSFILE` or `.pgpass` if not set. Finally, if not found in any of these locations, you will be prompted for the password.



Once the control file has been created, issuing the `gpload` command starts a chain of events:

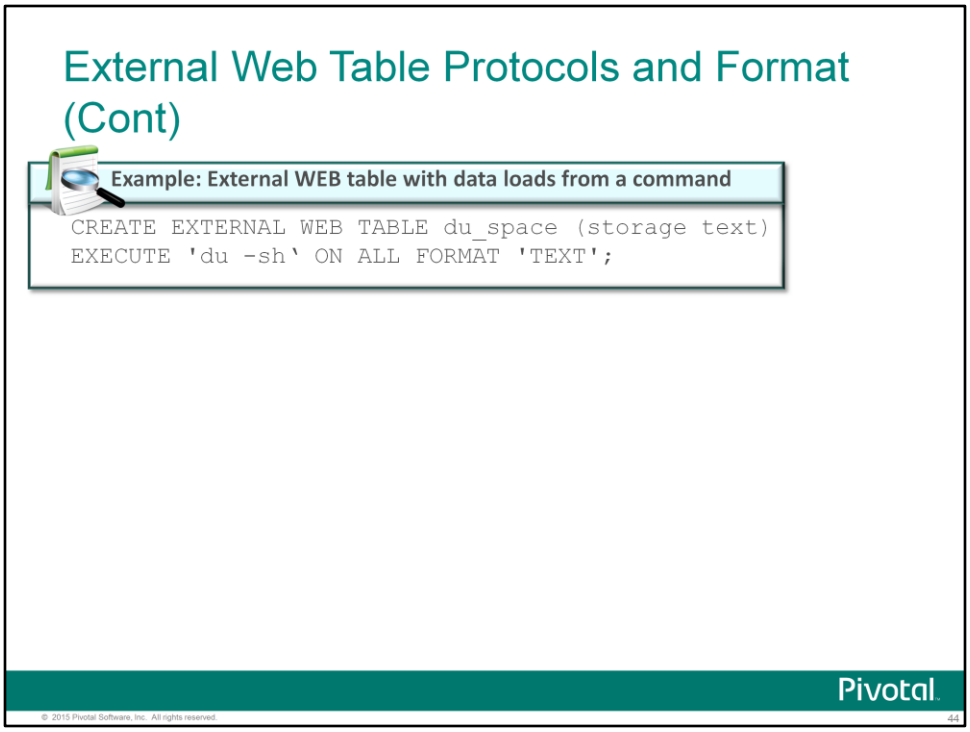
- The `gpfdist` program is started using one of the ports specified in the control file.
- `gpload` creates the external table based on the information provided in the control file.
- The data is loaded using the `INSERT INTO SQL` syntax into the target table.



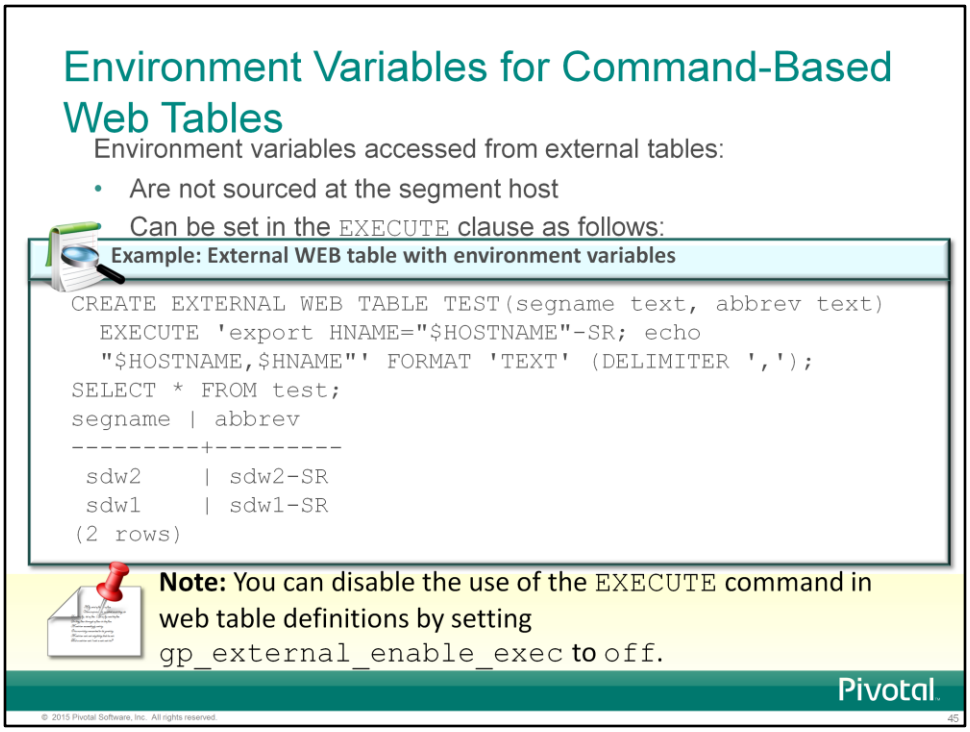
In the first example shown on the slide, an external web table is defined using the HTTP protocol. Three different URLs are associated with the external table, all using the CSV format.

The second example shows how to define an external table to a script. The script:

- Must be located on all segments in the same location.
- Must be executable by the superuser account.
- Will execute by one segment on each segment host.
- Executes in parallel on all segments.



In the third example, the external table is created on the command, `du -sh`. The command executes on all segments on all segment hosts.



When an external table command is executed, that command is executed from within the database and not from a login shell. If you rely on environment variables in external web table commands, such as the `$PATH` variable, keep in mind that the .bashrc or .profile of the current user will not be sourced at the segment host. You can set desired environment variables from within the EXECUTE clause of your external web table definition as follows:

```
CREATE EXTERNAL WEB TABLE ext_table (…) EXECUTE 'export
PATH=/usr/sbin'
```

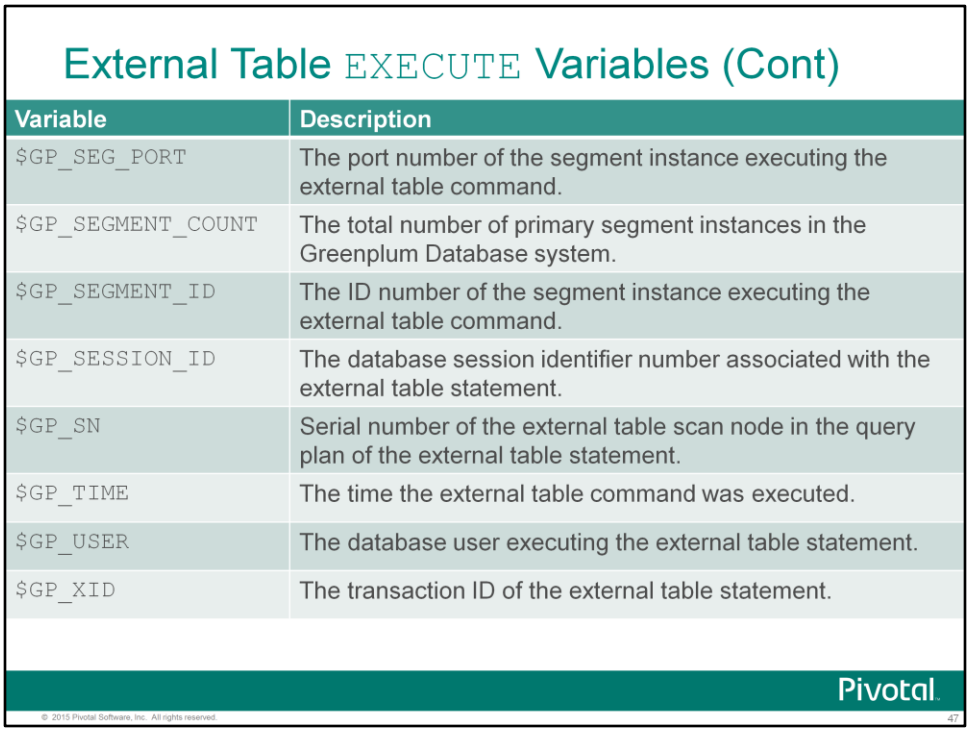
Note that you can disable the use of the `EXECUTE` command in web table definitions by setting the parameter, `gp_external_enable_exec`, to `off` in the `postgresql.conf` file.

External Table EXECUTE Variables

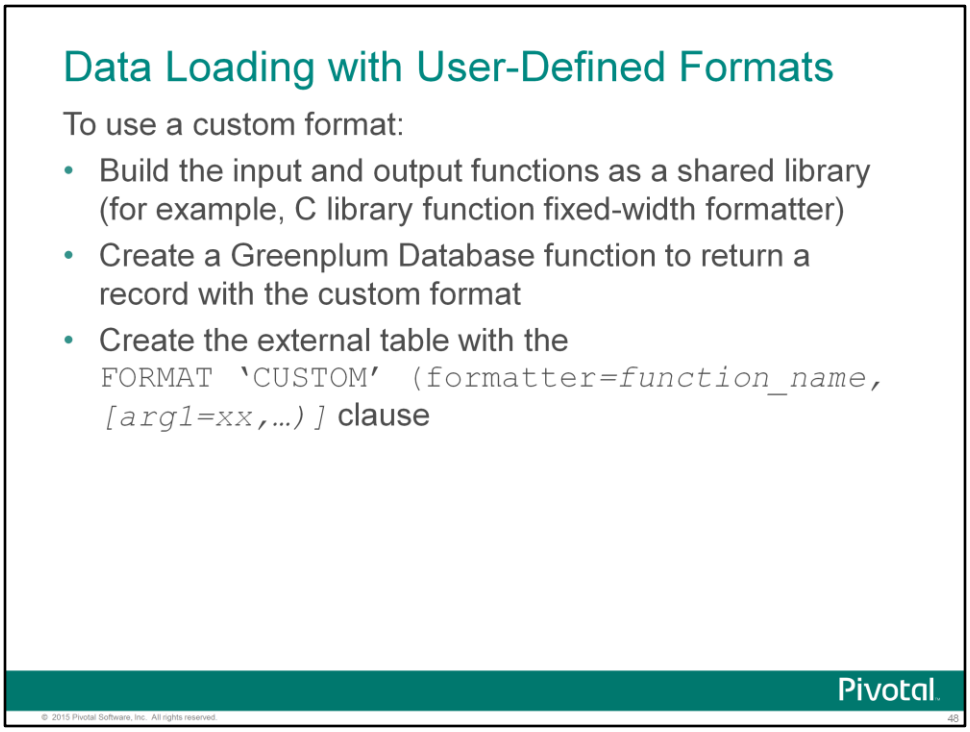
Variable	Description
$GP_CID	Command count of the session executing the external table statement.
$GP_DATABASE	The database that the external table definition resides in.
$GP_DATE	The date the external table command was executed.
$GP_MASTER_HOST	The host name of the Greenplum master host from which the external table statement was dispatched.
$GP_MASTER_PORT	The port number of the Greenplum master instance from which the external table statement was dispatched.
$GP_SEG_DATADIR	The location of the data directory of the segment instance executing the external table command.
$GP_SEG_PG_CONF	The location of the postgresql.conf file of the segment instance executing the external command.

Pivotal

The Greenplum Database variables shown in the table are available to operating system commands executed by an external table. These variables are set as environment variables in the shell that executes the external table command(s). They can be used to identify a set of requests made by an external table statement, at runtime, across the Greenplum Database array of hosts and segment instances.



External Table EXECUTE Variables (Continued)

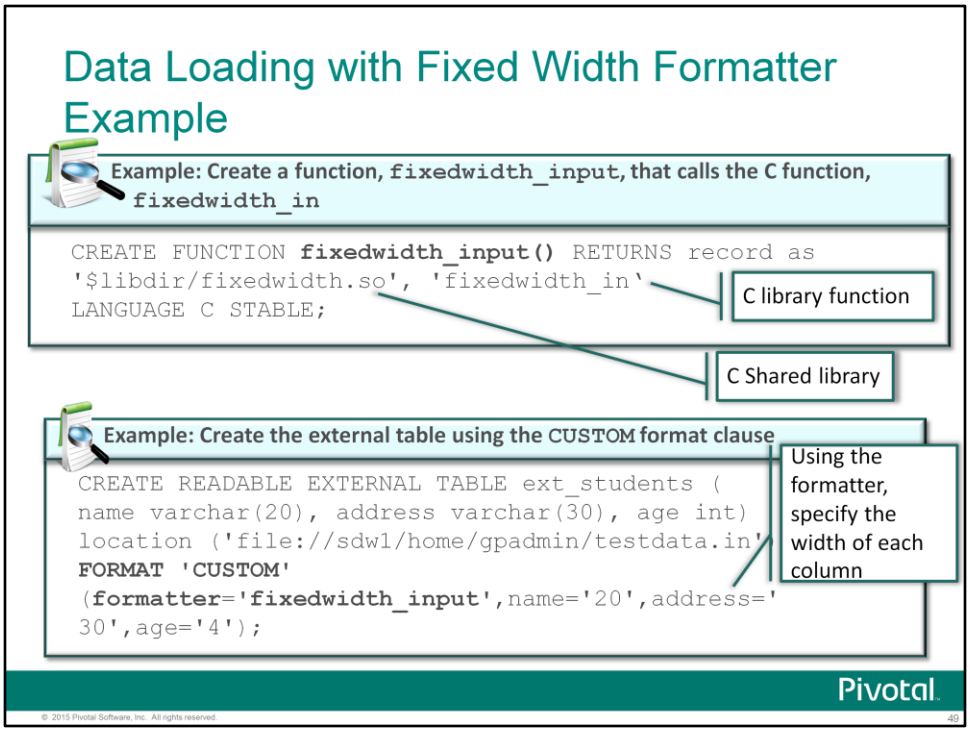


In addition to supporting data loads with the parallel file distribution program, Greenplum allows users to define their own custom formats, protocols, and locations to be used for data loads. The `FORMAT` clause with the `'CUSTOM'` argument allows you to specify the format of the data. The library that contains the formatting code can be registered as a function that is used for both reading and writing data to the final destination.

The overall purpose of the user-defined formats and protocols is to remove the need for the interim process, `gpfdist`, from the data load when performing an integration project. This method allows you to interact directly with the data in the method you choose, making the communication more streamlined and efficient for the project.

To use the custom format:

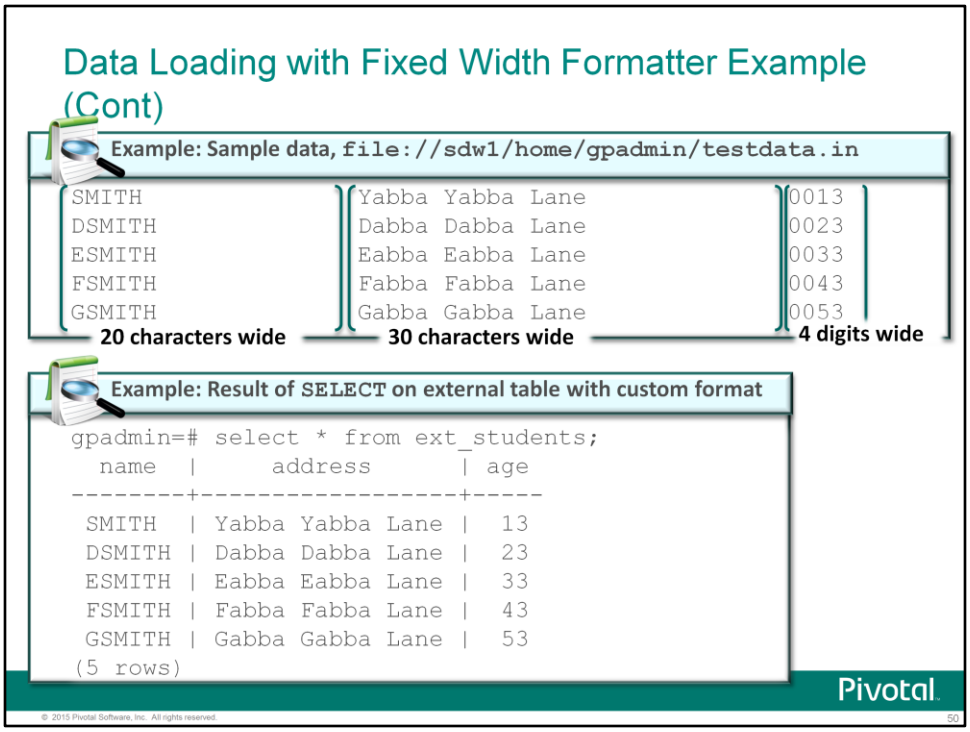
- First, build and install the shared library in a Greenplum accessible directory, such as `/usr/local/greenplum-db/lib`.
- Create a Greenplum function that will return a record of the custom format you have defined. The function should point to the installed library as well as the function within the shared library.
- Finally, create the external table specifying the Greenplum Database function you have created using the `FORMAT 'CUSTOM' ((formatter='GBDB_function', [arg1=, …]` clause.



In this example, a fixed width library, available with Greenplum, is used to load data that is in a fixed width format. The library is located in `/usr/local/greenplum-db/lib/postgresql` as `fixedwidth.so`.

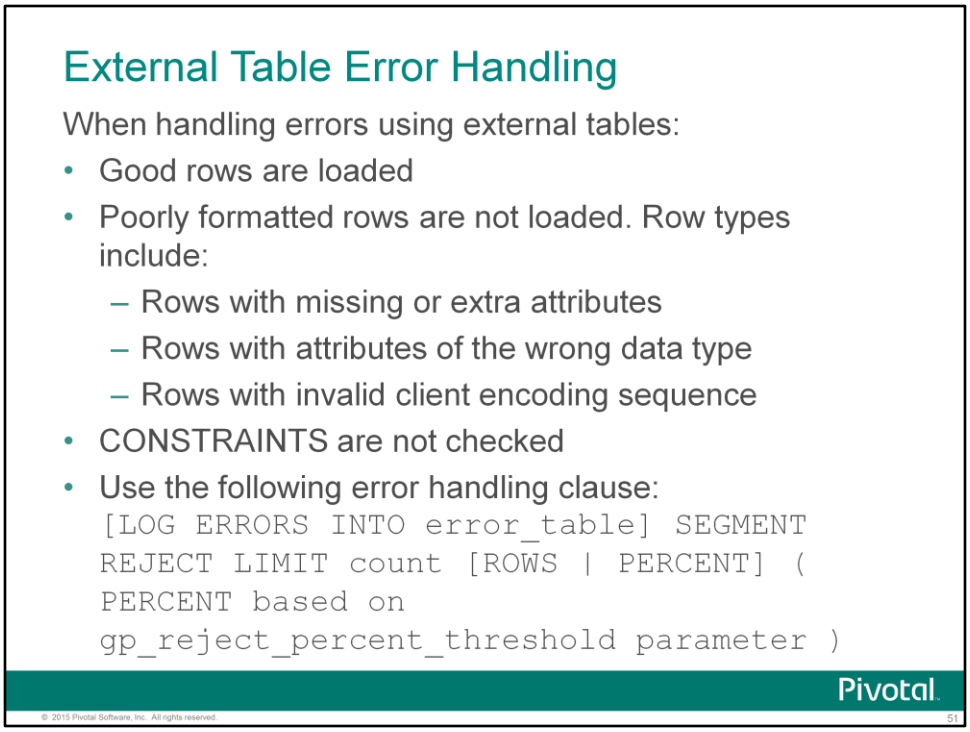
Create the function that will call this library. We will cover functions in greater detail later in the course. At this time, note that the function will return a record, or entire row, based on the format specified in the shared C library function, `fixedwidth_in`, from the shared C library, `fixedwidth.so`.

Once the function has been created, create the external table with the custom format. Pass in the name of each of the columns of the table as arguments to the `FORMAT 'CUSTOM'` clause and specify the width of each column.



With the data source in the specified location, verify that the content is read in without any issues.

The data source contains three columns with the first column 20 characters wide, the second column 30 characters wide, and the third column, 4 characters wide. The format defined in the external table requires that there is a total of 54 characters per record. If the data source does not meet that requirement, the load will fail.



The most common use of external tables is to select data from the table and load the data into regular database tables. This is typically done by issuing a CREATE TABLE AS SELECT or INSERT INTO SELECT command, where the SELECT statement queries external table data.

By default, if the external table data contains an error, the entire command fails and no data is loaded into the target database table. To isolate data errors in external table data while still loading correctly formatted rows:

- You define an external table with a `SEGMENT REJECT LIMIT` clause.
- Specify the number of error rows acceptable, on a per-segment basis, after which the entire external table operation will be aborted and no rows will be processed or loaded.

Note that the count of error rows is per-segment, not per entire operation. The following conditions then apply:

- If the per-segment reject limit is not reached, all rows that do not contain an error will be processed.
- If the limit is not reached, all good rows will be processed and any error rows discarded.

If you would like to keep error rows for further examination, you can optionally declare an error table using the `LOG ERRORS INTO` clause. Any rows containing a format error would then be logged to the specified error table.

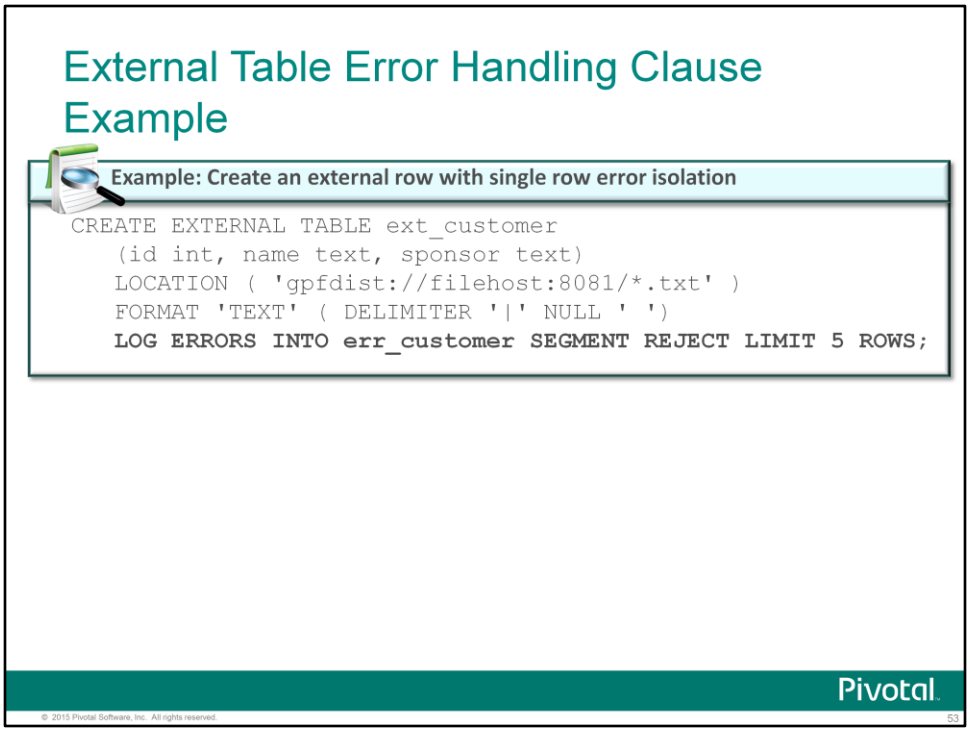
External Table Error Handling (Continued)

If PERCENT is used, the percentage of rows per segment is calculated based on the parameter `gp_reject_percent_threshold` . The default is 300 rows.

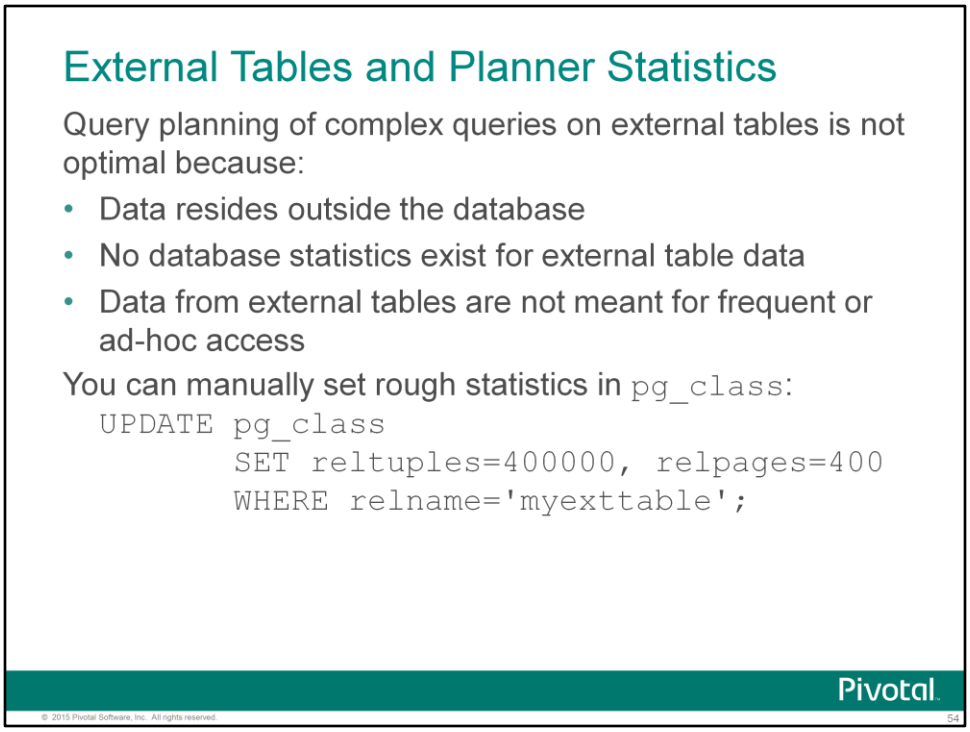
Single row error isolation mode only applies to external data rows with format errors, such as:

- Extra or missing attributes
- Attributes of a wrong data type
- Invalid client encoding sequences.

Constraint errors such as violation of a `NOT NULL`, `CHECK`, or `UNIQUE` constraint will still be handled in *all-or-nothing* input mode. Constraints are set on the target table, not the external table, so constraint errors are not caught until the violation row is inserted into the target table. A constraint violation will still cause the entire load to fail. However it is easy to use SQL commands to filter out constraint errors as a condition of the `INSERT` command.



In this example, an external table is created with the clause `SEGMENT REJECT LIMIT`. The data will therefore be scanned in single row isolation mode. This is helpful in isolating errors during data loads from an external table. Rows with formatting errors, extra or missing attributes, attributes of the wrong data type, and invalid client encoding sequences are caught and logged to the `err_customer` database table. A limit of 5 rows has been defined for this error handling routine.



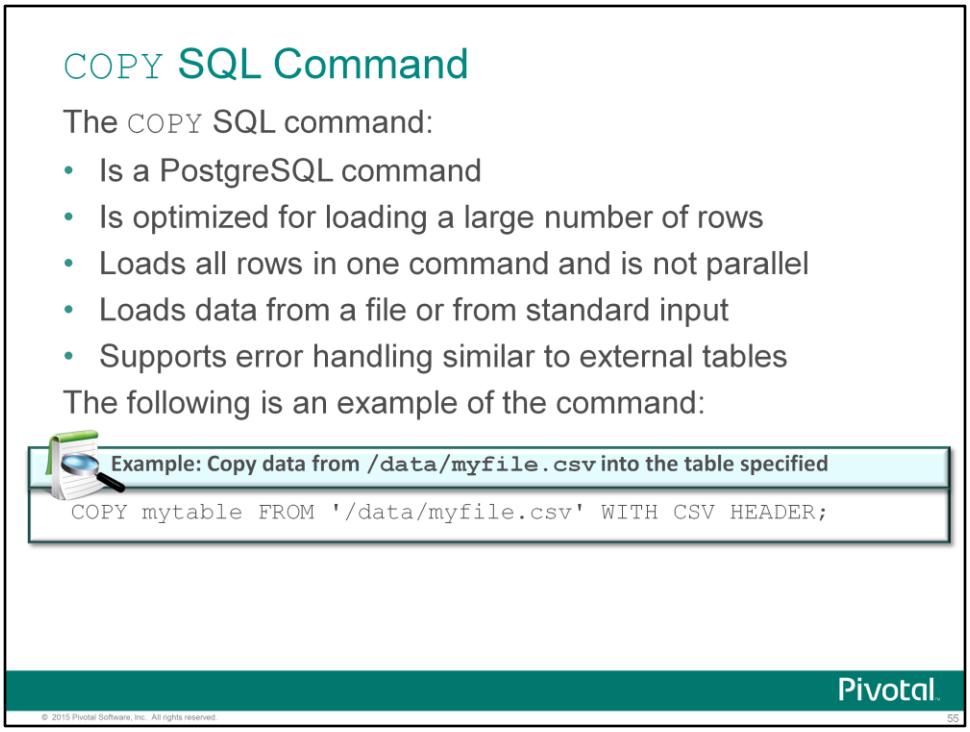
Query planning of complex queries involving external tables is not optimal because:

- There are no statistics to help the query planner choose a good plan.
- The data is not in the database.
- External tables are really not meant to be used in this way.

As a workaround, you can manually edit `pg_class` and `set reltuples`, the number of estimated rows, and `relpages`, the number of 32K size pages, after creating an external table definition. This is the only external table statistics that is currently used by the planner.

By default this is set to accommodate large tables to be safe – on millions of rows and 1000 pages.

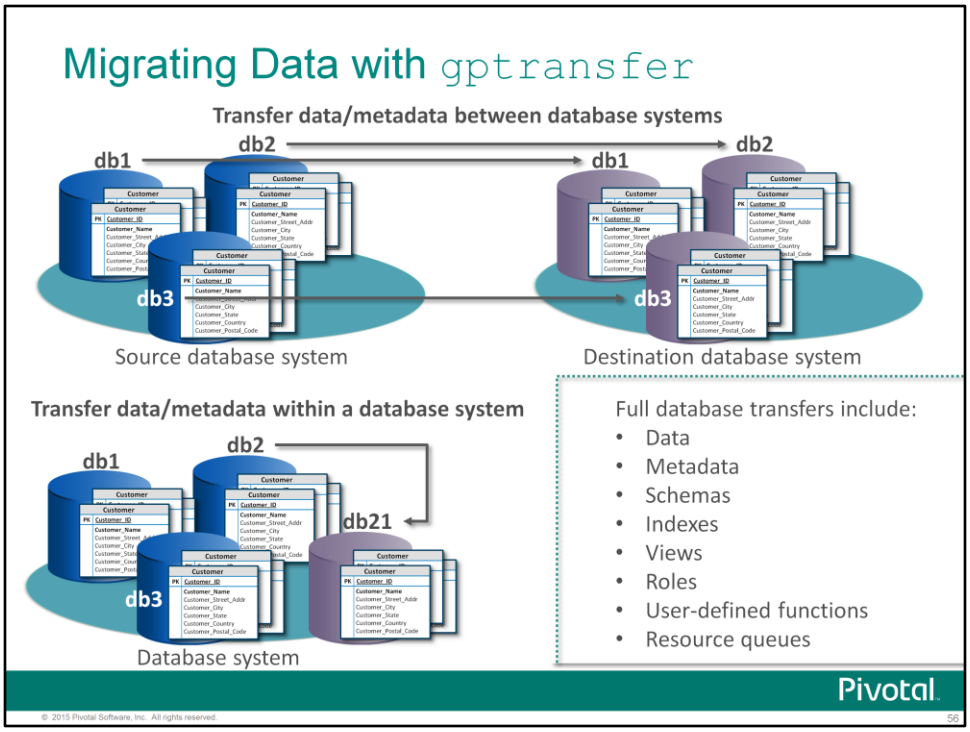
Other than in this use case, it is not recommended to update the `pg_class` catalog table. If an incorrect change is made to the table, it can result in problems for the Greenplum Database instance.



The COPY command:

- Is a PostgreSQL command that loads multiple rows in one command, instead of using a series of `INSERT` commands.
- Is optimized for loading large numbers of rows.
- Is less flexible than `INSERT`, but incurs significantly less overhead for large data loads. Since `COPY` is a single command, there is no need to disable `autocommit` if you use this method to populate a table.
- It loads data from a file or from standard output.
- Supports error handling.

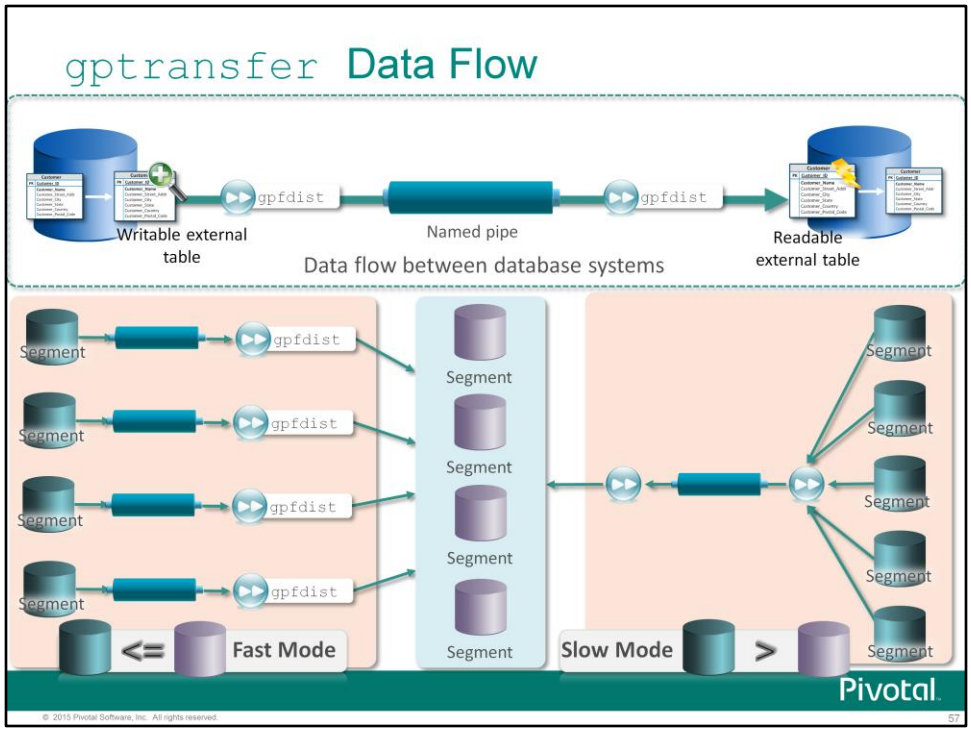
The table must already exist before copying to the table.



The `gptransfer` utility is used to transfer data objects and their metadata between databases. The databases can reside in the same database system or in a different source and target database system. The utility uses `gpfdist` to transfer data in parallel across all segments at the highest transfer rates available to the source and target database systems.

Data can be transferred between two database systems or between databases within the same cluster. The following options for data transfer are available:

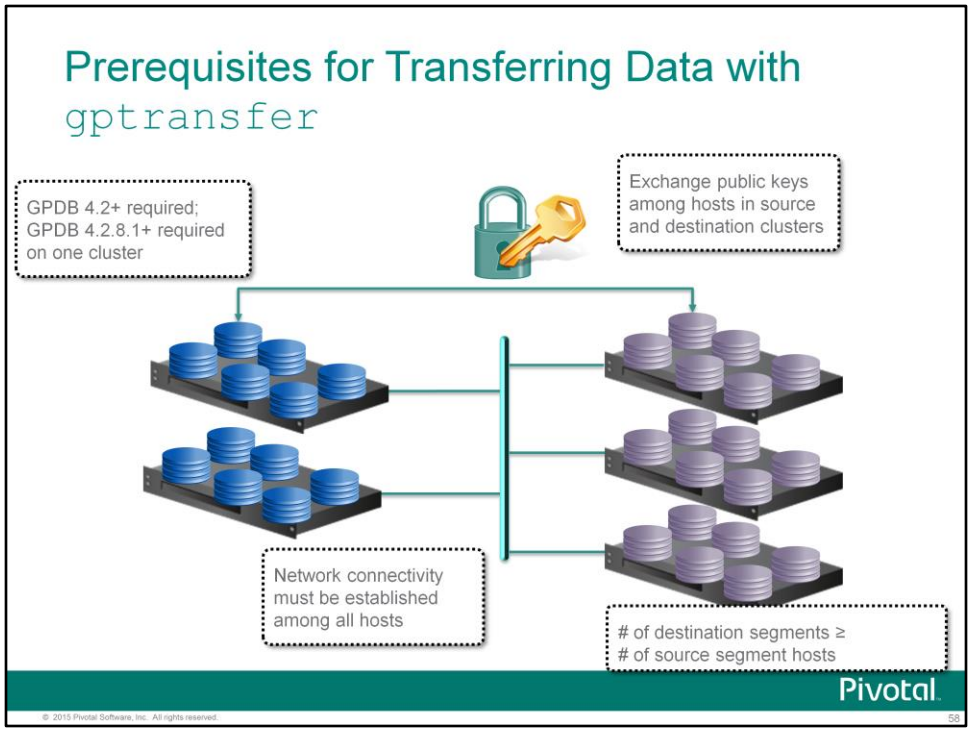
- The entire database from one cluster can be transferred to another cluster. All databases, excluding `postgres`, `template0`, and `template1`, are transferred from the host to destination cluster. In addition to transferring the data and metadata for the tables, the database schemas, indexes, views, roles, user-defined functions, and resource queues are also recreated to the destination system. The `postgres.conf` and `pg_hba.conf` files must be manually transferred as these are not included as part of the transfer process. Any extensions that were previously found in the source database system must be installed on the target database system. These are not automatically transferred.
- Database tables that you specify can be transferred from one cluster to another, or within the same cluster to a different target name.



The `gptransfer` utility uses `gpfdist` to transfer data from the source to the destination system. External tables are used for transferring the data, where writable external tables are created on the source database system for each database table to be transferred. A readable external table is created on the destination system for each database table that will be created. Named pipes transfer the data between the writable external table on the source system to the readable external on the target system. Once the transfer is complete, all external tables, named pipes, and `gpfdist` processes are reaped.

The number of `gpfdist` processes used depend on the number of hosts on the source system to the number of segments on the destination system. `gptransfer` uses fast mode or slow mode depending on your setup.

- If the number of segments on the destination cluster is greater than or equal to the number of segments on the source cluster, `gptransfer` uses fast mode. In fast mode, one named pipe and one `gpfdist` process are configured for each segment within the source cluster. Each `gpfdist` process points to a writable external table for that segment. This is an optimal solution that maximizes the data transfer rates between systems.
- If the number of segments in the destination cluster is less than those found in the source cluster, `gptransfer` uses the slow mode. In the slow mode, each segment host is configured with a single writable external table associated a named pipe. Data from all the segments on the segment host are consolidated to the writable external table and are funneled through the `gpfdist` process. The transfer rate is reduced in this case because there are fewer `gpfdist` processes, one per segment host, serving data to the destination cluster.

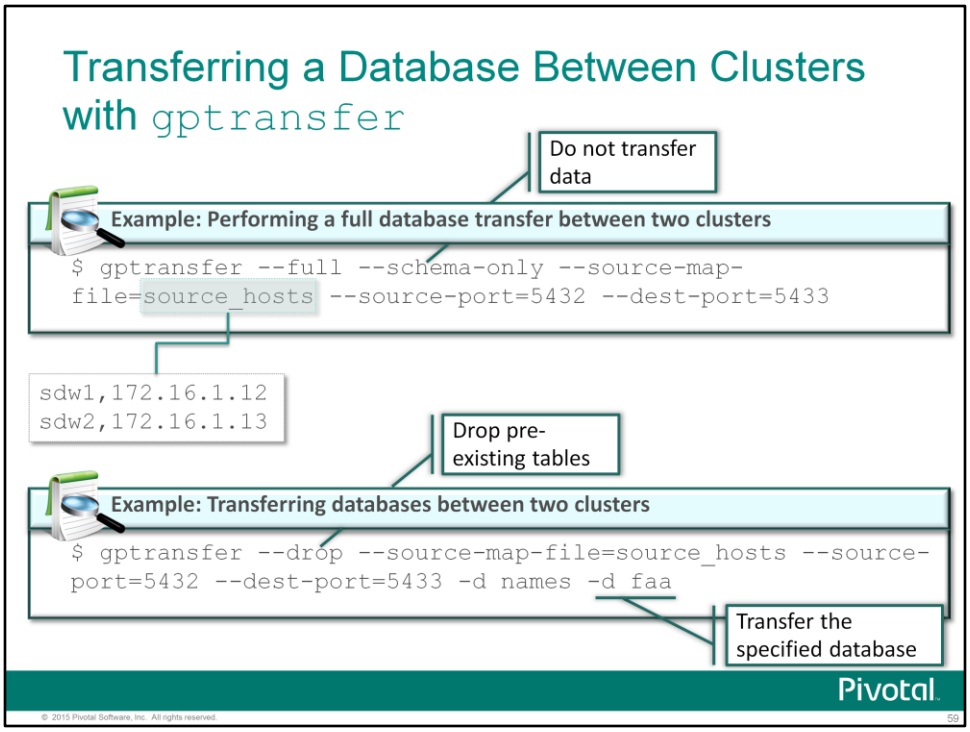


`gptransfer` requires several conditions are met before attempting to transfer data:

- Data can only be transferred between Greenplum database systems. There is currently no support for other databases, including HAWQ.
- At least one of the Greenplum Database systems must have the `gptransfer` utility installed. The utility was introduced in GPDB 4.2.8.1. This does also require that both systems are at or greater than GPDB 4.2, where one of the systems is running GPDB 4.2.8.1 or higher.
- The utility can be executed from either the source or destination system. If you are transferring data between systems, you must specify which system acts as the source and which acts as the destination system.

There are additional networking and authentication requirements that must be met before proceeding:

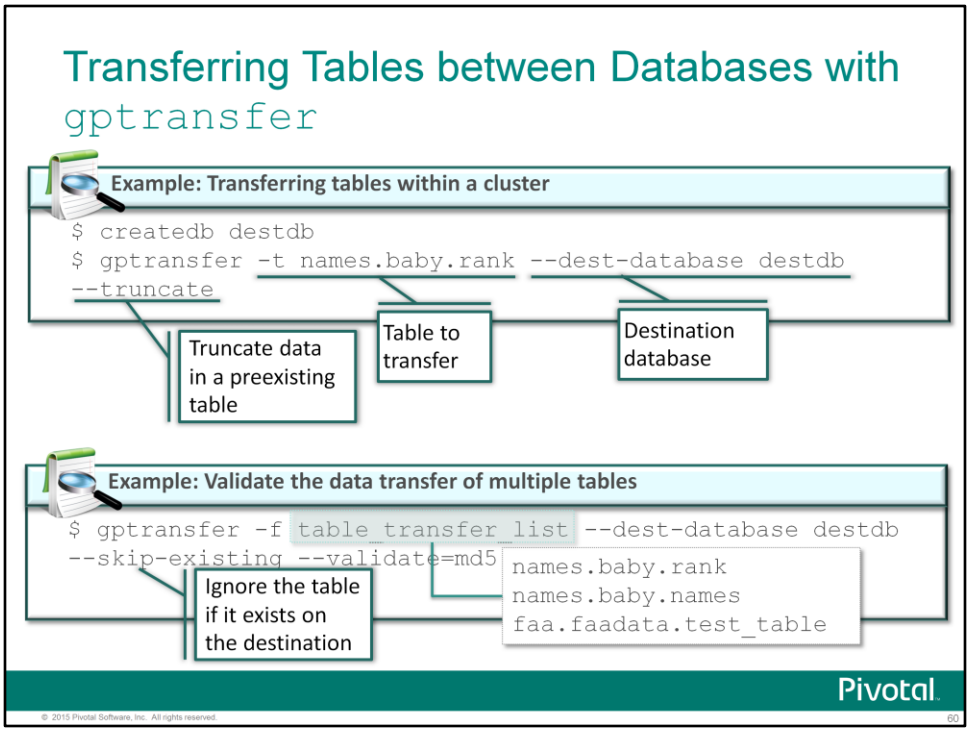
- `gptransfer` requires that all segment hosts in both clusters have network connectivity to each other and have exchanged public keys with each other. Create a host file with the list of all segment hosts in both clusters and use the `gpssh_exkeys` utility to exchange public keys among the hosts. `gptransfer` uses IP addresses for communication between clusters to avoid issues with name resolution.
- You must have the same or more segments defined in the destination cluster than there are segment hosts defined in the source cluster. For example, if you have 2 segment hosts in the source cluster, you must have two or more segments in the destination cluster.



The first example highlights the syntax for transferring data from one cluster to another. As the clusters share the same hosts, the port options are used to specify which is the source cluster and which is the destination cluster. When using the `--full` option to transfer between database systems, it is recommended that the data and metadata be transferred in stages. First, transfer and recreate the schema and underlying objects. Once the metadata has been created, transfer the actual data between systems. Use the --truncate or --drop options to prevent a failure should the table already exist. When performing a full database transfer, the destination database cannot have any preexisting databases other than the system defined databases.

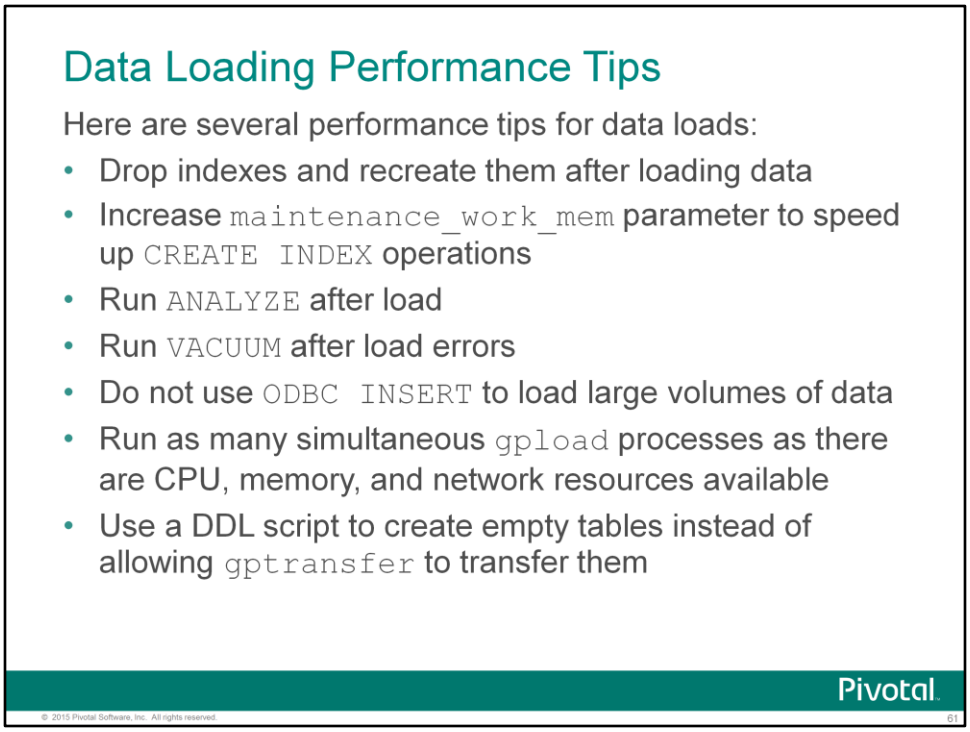
You must also create a host mapping table to define the segment hosts in the cluster along with their IP addresses. Hosts are defined one to a line. If the clusters do not share the same host, include the `--source-host=<hostname>` and `--dest-host=<hostname>` as part of the command line.

The next example shown continues the process of transferring an entire database. You specify the database you wish to transfer using the `-d <database_name>` option. Multiple databases can be specified, each preceded with the `-d` option. In this example, two databases are transferred between systems. The `--drop` option is used to drop the table if it already exists on the destination system.



In the first example, a database which will act as the destination for a transfer is created. The table rank, found in the baby schema of the names database is copied to the destination database specified. If the table existed, it is truncated and populated with the data being transferred. Multiple tables can be specified on the command line with each preceded by the `-t` option. As with any data moves, any indexes associated with the table are recreated during the transfer.

The second example shows multiple tables, specified in a file and called with the `-f <filename>` option, are transferred between databases. Each table to be transferred is listed on a line by itself with the notation *database.schema.table*. The `--skip-existing` option specifies that `gptransfer` skip the table if it already exists in the destination database. This transfer is validated using MD5. You can use MD5 or a row count to validate that the data has been transferred successfully. If validation fails, the tables that failed are logged to the file, `failed_migrated_tables_yyyymmdd_hhmmss.txt`, where *yyyymmdd_hhmmss* is a time stamp of when the data transfer process was started.

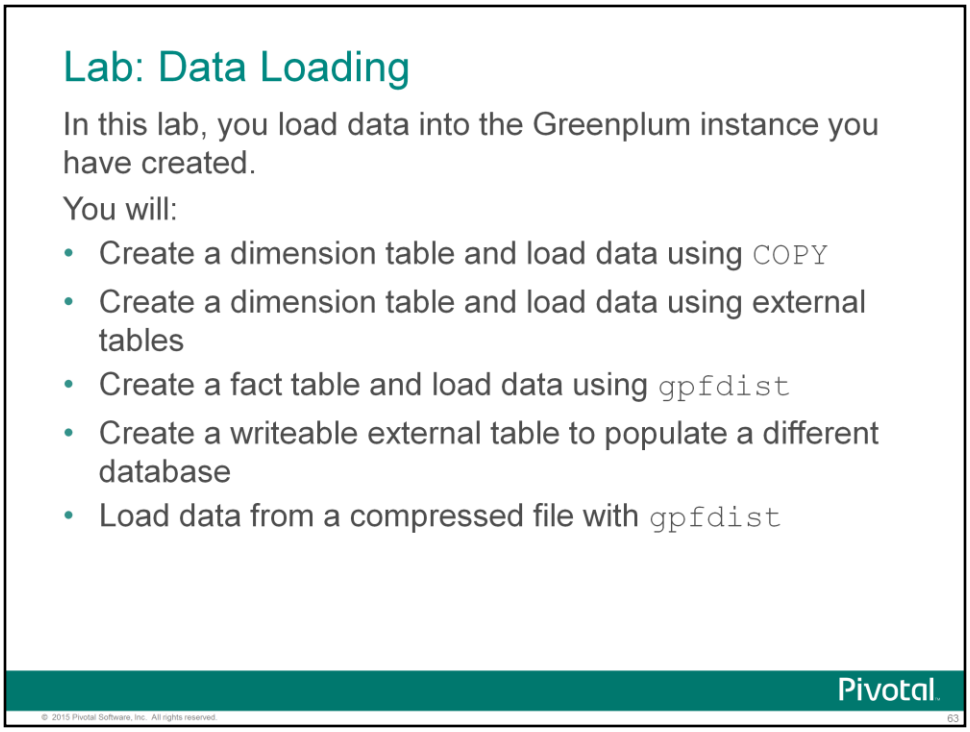


The following are performance tips to be aware of when loading data:

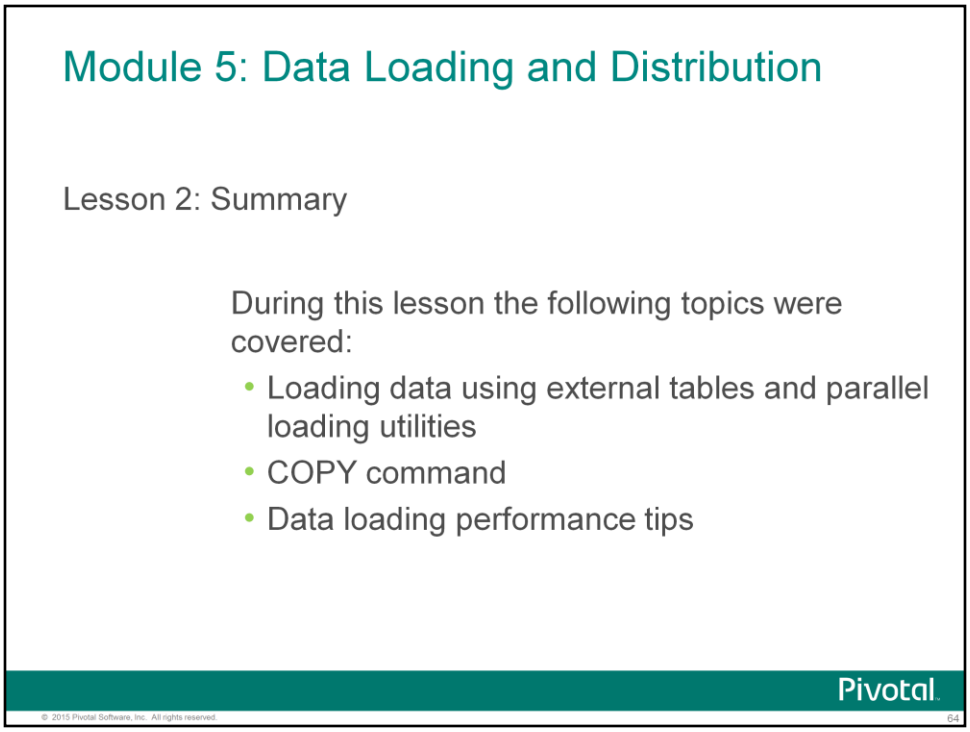
- **Drop Indexes Before Loading** — You can load data quickly by creating the table, loading the data, then creating any indexes needed for the table. Creating an index on pre-existing data is faster than updating it as each row is loaded. If you are adding large amounts of data to an existing table, it can be a faster to drop the index, load the table, and recreate the index.
- **Increase `maintenance_work_mem`** – Temporarily increasing the `maintenance_work_mem` server configuration parameter speeds up the `CREATE INDEX` commands. It will not help performance of the load itself. Dropping and recreating indexes should be done when there are no users on the system.
- **Always Run ANALYZE After Loads** — Whenever you have significantly altered the data in a table, running `ANALYZE` is strongly recommended. Running `ANALYZE` ensures that the query planner has up-to-date statistics about the table. With no statistics or obsolete statistics, the planner may make poor decisions during query planning, leading to poor performance on any tables with inaccurate or nonexistent statistics.

Data Loading Performance Tips (Continued)

- **Run `VACUUM` After Load Errors** — Both `COPY` and selecting from external tables are *all or nothing* operations. Both will stop a load operation if it encounters an error. When this happens, the target table may already have received earlier rows in the or load operation. Although these rows will not be visible or accessible, they still occupy disk space. This may amount to a considerable amount of wasted disk space if the failure happened well into a large load operation. Invoking the `VACUUM` command will recover the wasted space.
- **`ODBC INSERT`** – Insert rates, as opposed to `COPY` and inserting using external tables, can only load data at 60 INSERTs per second when the INSERTs are batched in one big transaction. You can get 400-500 statements per second if the statements are `UPDATE`, `DELETE` or `SELECT`.
- **Run as many simultaneous gpload processes as resources** – Take advantage of the CPU, memory, and networking resources available and run as many gpload processes at a single time as the system can handle. By taking advantage of resources, you increase the amount of data that can be transferred from ETL servers to the Greenplum Database.
- **Do not use gptransfer to transfer empty tables** – If there are a significant amount of empty tables in your transfer process, consider using a DDL script to recreate those empty tables instead of allowing the utility to do it. The overhead associated with transferring data with gptransfer is significant and should not be used lightly to transfer these tables.



In this lab, you load data into the Greenplum instance you have created.



This lesson covered how to load data into new and pre-existing tables using various loading methods, including the parallel loading utilities, external tables, and the COPY command. Performance tips for enhancing and improving data load times was also discussed.

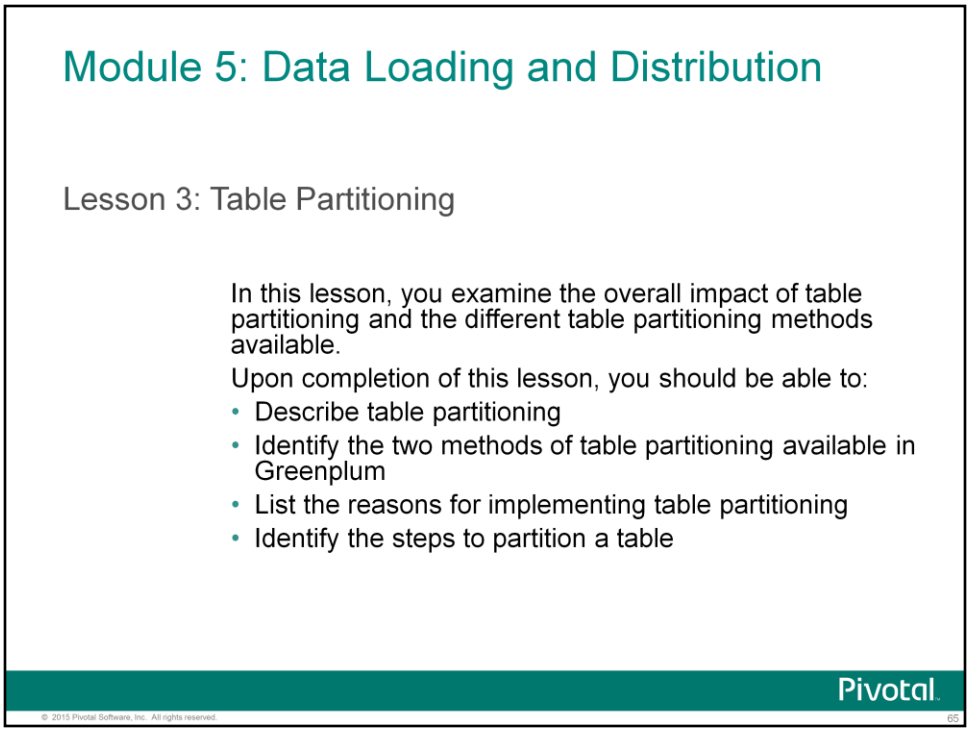


Table partitioning allows you to break your tables into consumable pieces that can improve performance during table scans.

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

- Describe table partitioning in the Greenplum Database.
- Identify the two methods of table available for partitioning a table.
- List the reasons for implementing table partitioning.
- Identify the steps you use to partition a table.

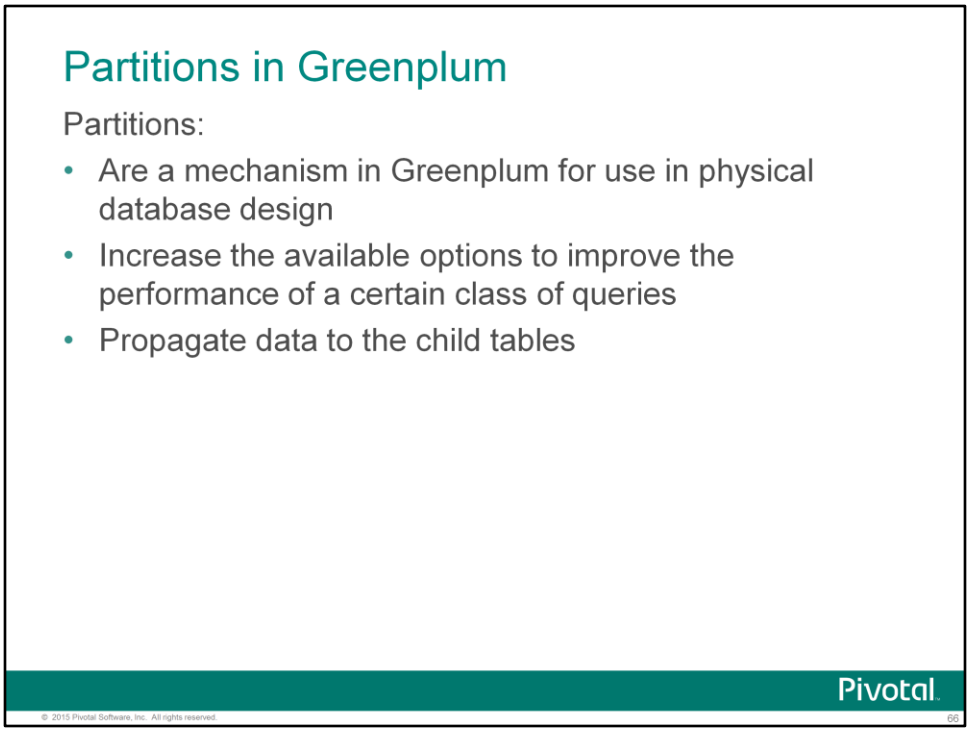


Table partitioning is used to logically divide large tables to improve query performance and facilitate data warehouse maintenance tasks.

All of the schema information from the parent table propagates to its children through table inheritance. However, parent and child need not have the same columns. It is important to keep in mind that parent tables contain no data and indexes and privileges are not passed from parent table to child tables. Therefore, if using indexes explicitly create indexes on each child table and grant permissions on each child table.

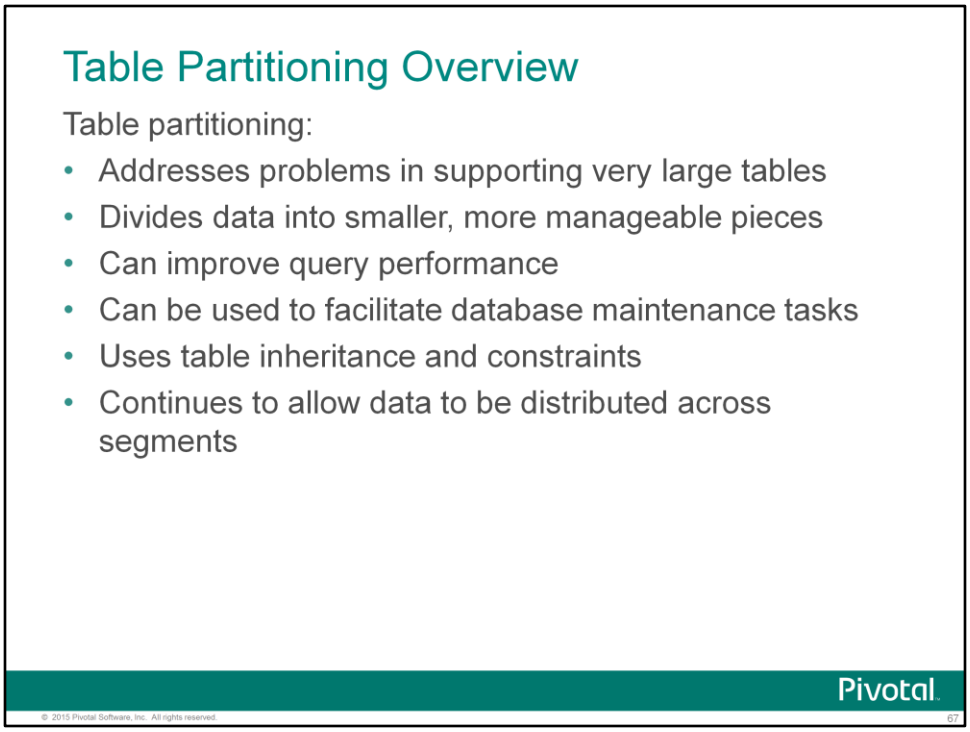
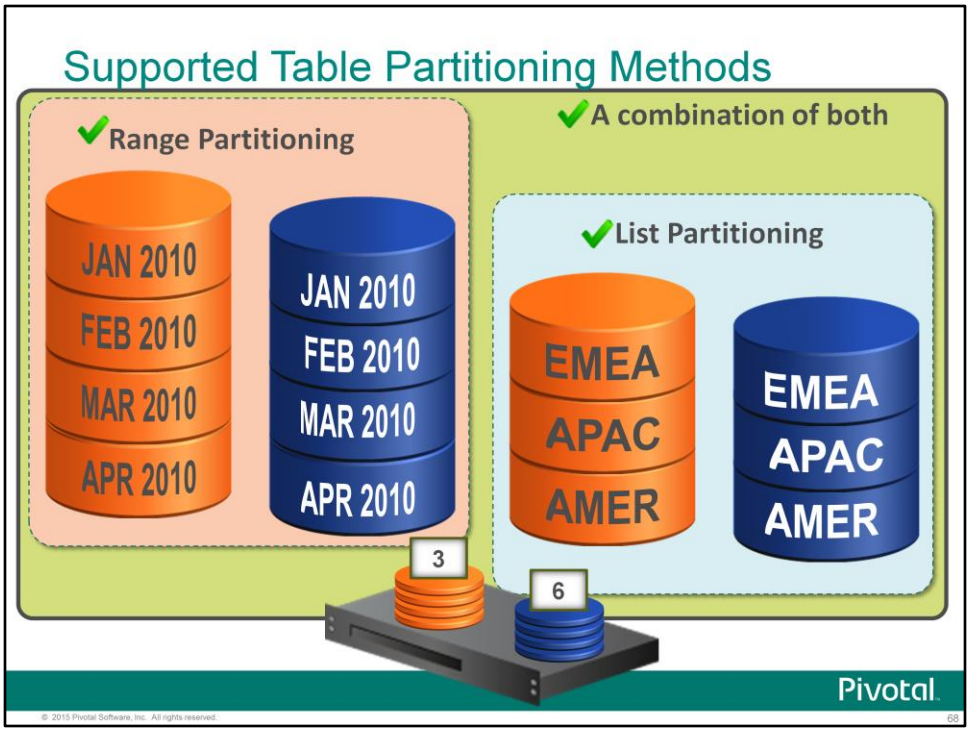


Table partitioning:

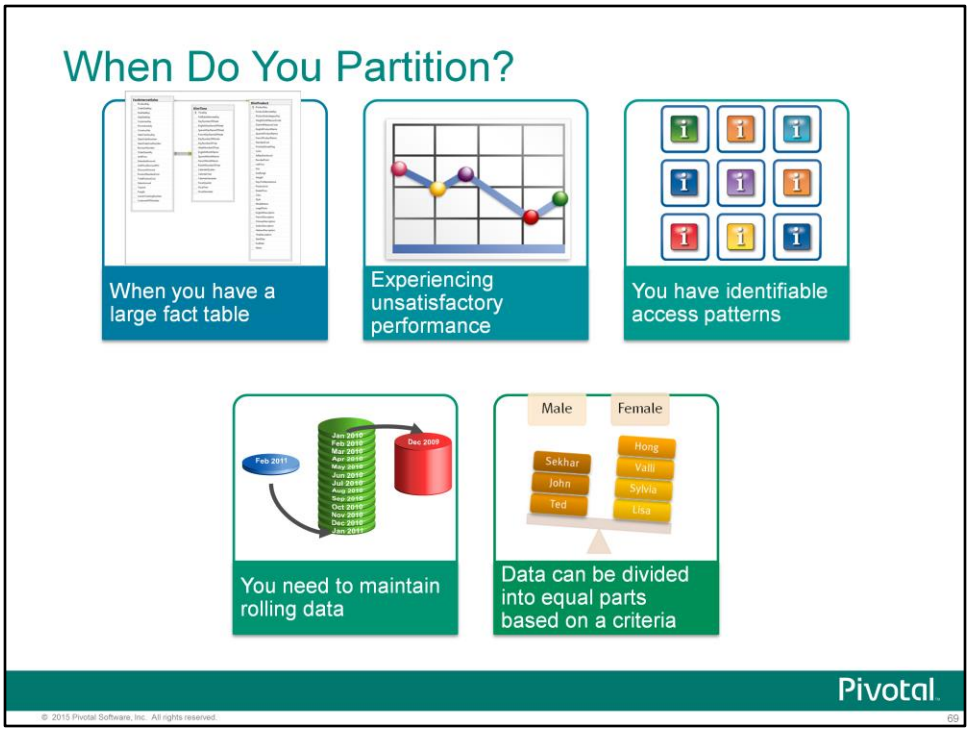
- Addresses the problem of supporting very large tables, such as fact tables, by allowing you to divide them into smaller and more manageable pieces.
- Can improve query performance by scanning only the relevant data needed to satisfy a given query. Data is eliminated either during the planning phase or at runtime. This helps to reduce the number of rows returned for joins and further actions.
- Can be used to facilitate database maintenance tasks such as rolling old data out of the data warehouse or speeding up the update of indexes.
- Works using table inheritance and constraints.
  Table inheritance creates a persistent relationship between a child table and its parent table(s), so that all of the schema information from the parent table propagates to its children.
  `CHECK` constraints limit the data a table can contain based on some defining criteria. These constraints are also used at runtime to determine which tables to scan in order to satisfy a given query.

In Greenplum Database, partitioned tables are distributed across the segments as is any non-partitioned table. Partitioning is the method of logically dividing big tables to improve query performance and maintenance. It does not effect the physical distribution of the table data.



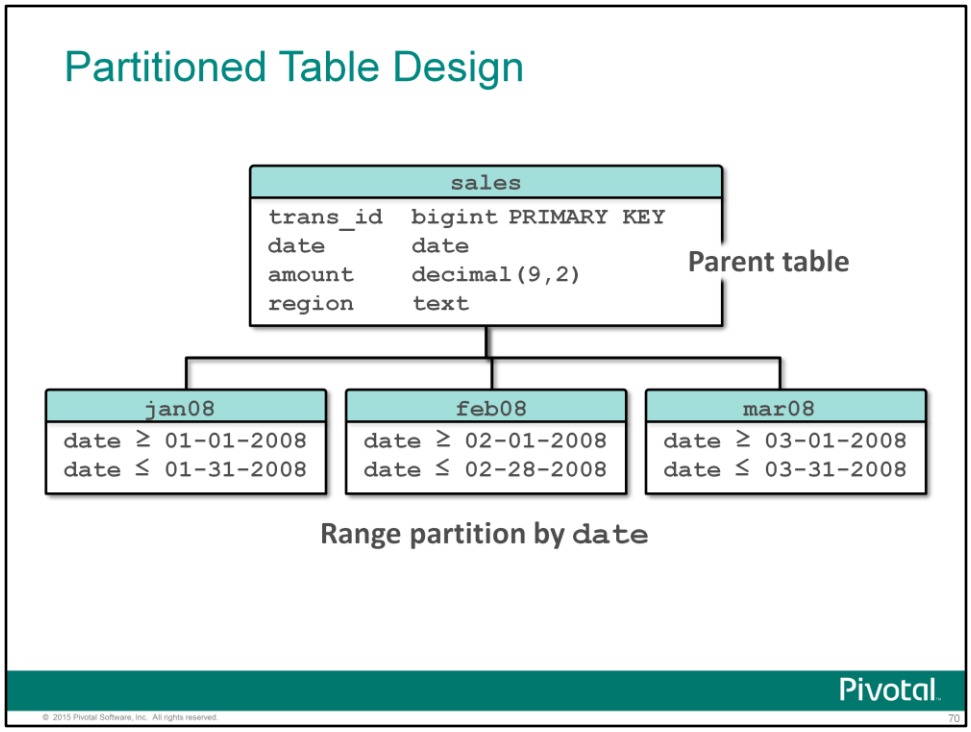
Greenplum Database supports:

- Range partitioning, where the data used for partitioning is based on a numerical range, such as date or price.
- List partitioning, where data used for partitioning is based on a list of values, such as gender or region.
- A combination of range and list partitioning.



There are several reasons why you would partition your data:

- **Large fact table** – Large fact tables are good candidates for table partitioning. If you have millions or billions of records in a table, you will see performance benefits from logically breaking that data up into smaller chunks. For smaller tables with only a few thousand rows or less, the administrative overhead of maintaining the partitions will outweigh any performance benefits you might see.
- **Unsatisfactory performance** – As with any performance tuning initiative, a table should be partitioned only if queries against that table are producing slower response times than desired.
- **Identifiable access patterns** – Examine the WHERE clauses of your query workload and look for table columns that are consistently used to access data. For example, if most of your queries look up records by date, then a monthly or weekly date-partitioning design might be beneficial.
- **Maintaining rolling data** – Is there a business requirement for maintaining historical data? Your data warehouse may require you keep the past twelve months of data. If the data is partitioned by month, you can drop the oldest monthly partition and load current data into the most recent monthly partition.
- **Dividing data into equal parts** – Choose a partitioning criteria that will divide your data as evenly as possible. If the partitions contain a relatively equal number of records, query performance improves based on the number of partitions created. For example, by dividing a large table into 10 partitions, a query will execute 10 times faster than it would against the unpartitioned table.



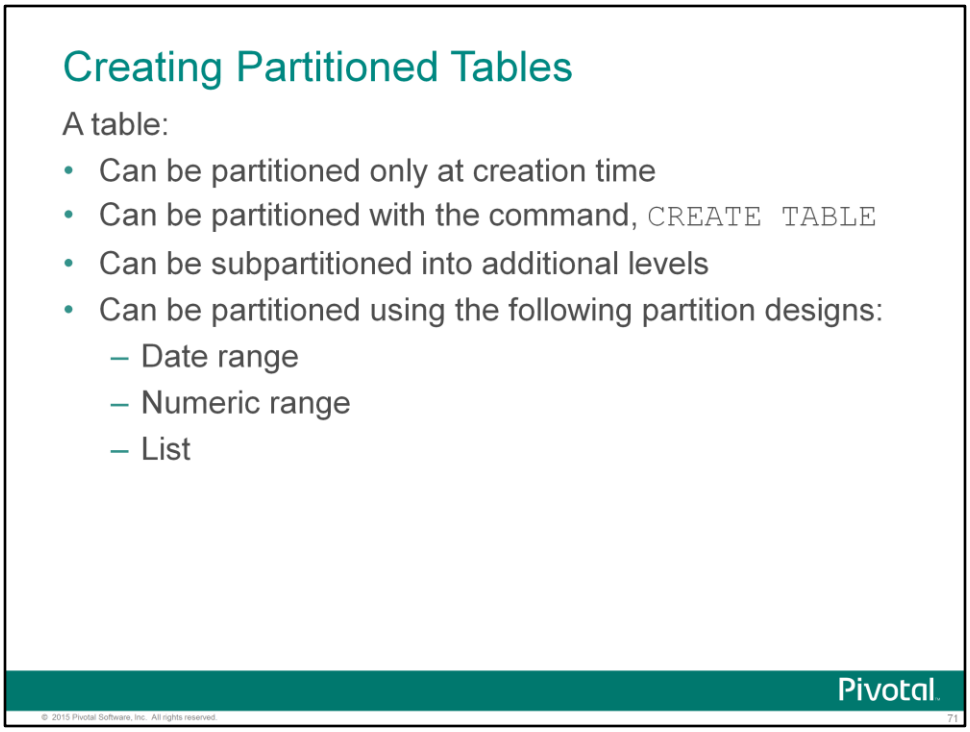
When you partition a table in Greenplum Database, you:

- Create a top-level or parent-level table
- Create one or more levels of sub-tables, or child tables
- Push the data directly into the lowest level of the table

Internally, Greenplum Database creates an inheritance relationship between the top-level table and its underlying partitions.

Each partition is created with a distinct `CHECK` constraint. This limits the data that the table can contain. The `CHECK` constraints are also used by the query planner to determine which table partitions to scan in order to satisfy a given query predicate.

Partition hierarchy information is stored in the Greenplum system catalog so that rows inserted into the top-level parent table appropriately propagate to the child table partitions.

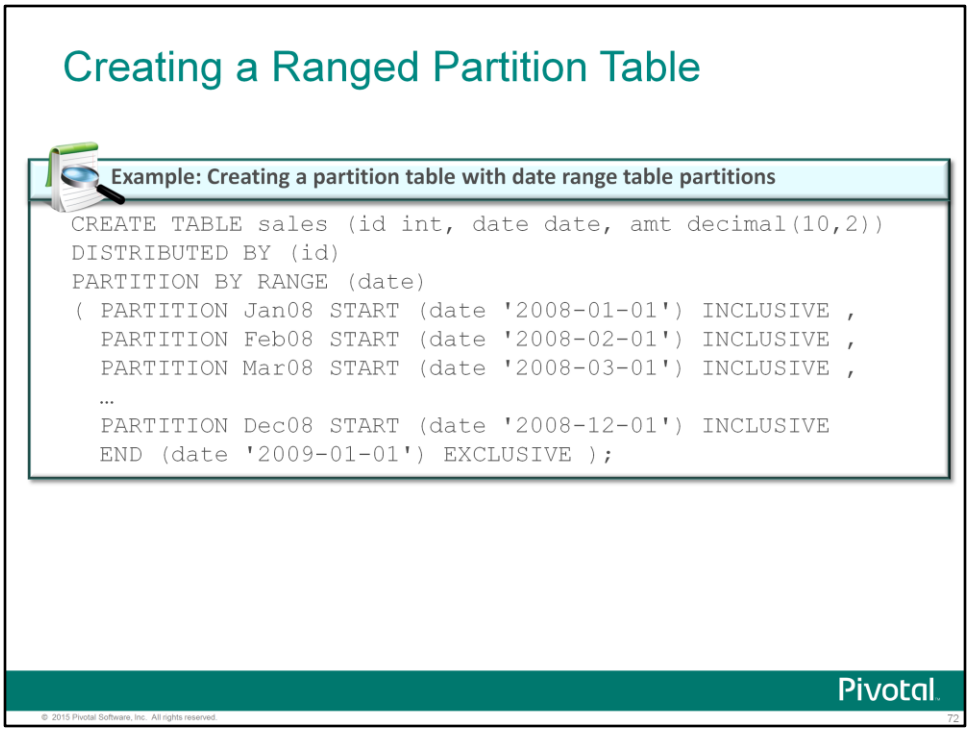


A table can only be partitioned at creation time using the `CREATE TABLE` command.

The first step in partitioning a table is to decide on the partition design (date range, numeric range, or list of values) and choose the column(s) on which to partition the table. Decide how many levels of partitions you want. For example, you may want to date range partition a table by month and then further subpartition the monthly partitions by sales region.

You can partition a table using:

- A date range
- A numeric range
- A list



You can define a date range or a numeric range for a partitioned table:

- A date range uses a single date or timestamp column as the partition key column. You can use the same partition key column to further subpartition a table if necessary. For example, you can partition by month and then subpartition by day. When date partitioning a table, consider partitioning by the most granular level you are interested in. For example, partition by day and have 365 daily partitions, rather than partition by year then subpartition by month then subpartition by day. A multi-level design can reduce query planning time, but a flat partition design will execute faster at query run time.
  You can have Greenplum Database automatically generate partitions by giving a `START` value, an `END` value, and an `EVERY` clause that defines the partition increment value.
  By default, START values are always inclusive and END values are always exclusive.
  Note: You can declare and name each partition individually.

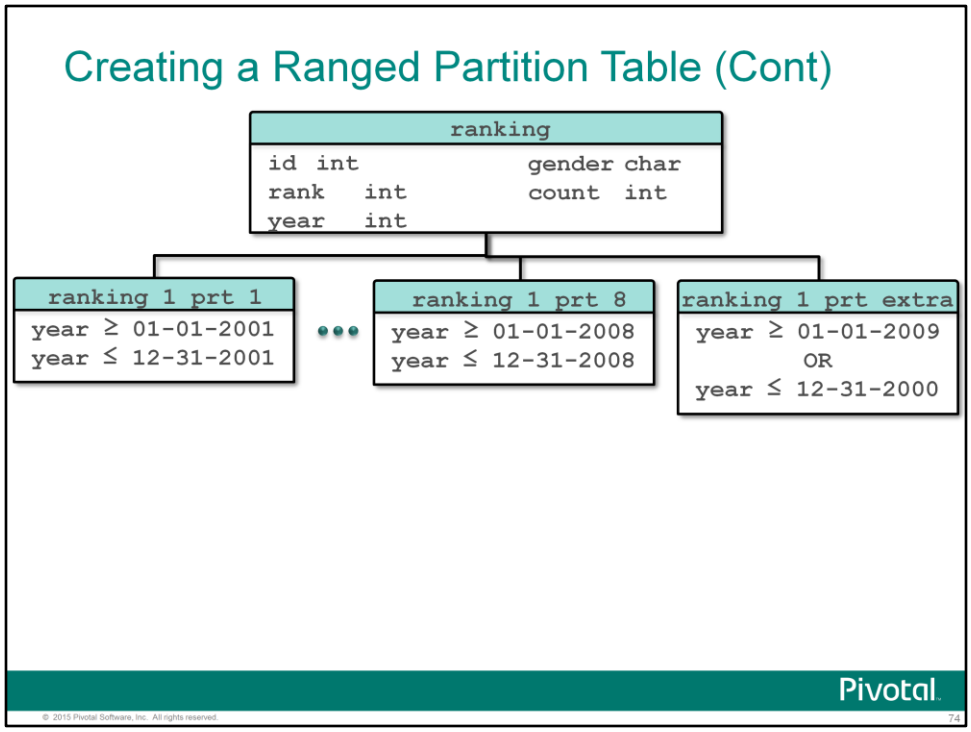
Creating a Ranged Partition Table (Cont)

Example: Creating a partition table with numeric range table partitions

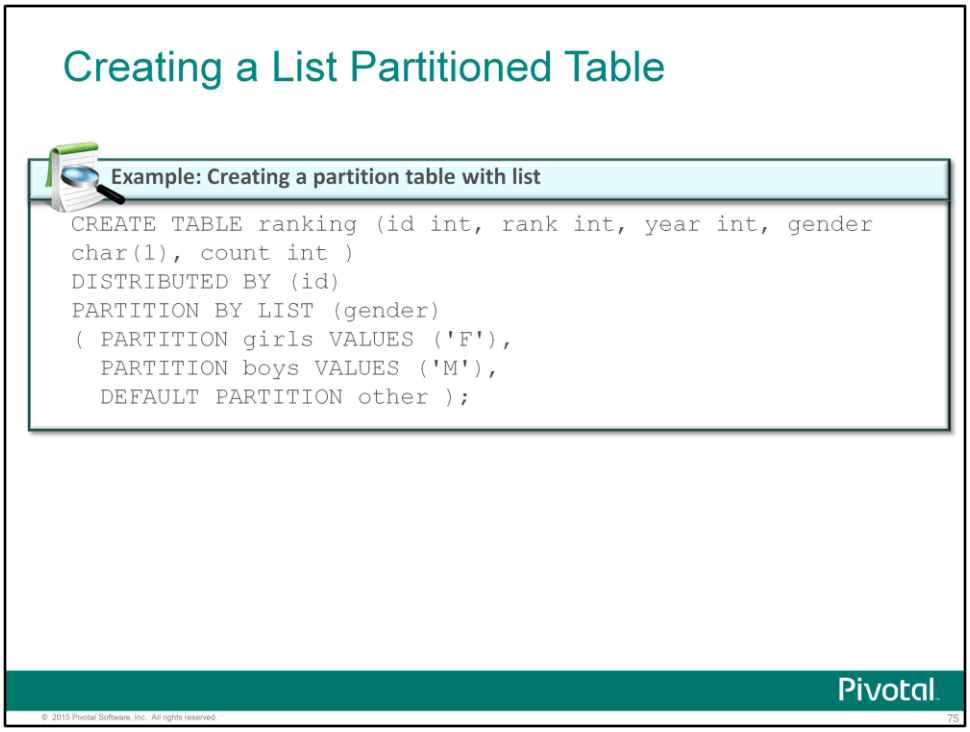
```
CREATE TABLE ranking (id int, rank int, year int, gender
char(1), count int)
DISTRIBUTED BY (id)
PARTITION BY RANGE (year)
( START (2001) END (2008) EVERY (1),
  DEFAULT PARTITION extra );
```

Pivotal

- A numeric range uses a single numeric data type column as the partition key column.

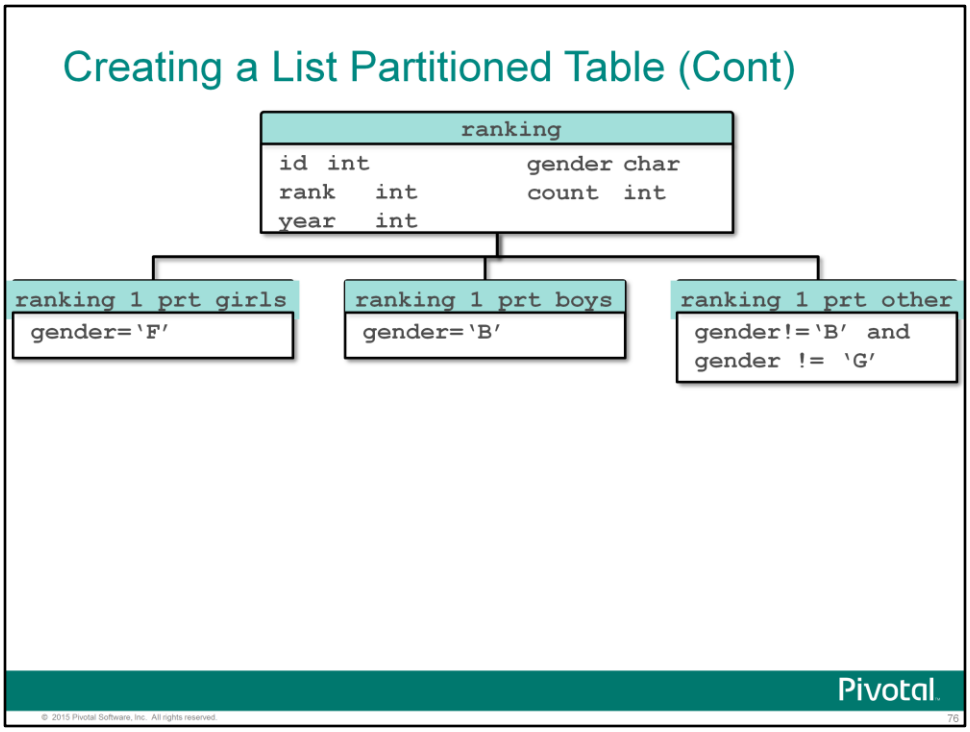


- The child tables created are created with the name, `ranking_1_prt_#`, where # represents the sequential number assigned to the table. In this example, the first child table created for the year 2001 is assigned the name, `ranking_1_prt_1`, while the last child table created is `ranking_1_prt_8`. The default child table created is `ranking_1_prt_extra`.

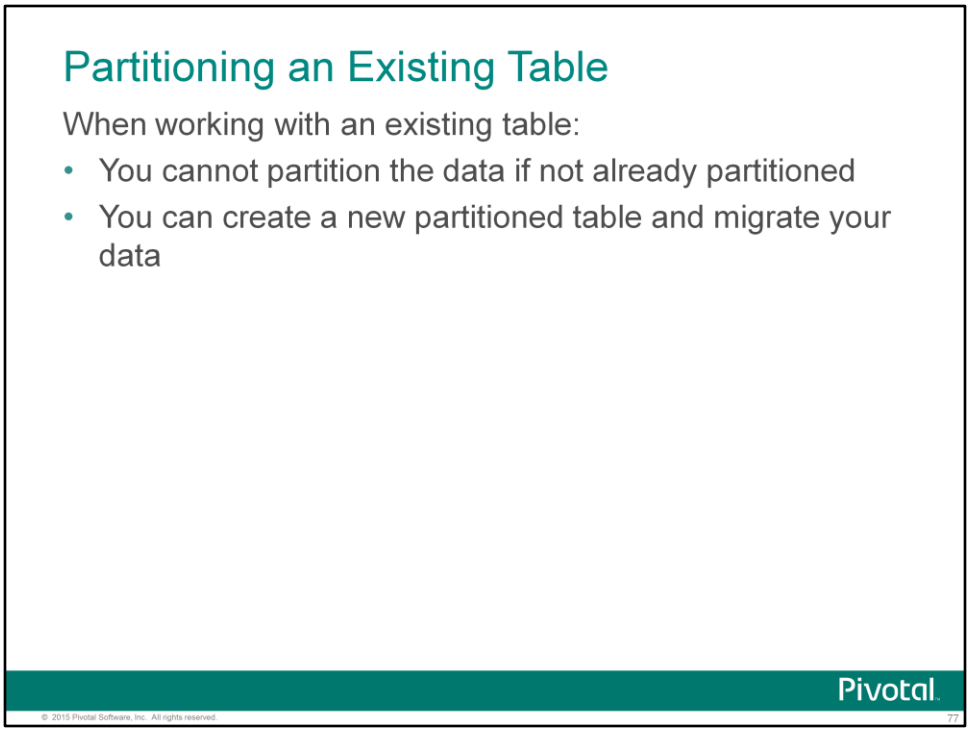


A list partitioned table:

- Can use any data type column that allows equality comparisons as its partition key column.
- Can have a multi-column, or composite, partition key, while a range partition only allows a single column as the partition key.
- Requires you declare a partition specification for every partition, or list value, you want to create.



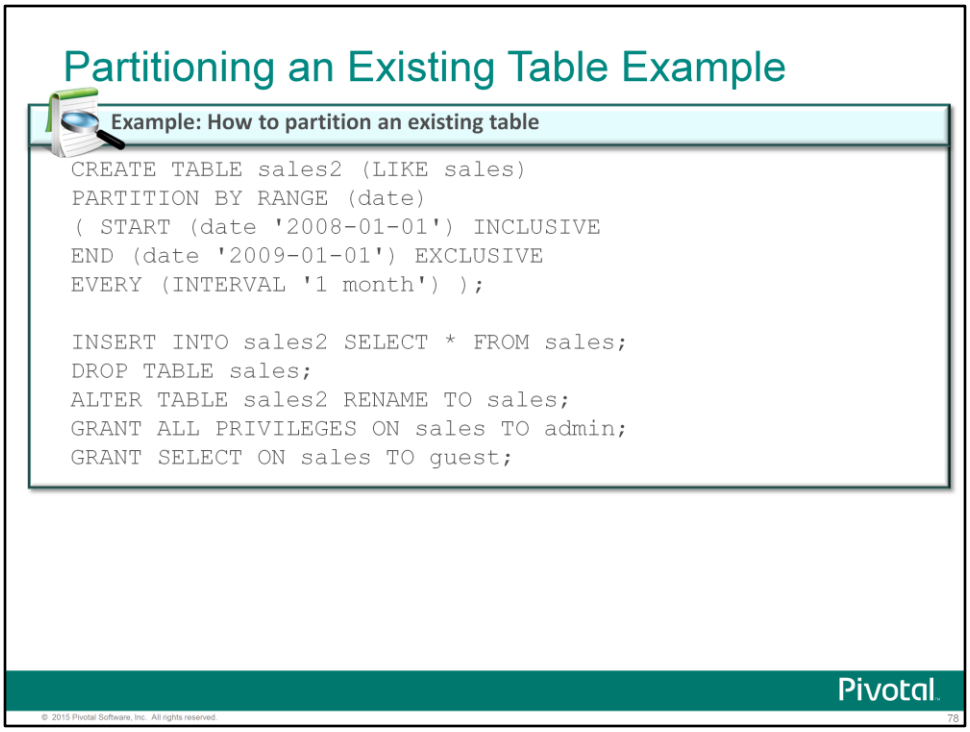
For the example provided, each list was assigned a specific name for a specific value. All entries not defined will be assigned to the default table.



It is not possible to partition a table that has already been created. Tables can only be partitioned at `CREATE TABLE` time.

To partition an existing table, you must:

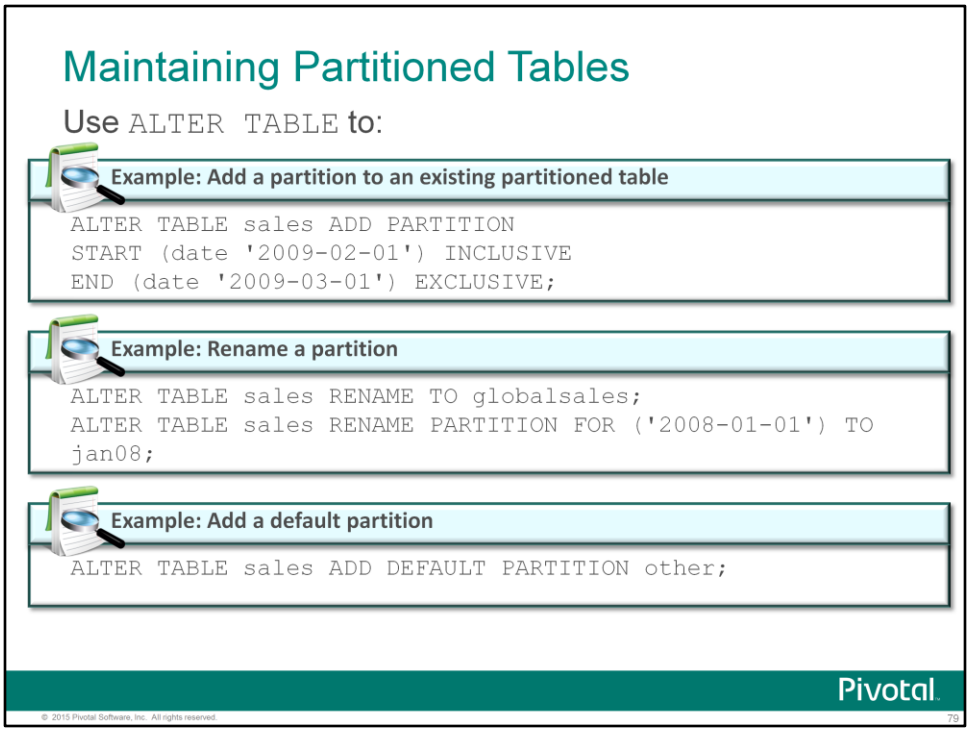
1. Recreate the table as a partitioned table.
2. Reload the data into the newly partitioned table.
3. Drop the original table and rename the partitioned table to the original name.
4. Grant any table permissions to the new table that were granted on the original table.



The example shown highlights one method of partitioning an existing table. To perform this action, a second table is created based on the definition of the table to be partitioned. While creating the table, you define the partition for the table, in this case, using a date range.

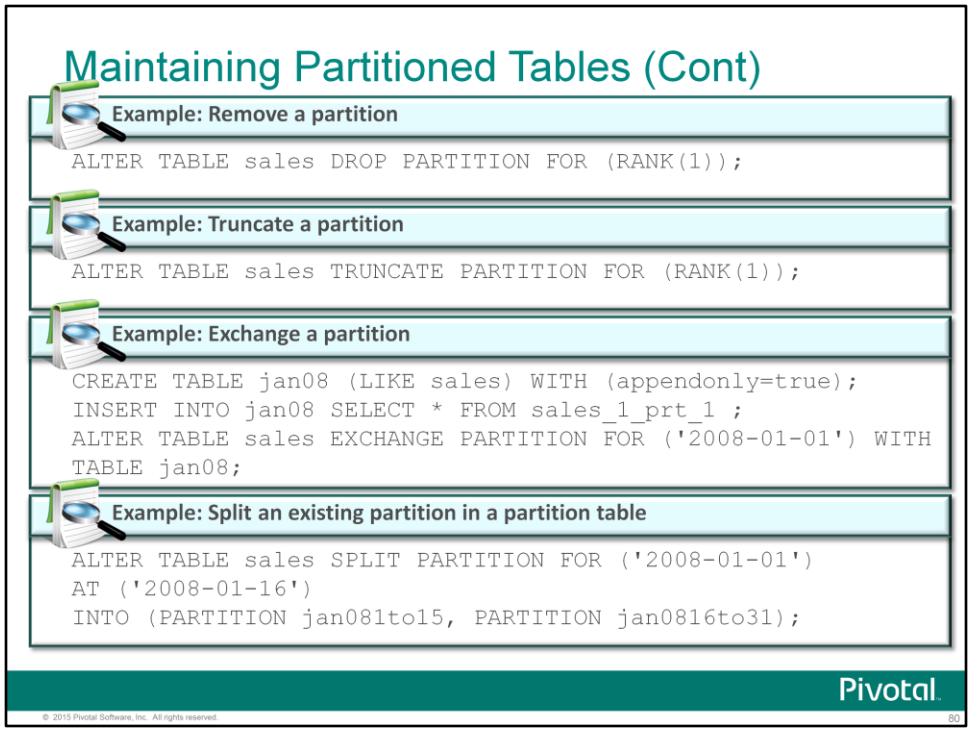
Once the table has been created, insert data from the previous table into the newly created and partitioned table. The data will be automatically segregated based on the partition you have defined.

Next, drop the original table and change the name of the new table to the name of the original table and grant all appropriate privileges to the table.

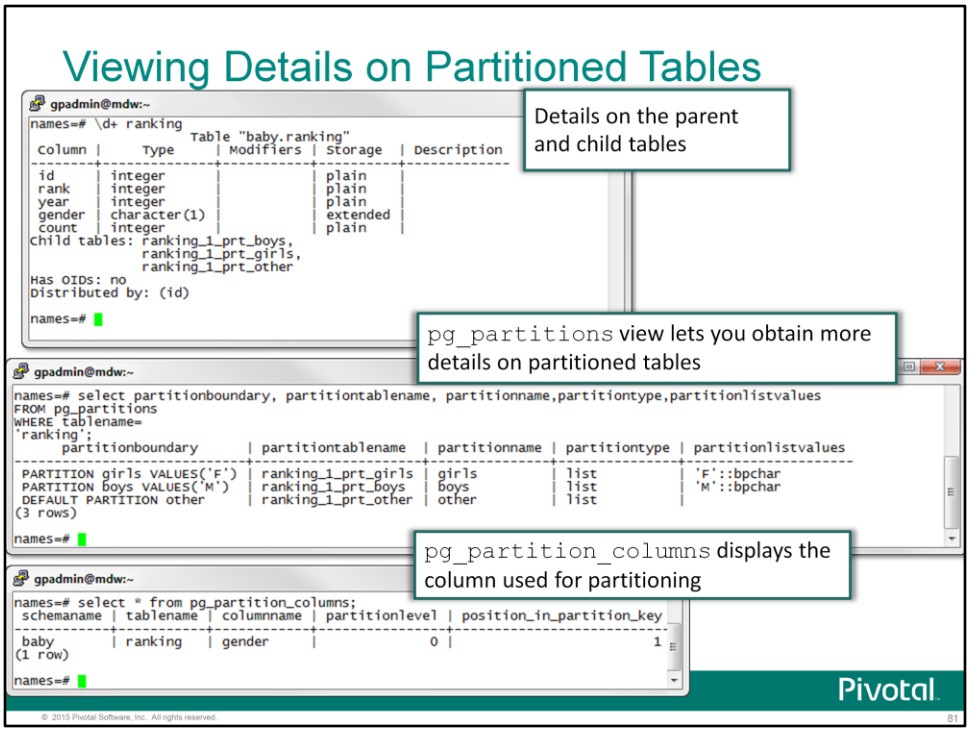


You maintain a partitioned table using the `ALTER TABLE` command to:

- Add a partition to an existing partition design. You can:
  - Add a partition to a table that already has subpartitions defined using the `ADD PARTITION` clause
  - Add a partition to a table that does not have subpartition templates defined using the `ADD PARTITION` clause followed by the `SUBPARTITION` statements
  - Add a subpartition using `ALTER PARTITION FOR (…) ADD PARTITION …`
- Rename a partition by renaming the top-level parent table to update the child table created or by renaming the partition directly. You cannot directly rename the child table name.
- Add a default partition to an existing partition design. If the partition design is multi-level, you must define a partition default for each level.



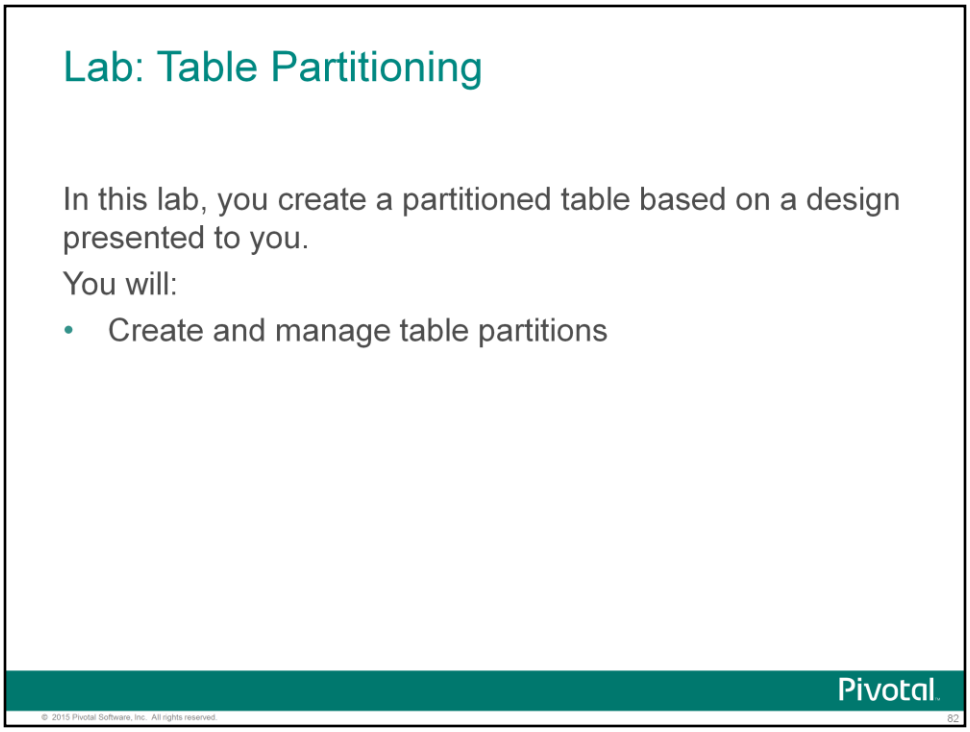
- Drop a partition using the `DROP PARTITION` clause. When you drop a partition that has subpartitions, the subpartitions and all data are automatically dropped.
- Truncate a partition using the `TRUNCATE PARTITION` clause. When you truncate a partition that has subpartitions, the subpartitions are automatically truncated.
- Exchange a partition using the `EXCHANGE PARTITION FOR` clause. Exchanging a partition involves swapping in another table in place of an existing partition. You can only exchange partitions at the lowest level of the partition hierarchy. Only partitions that contain data can be exchanged.
- Split a partition using the `INTO` clause. Splitting a partition involves dividing an existing partition into two. You can only split partitions at the lowest level of the partition hierarchy. Only partitions that contain data can be split. The split value you specify goes into the latter partition.



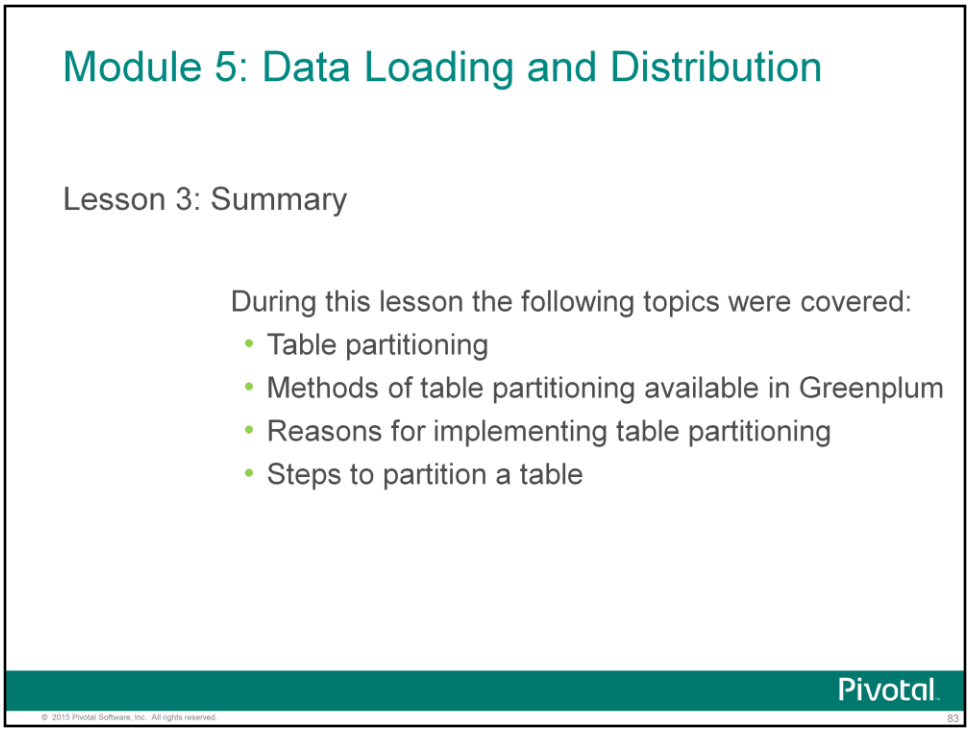
Some partitioning information can be obtained on tables with the `\d+` PSQL command. All child tables are displayed by their names.

You can obtain additional information by viewing the `pg_partitions` view. In the example shown, the `partitionboundary` column displays how the partitions are divided and the values used to create the specific partition. Next, the partition table name is displayed with the `partitiontablename` column. The partition name follows with the `partitionname` column, followed by the partition rank. The partitiontype column displays whether the partition is a list or ranged column. As it is a list column, the partitionlistvalues is used to display the values and the data type. The `partitionrangestart` and `partitionrangeend` columns provides similar information for ranged tables.

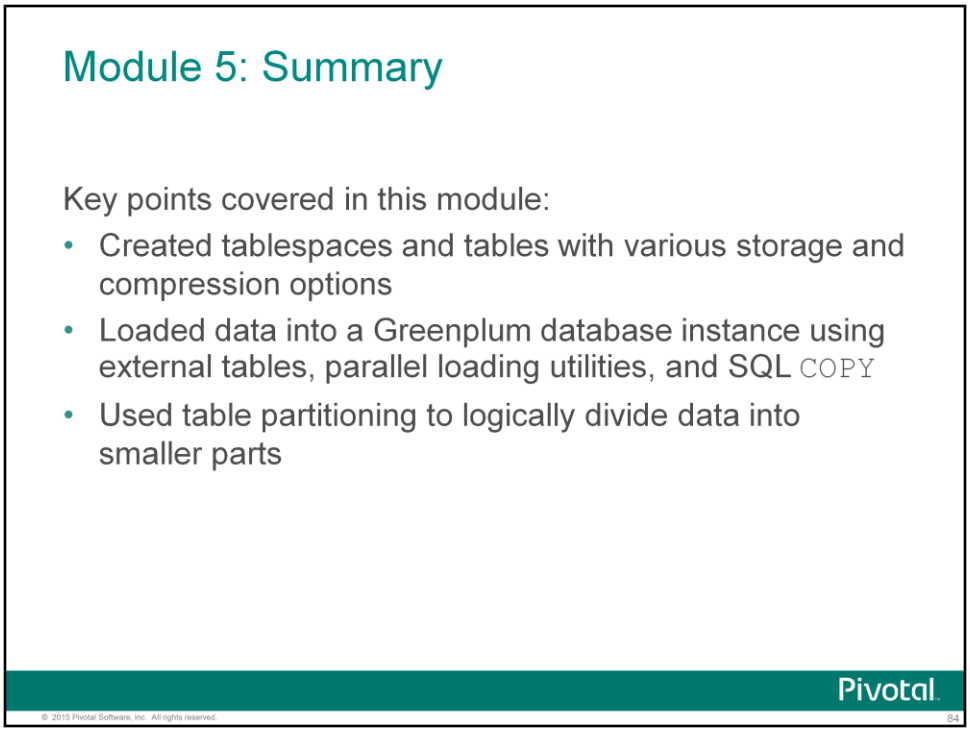
The `pg_partition_columns` view displays the column used for partitioning. The `position_in_partition_key` column is used to display the position of the column in a multi-column composite partition key found in list partitions.



In this lab, you create a partitioned table based on a design presented to you. You then modify the table design.



This lesson covered how to partition a table, where the data is stored within the table, and the various table partition methods. It also discussed why you would want to partition your table, the performance improvements that you could see with partitioning, and the potential impacts. The steps to create and manage partitions were provided.



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

- Create and manage tablespaces and tables of varying storage methods
- Load data into a Greenplum database instance using external tables, parallel loading utilities, and SQL `COPY`
- Use table partitioning to logically divide data into smaller parts.