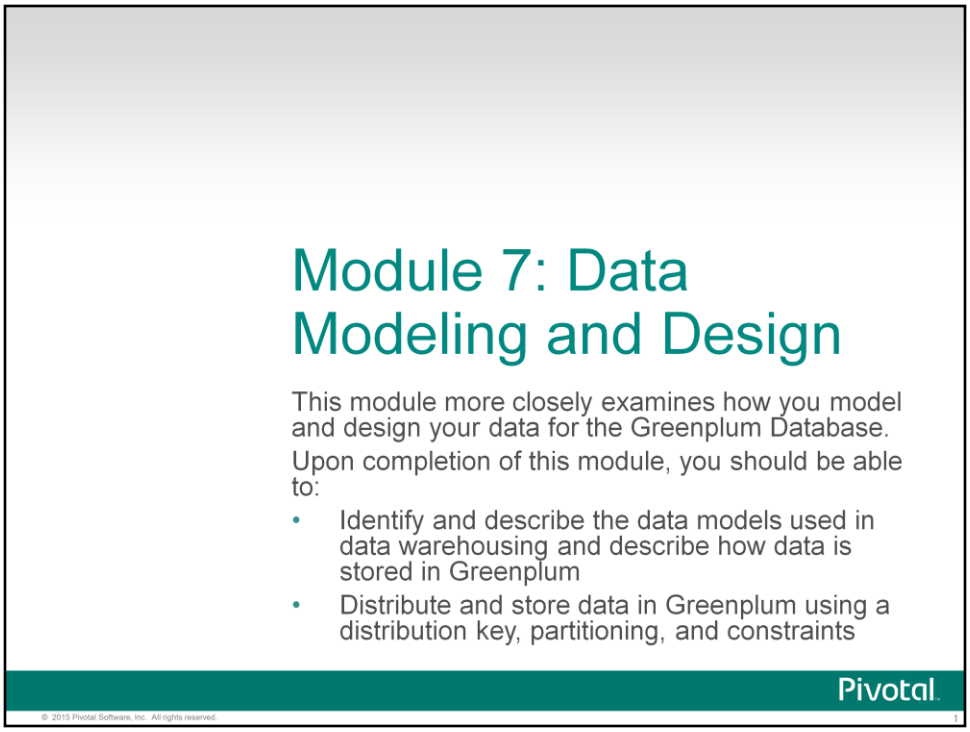
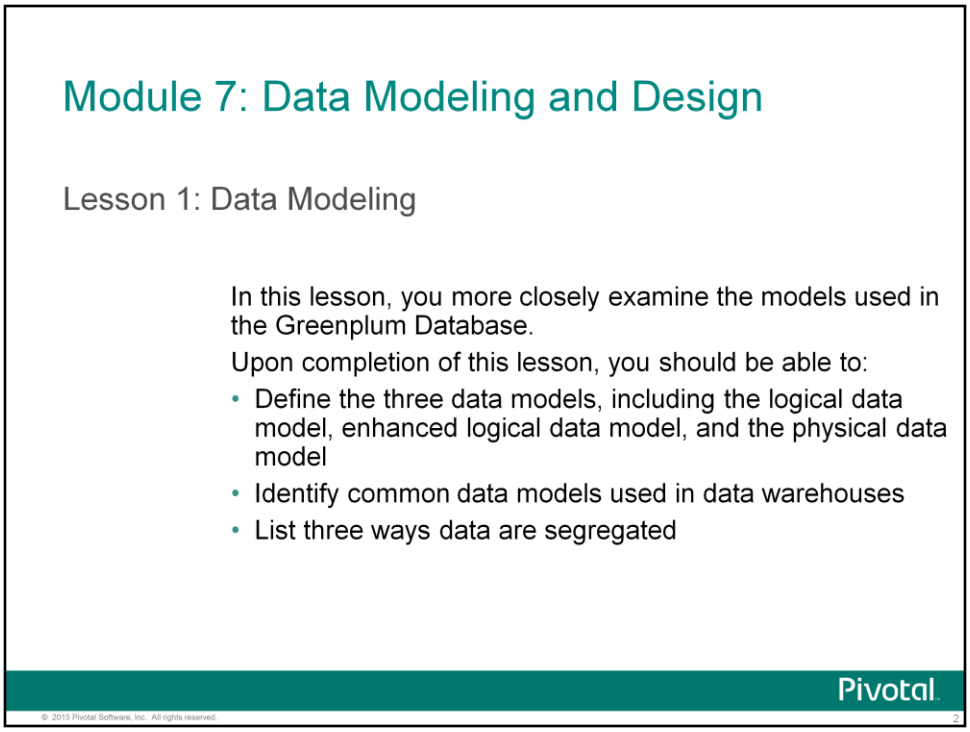


Key decisions must be made on how data is designed for the Greenplum Database. These decisions can affect business decisions that are based on data and determine the overall performance for accessing data.

In this module, you will:

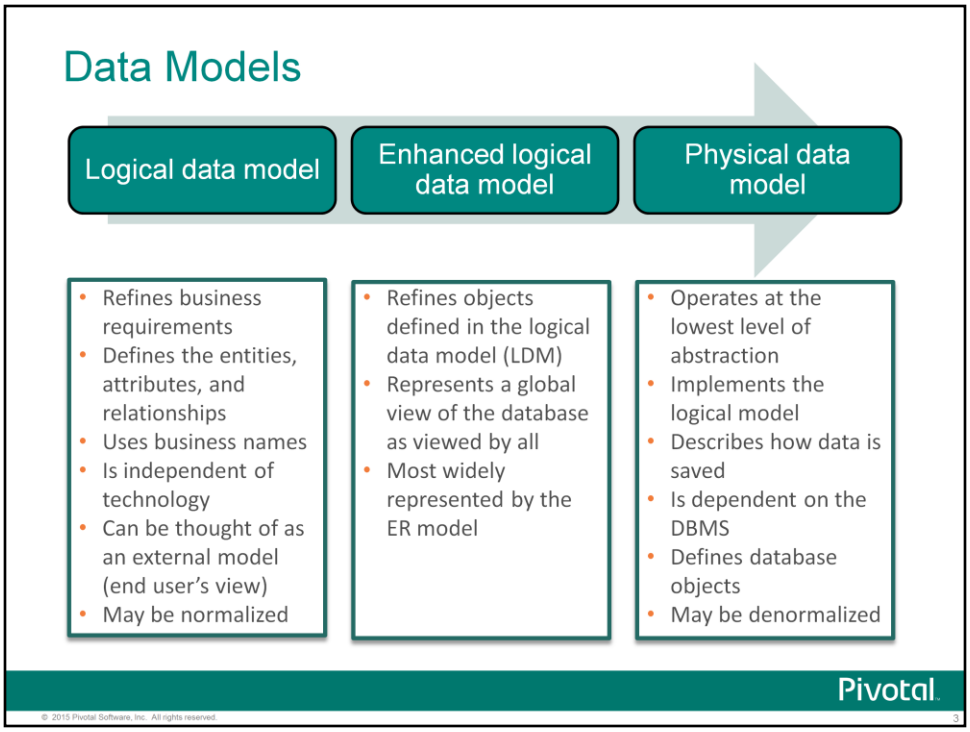
- Identify and describe the data models used in data warehousing and describe how data is stored in Greenplum
- Distribute and store data in Greenplum using a distribution key, partitioning, and constraints



Data modeling is a key area of data warehouse design. Three models were presented earlier, with benefits for OLAP and OLTP. It is important to gather requirements and build the data model before implementing it. A poorly designed database can have repercussions beyond performance issues, affecting the expected outcome of queries, and therefore, affecting business decisions.

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

- Define the concepts, logical, enhanced logical, and physical data models and implement appropriate data models for Greenplum.
- Identify the common data models used in data warehouses.
- List three ways in which data are segregated.

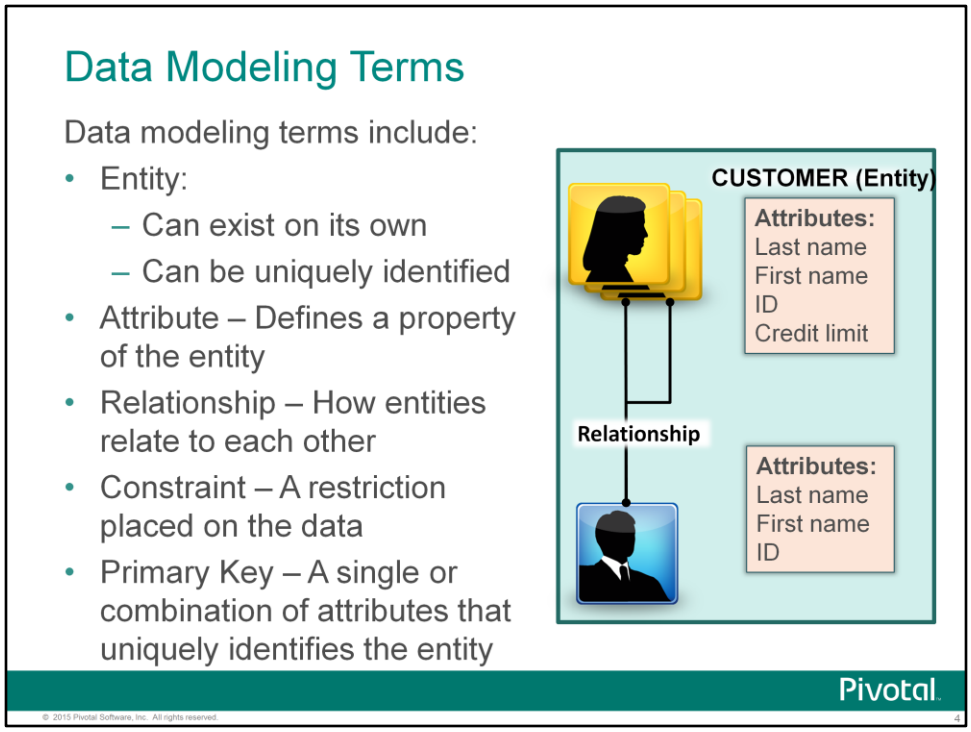


Data modeling:

- Is the first step in designing the database and refers to the process of creating a specific data model for a determined problem.
- Is an iterative process where you continuously refine the requirements and the details of the data model.
- Acts as a communication tool to show, in great detail, how business requirements translate into business decisions, and the business rules each organization creates and uses can affect another organization or the business overall.
- Shows the different ways in which data is viewed by individuals and organizations.
- Provides the basic building blocks for entities, attributes, relationships, and constraints .
- Realizes a physical model where data is represented at the lowest level of abstraction.

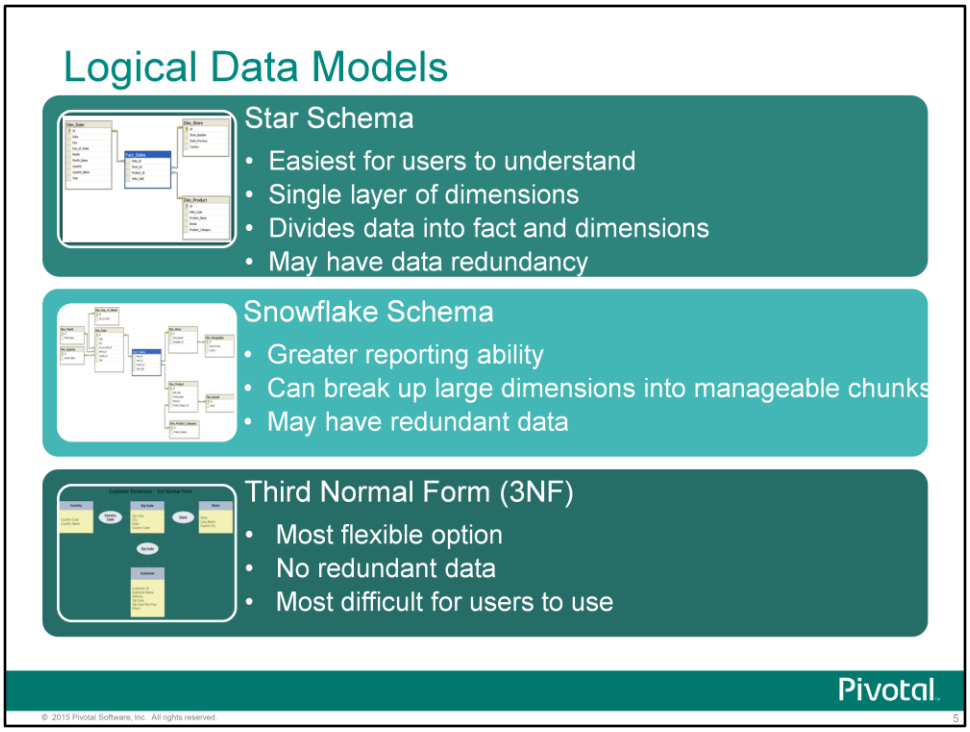
Data design flows from creating and refining the logical business model, where you define the basic building blocks based on the requirements gathered from stakeholders, to an enhanced logical model, where you further refine the logical data model with detailed information on the objects, to the physical design, where you implement the logical data model.

Note: Normalization is concerned with relationships. It is a process for evaluating and correcting table structures or models to minimize data redundancies. Denormalized data is not concerned with redundancies.



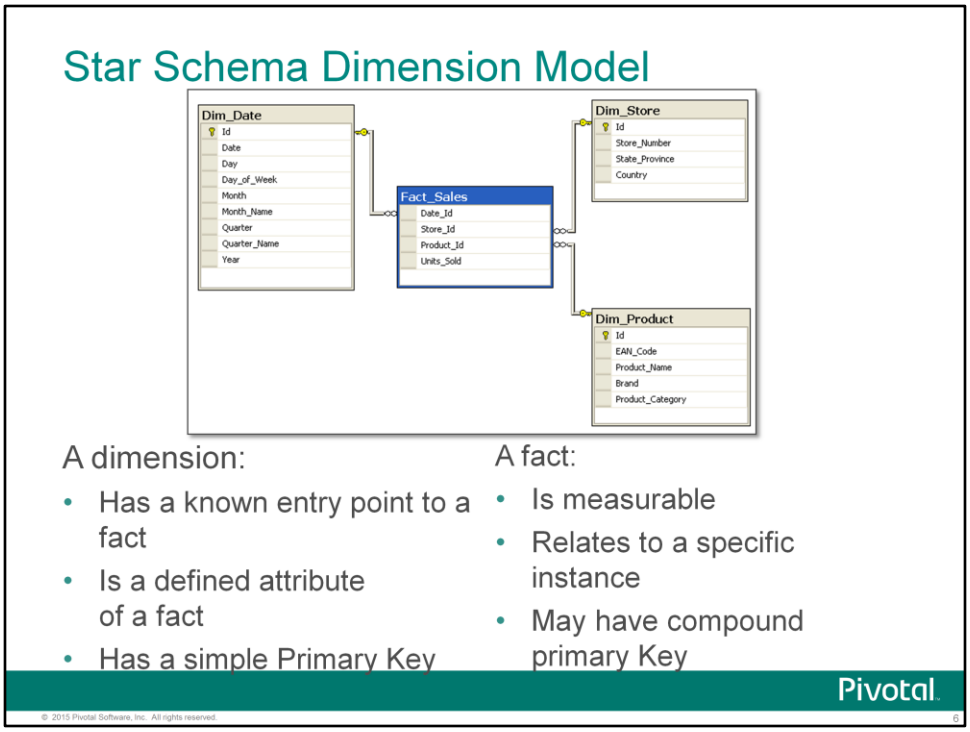
The following terms are commonly used in data modeling and describe the basic building blocks of the model:

- **Entity** – This describes the unique person, place, thing, or event about which the data are collected and stored. It can exist on its own.
- **Attribute** – The attribute is a characteristic or description of the entity. It defines the property of that entity.
- **Relationship** – The relationship shows how the association between entities and how they relate to each other.
- **Constraint** – The constraint is a restriction placed on the data, helping to ensure integrity. For example, the credit limit for the customer must be entered in numbers greater than or equal to 10,000.
- **Primary key** – This term describes the attribute or combination of attributes that uniquely identifies the entity. For example, the combination of first and last name may be enough to describe the customer. However, it is more likely that the ID attribute is sufficient and unique enough to identify the customer, as long as there is a constraint on that attribute requiring that no two IDs are the same.



One of the more commonly used models in a data warehouse and OLTP systems include:

- **A star schema** – This is one of the easiest dimensional models to understand as it represents data as a single layer of dimensions, dividing data into facts and dimensions.
  There can be one or more fact tables represented in this model. The fact table contains measures of interest, while the dimension or lookup tables, which connect to the fact table, provides detailed information on the attributes. Lookup tables do not have a relationship with each other, instead, relying on their relationship to the fact table.
- **A snowflake schema** – The snowflake schema dimensional model is an extension of the star schema and offers greater reporting capabilities. It divides the large fact table into more manageable portions, by normalizing the dimensional table into multiple lookup tables. Each lookup table may represent another level in the dimensional hierarchy. This type of model may incur redundant data.
- The third normal form (3NF) – The 3NF form.

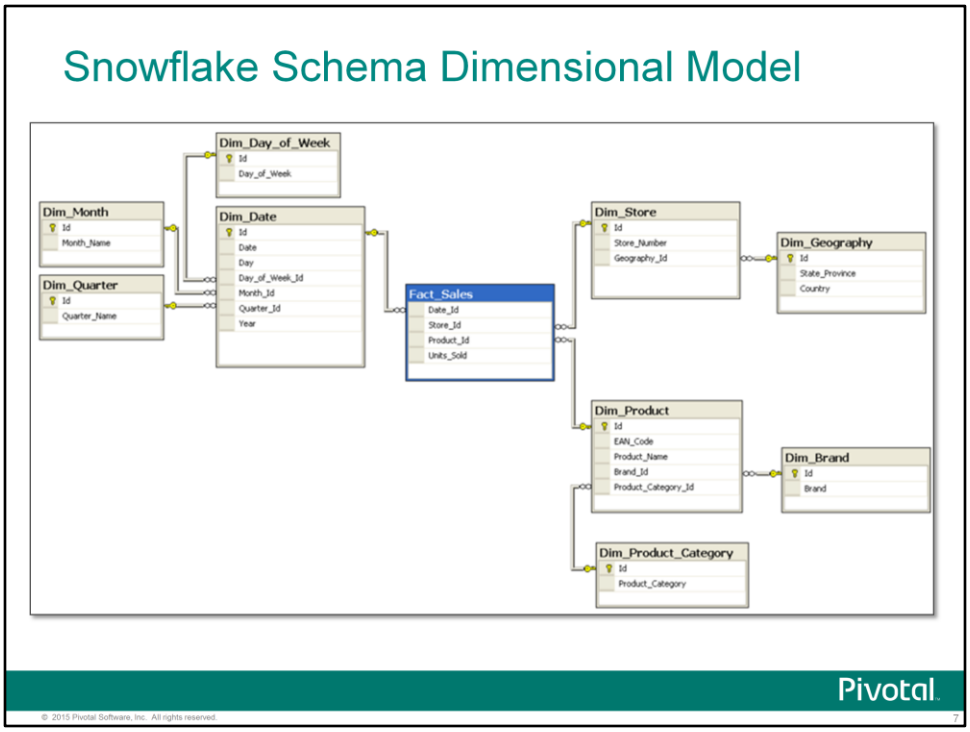


The dimension table:

- Also known as a lookup table, has a known entry point to a fact.
- Provides a description of the fact. For example, the `Dim_Store` dimension table contains descriptions of the store, including the store number, ID, and location.
- Has a single primary key for each record. This primary key uniquely identifies the store in the `Dim_Store` table.

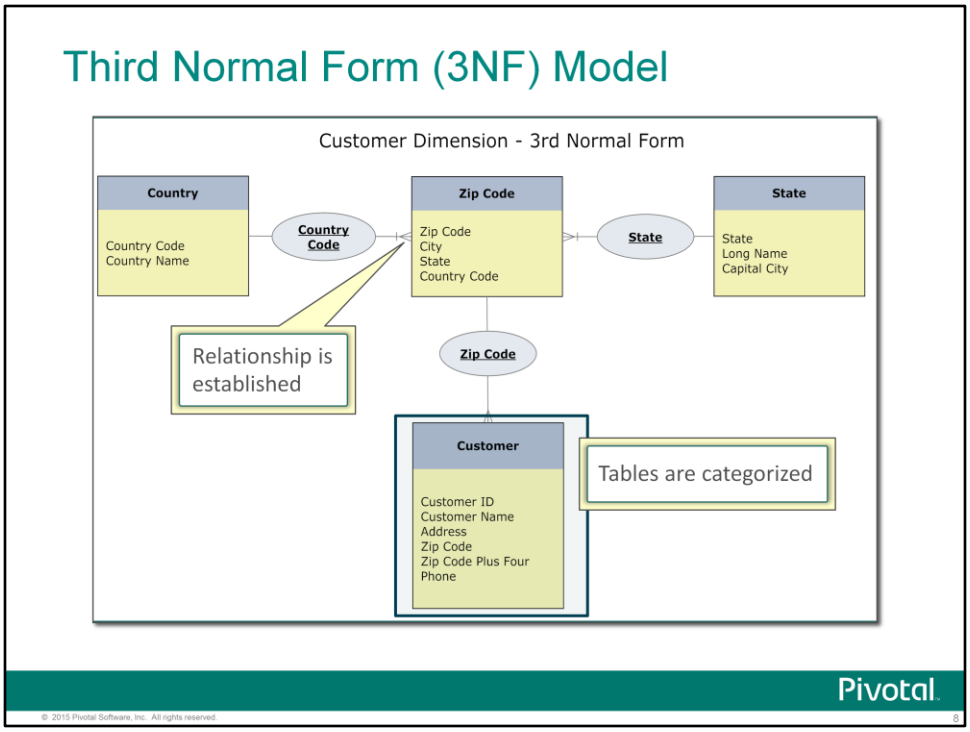
The fact table:

- Combines the information provided in the lookup tables to provide a measurable account of a unique record.
- The fact table in this example, Fact_Sales, identifies the date, store ID, product ID, and number of units sold for each unique record.
- A unique record can be determined by a combination of primary keys, or a compound primary key. It may not be enough to uniquely identify a transaction or sale just by the `Date_Id`. It may require that all four fields in the table uniquely identify a sale.



In the snowflake schema, the dimensional tables are further normalized into multiple lookup tables. Your data is no longer represented in a single dimension, but is now shown as a dimensional hierarchy.

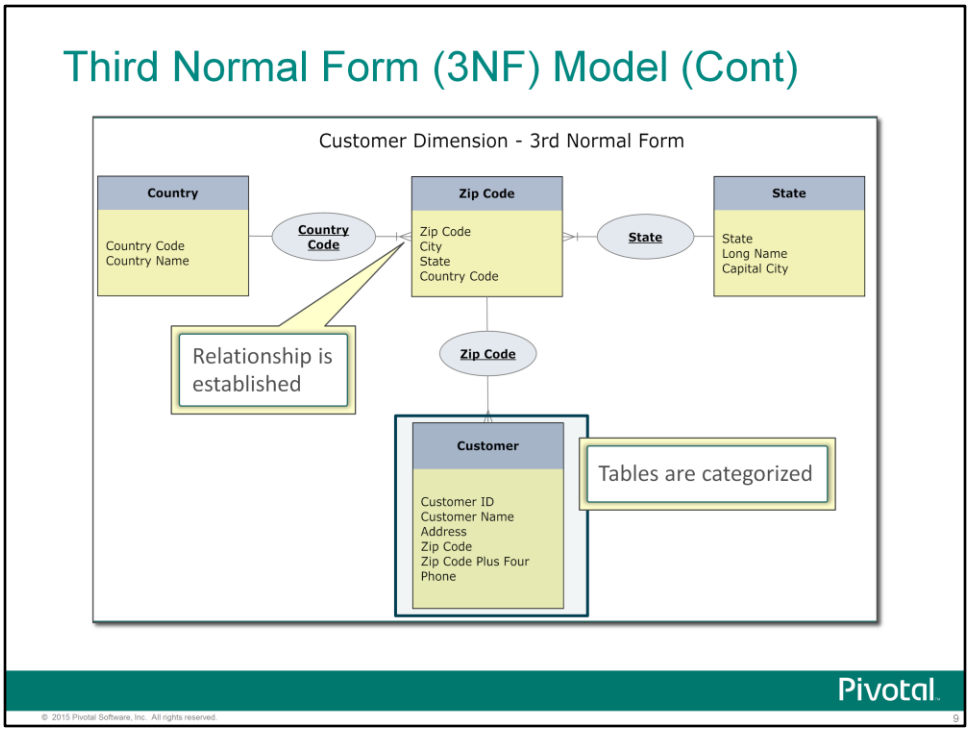
Snowflake schema offer some improvement in query performance due to minimized disk storage requirements and joining on smaller tables. This comes at a price however. Additional maintenance is required due to the increased number of lookup tables generated by this type of model.



The third normal form (3NF) design normalizes data to remove redundancy and groups data together by subject areas or categories. This can be difficult for users to understand, but it is more suited to capturing complex the complexities of real world entities and the resulting data. This type of model can be used to help identify patterns and trends instead of simply providing reporting support and support for well established relationships.

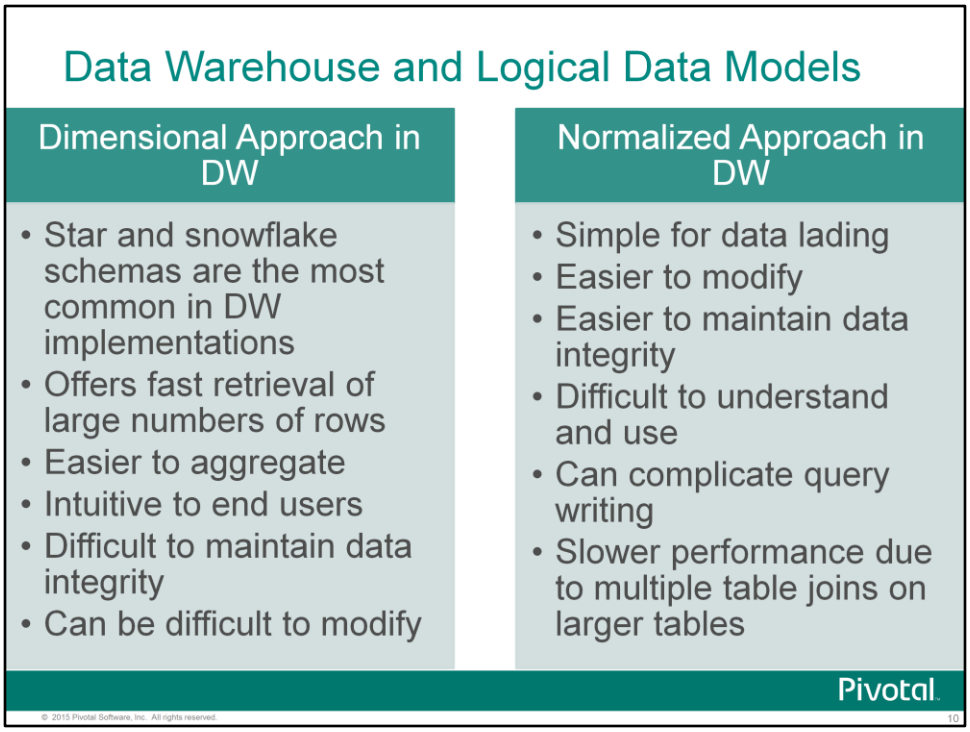
Getting this form can sometimes be an iterative process, but can be accomplished by going through lower forms:

- 1NF requires the existence of a primary key and the identification of dependencies. The primary key can be a combination of multiple fields. This primary key must be able to uniquely identify each record. Dependencies are fields that are dependent, in some way, on the primary key.
- 2NF requires the table first be in 1NF and there are no partial dependencies. A partial dependency is when an attribute is dependent on a portion of the primary key. If a primary key consists of three fields, a partial dependency occurs if another field is dependent on only one of those three fields. A table with a primary key that contains only a single attribute is automatically in 2NF.
- 3NF requires that the table be in 2NF and contains no transitive dependencies. A transitive dependency exists when an attribute, other than one involved in the primary key, is dependent on another attribute that is not a part of the primary key.

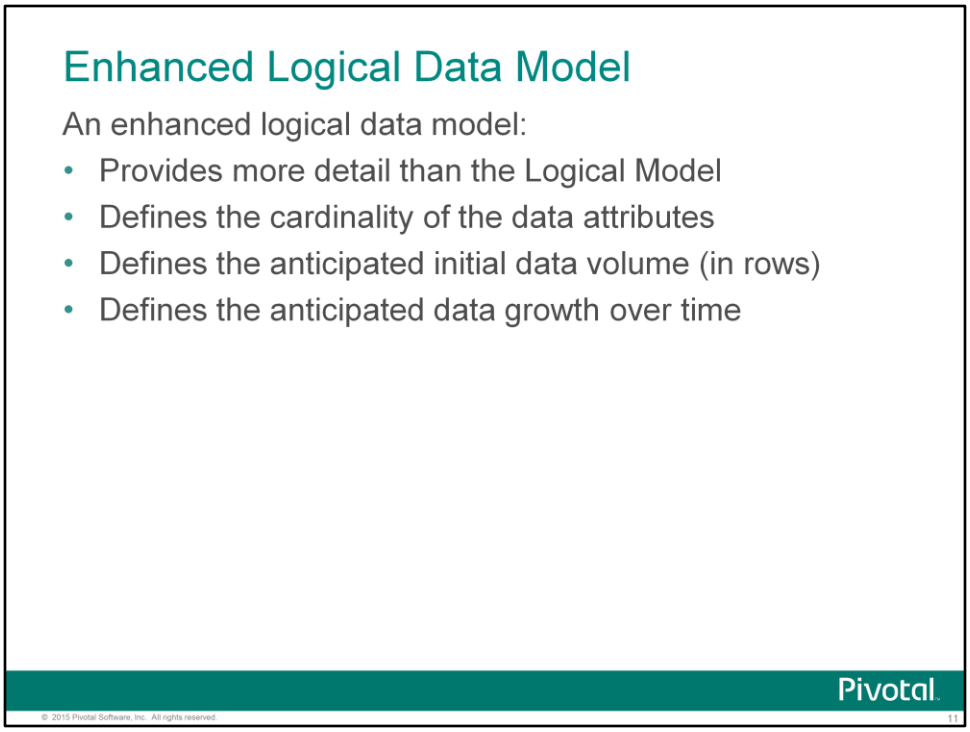


The example on the slide shows the following relationships:

- A country can have multiple zip codes.
- A state can have multiple zip codes.
- In a zip code, you can contain multiple customers.



There are benefits to using one type of model over another. Dimensional models can offer better query performance while normalized models can provide insight into complex data. Real-time data may be easier to implement in 3NF than in dimensional models, but dimensional models offer greater reporting capabilities.



Often represented as entity relationship (ER) diagrams, the enhanced logical data model extends the logical data model by providing more details about the entities defined earlier, including:

- Defining the cardinality of data attributes, which is the relationship of a given table to another. Cardinality can be defined as a one-to-one relationship, one-to-many relationship, or many-to-many relationship.
- Defining the anticipated initial volume of data expected to be loaded into systems.
- Defining the anticipated growth of data over a period of time.

Enhanced Logical Data Model Example

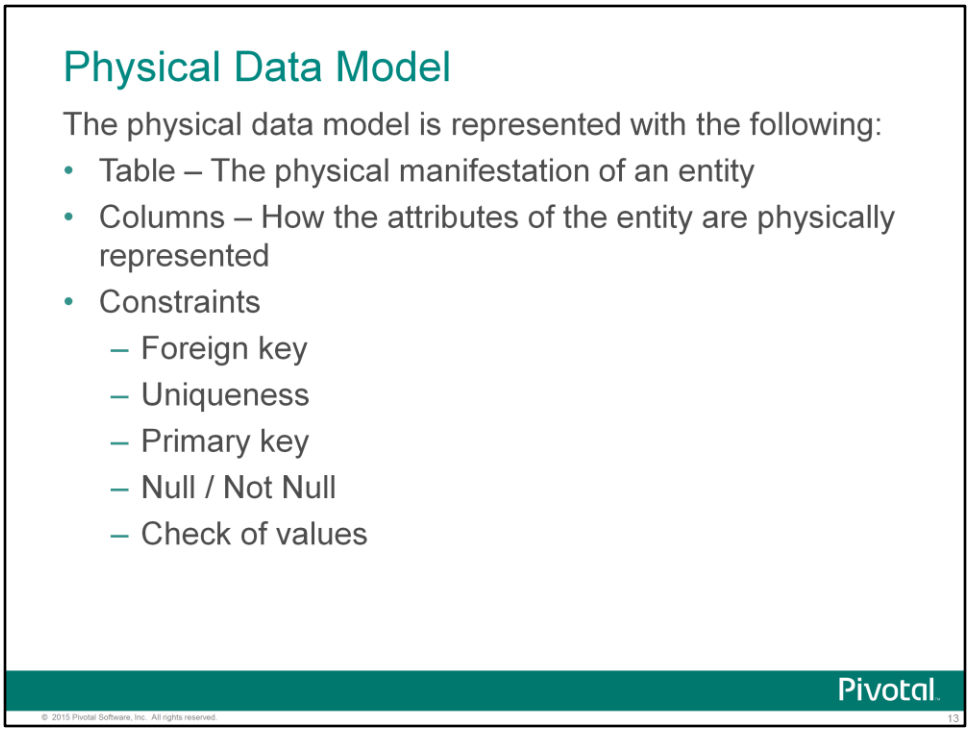
The following is an example of the enhanced logical data model:

Entity: Store
Source: SOLS (Stores On-Line System)
stores table
Row Count at Source: 35
Anticipated Annual Growth Rate: 100

Attribute	Source	Uniqueness	Data Type	Data Example
Store ID	Stores	Unique	Small Integer	1,2 … N
Store Name	Stores	Non	Character	Tom's Groceries
…				
Has Pharmacy	Stores	Non	Character	T,F

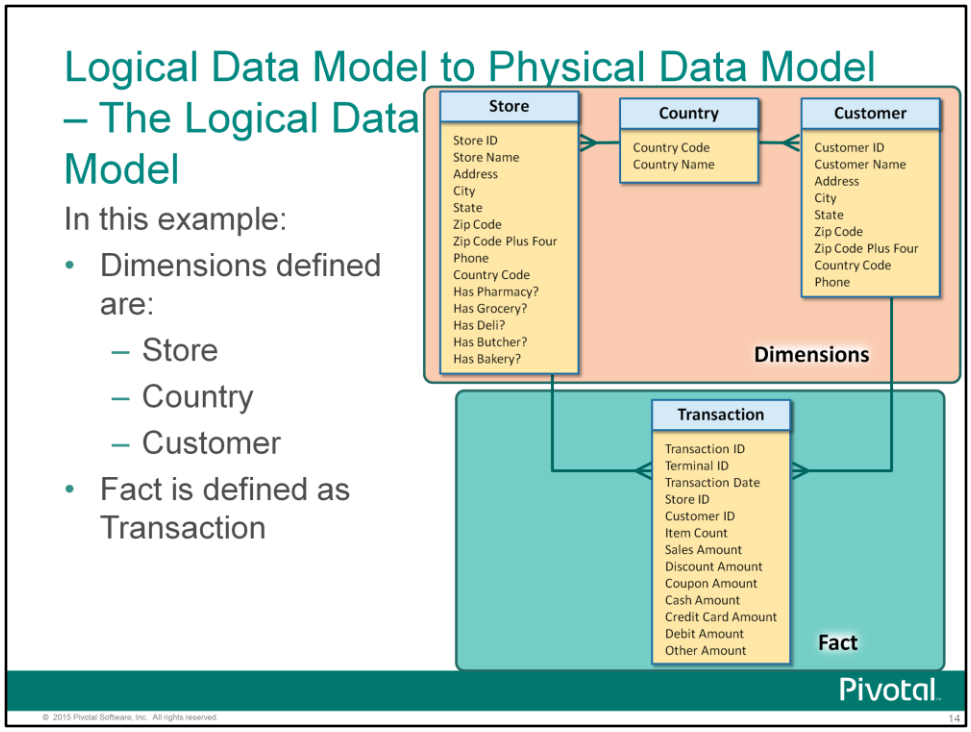
In the example shown, the detailed information is provided about the entity, `Store`, including the source for this data, the anticipated row count and the anticipated growth rate for that object.

The entity is described by its attributes in the table at the bottom of the slide. Data types and examples of the expected values are included in the model.



The physical data model defines the objects that will be created in the database. This includes:

- **Table** – A table is a physical representation of the entity defined in the logical data model.
- **Columns** – Each of the attributes defined for the entity are displayed as columns in the table.
- **Constraints** – Constraints are any restrictions on the attributes as they are entered in the table. Constraints can be any of the following:
    - Foreign keys which shows the relationship from one table to another.
    - Uniqueness where you specify that a value must be unique.
    - Primary key where you define the set of attributes that uniquely identify each record.
    - Null or not null where the column may or may not have a null value.
    - Check of values where you define acceptable values for the column.



This example shows how a logical data model in 3NF is implemented in a physical data model.

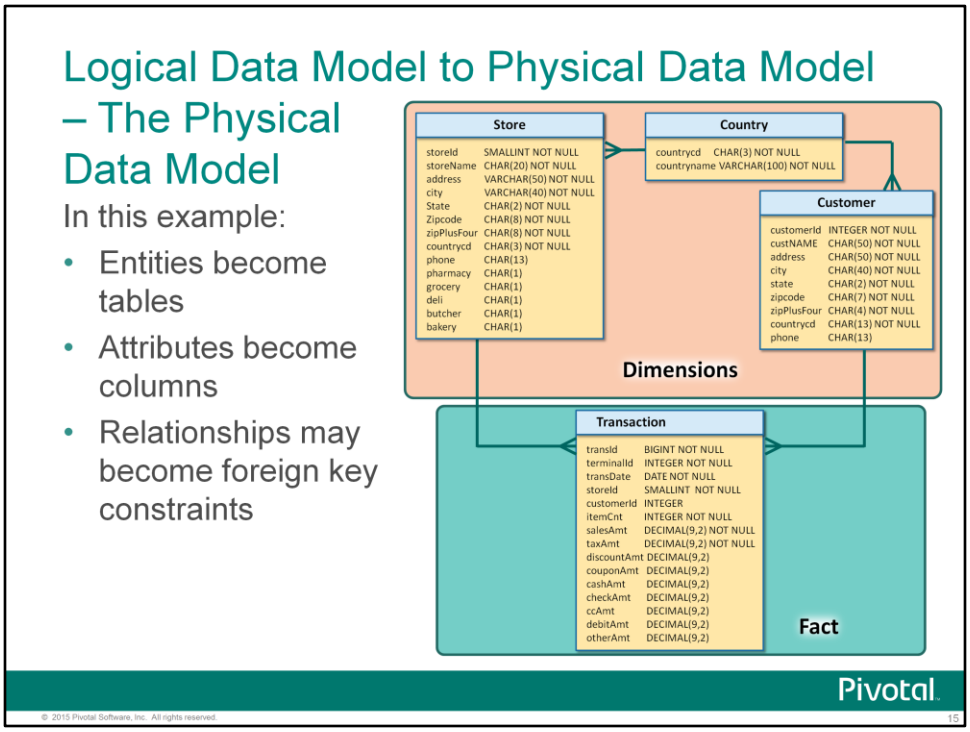
On this slide, three dimension tables are defined:

- Store
- Country
- Customer

The fact table is defined as Transaction.

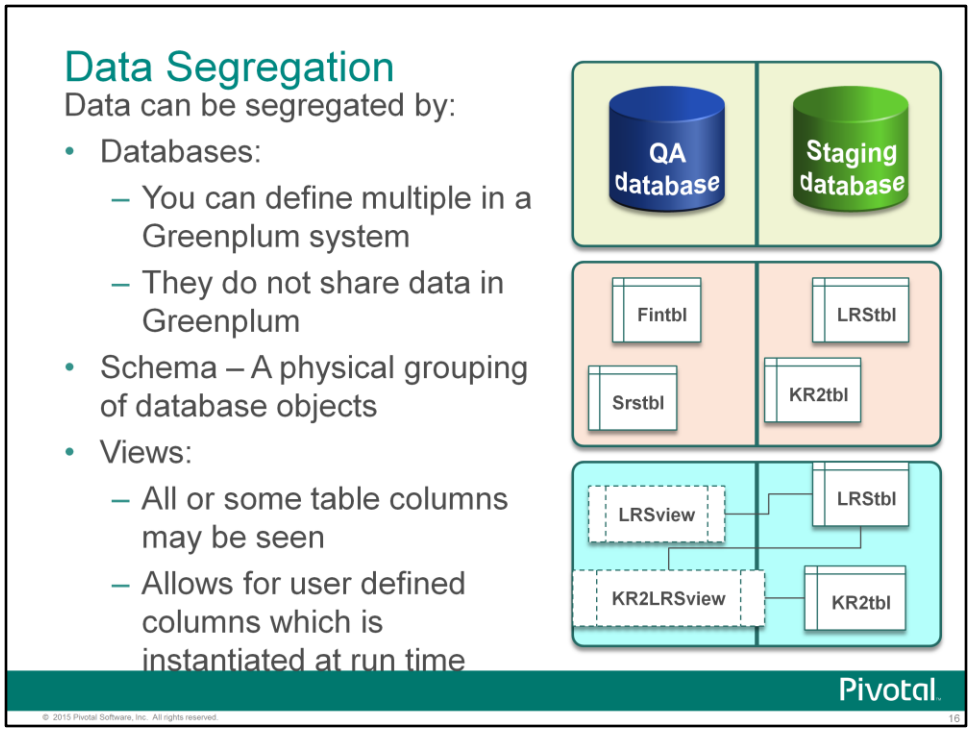
Relationships between the entities are as follows:

- Country has a one-to-many relationship with Store
- Country has a one-to-many relationship with Customer
- Customer has a one-to-many relationship with Transactions
- Store has a one-to-many relationship with Transactions



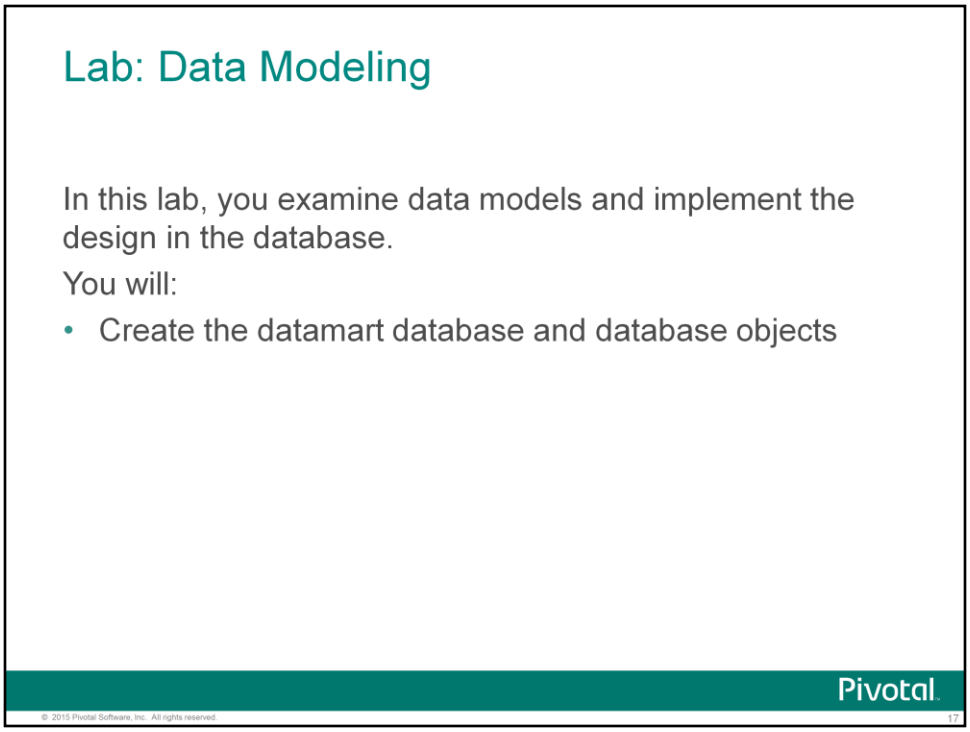
In the physical data model for the logical data model shown on the previous slide:

- Any of the entities defined as either a dimension or fact is now defined as a table.
- Attributes defined for each of those entities become columns.
- Relationships defined between the entities can become foreign keys.

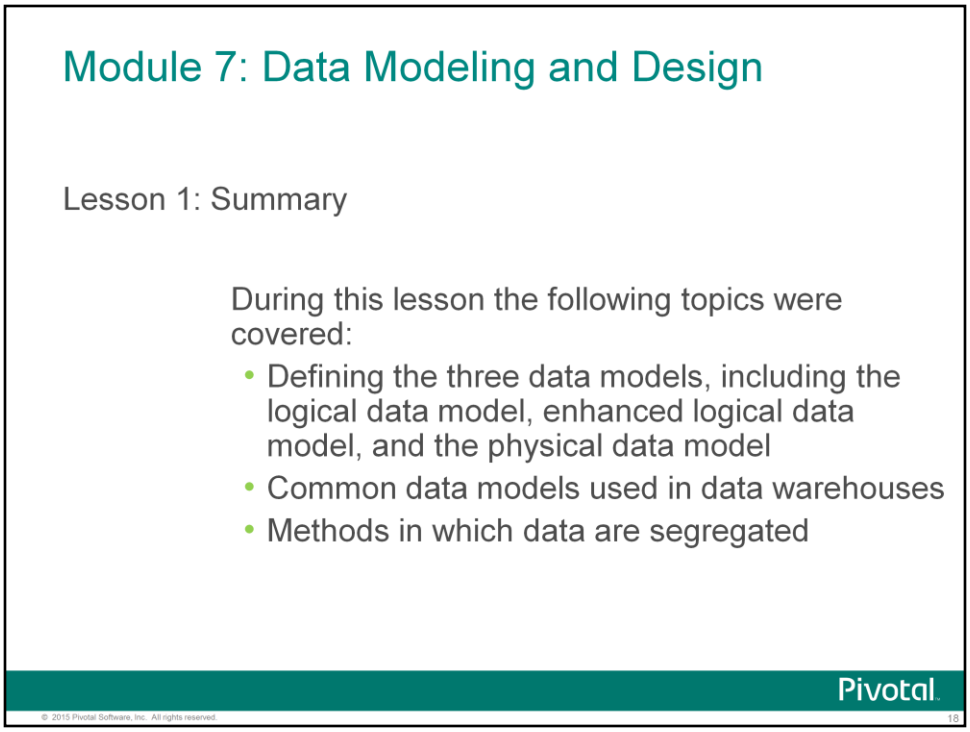


Data segregation can occur at any of the following levels:

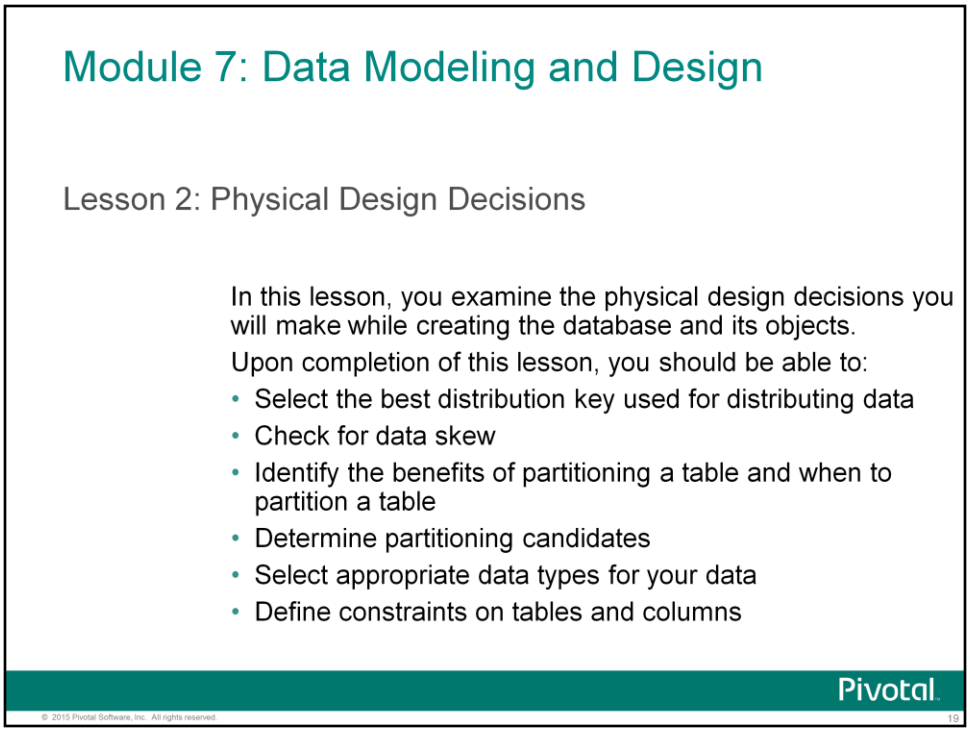
- **Databases** – You can define multiple databases in a Greenplum system. While databases can share roles, they can not view or access data in another database.
- **Schema** – Schemas are used to physically group database objects with each other. A role with appropriate privileges can access data in different schemas. However, the data is physically separated and can be accessed by putting the schema name in front of the object name.
- **Views** – Views provide a functional view of data, hiding the constructs of the table from the user. A view can be created that allows three of five columns of data to be visible to a user.



In this lab, you examine data models and implement the design in the database.



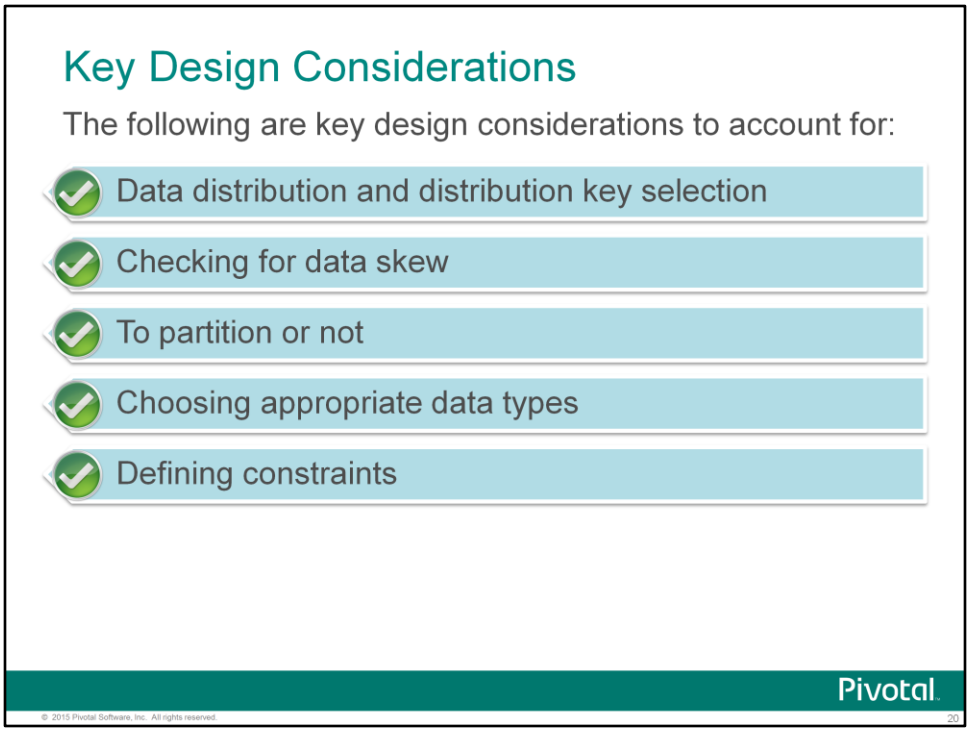
This lesson provided an introduction to various data models, including logical, enhanced, and physical data model. Going through the process of defining your data based on the data model allows you to map business requirements to database models. The lesson reviewed the data models commonly found in data warehouses and the methods used to segregate data.



You must take several key items into account when creating the physical design for your database. These decisions have a great impact on the overall performance of data storage and queries.

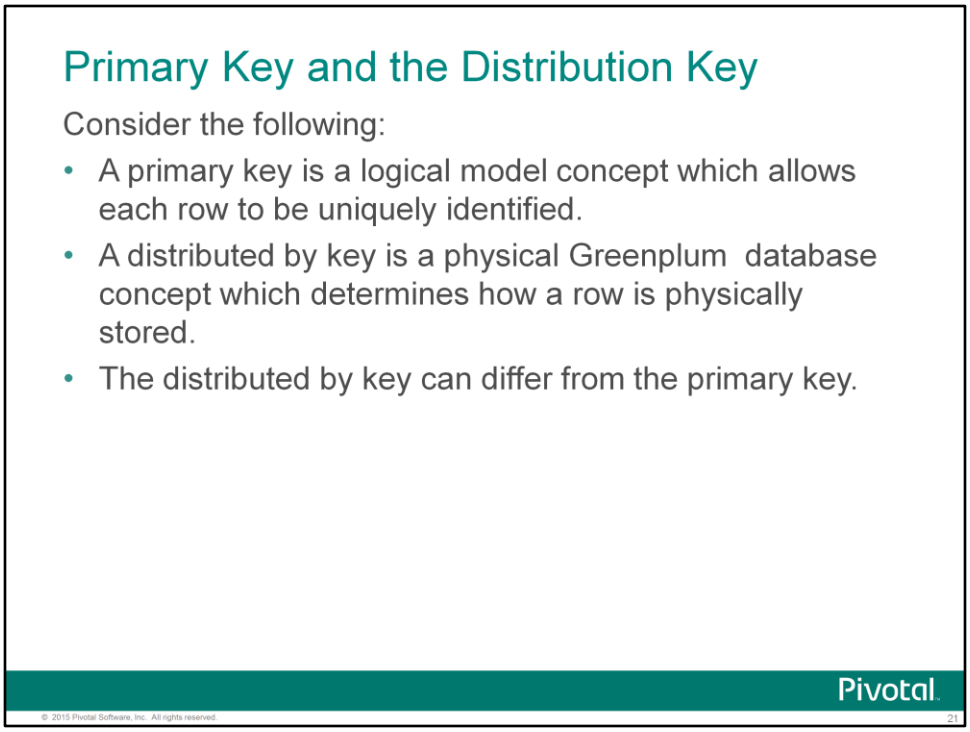
Upon completion of this lesson, you should be able to:

- Select the best distribution key that will be used for distributing data.
- Check for data skew.
- Identify the benefits of partitioning a table and when to partition the table.
- Determine partitioning candidates.
- Select appropriate data types for your data.
- Define constraints on tables and columns.



Greenplum provides several options meant to improve overall query performance for your database implementation. This requires that you make informed decisions when defining the physical data model for your database. You must take into account the following to help you successfully design your system:

- **Data distribution and distribution key selection** – Select the best distribution key that matches your queries. Try not to select columns that are rarely used or are not in the distribution policy for the tables. This will result in slow queries.
- **Data skew** – If you have cases where queries are performing slowly, check for data skew. Data may be unbalanced across the segments. Verify that it is more evenly distributed. If not, consider your options for rebalancing the data.
- **To partition or not** – The decision to partition tables is based on your data and the speed of queries. It takes large tables and breaks them into more manageable pieces which can provide an improvement in performance.
- **Choosing appropriate data types** – Data types can affect storage, which can affect database size, and ultimately, affect performance. Choose the smallest data type that can accommodate your data.
- **Defining constraints** – Use constraints to set parameters around your data and give you more control over data in your tables.



Primary keys in the logical model concept is usually an issue for Terradata users. Understanding that a primary key is a logical model concept where a distributed key is a physical storage concept is key to understanding how your data works.

A well designed Greenplum schema could have a large percentage of tables where the physical distributed by key is different from the logical primary key (tables with common distributed by keys provide faster join operations by reducing number of motion files).

Compare and Contrast – Distribution and Primary Keys

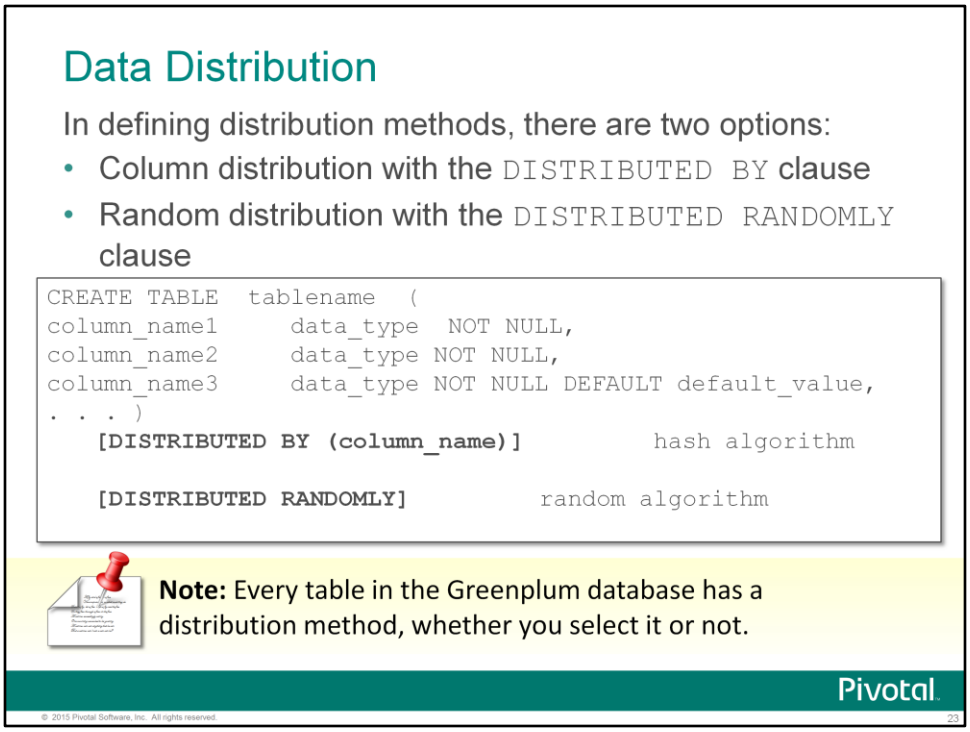
Primary Key	Distribution By Key
Logical concept of data modeling	Physical mechanism for storage
Optional for Greenplum (becomes the distribution index if specified)	Requirement for every Greenplum table
No limit to number of columns	Defined at table creation time or modifiable later
Documented in data model	Can be non-unique
Must be unique	Can uniquely or non-uniquely identify each row
Uniquely identifies each row	Can be NULL
Value can not be changed	Defines a storage path
Can not be NULL	Chosen for distribution and performance
Does not define an access path (DBMS independent)	
Chosen for logical correctness	
Combination of UNIQUE and NOT NULL constraint	

Pivotal

This slide compares the differences between a primary key versus a distribution key.

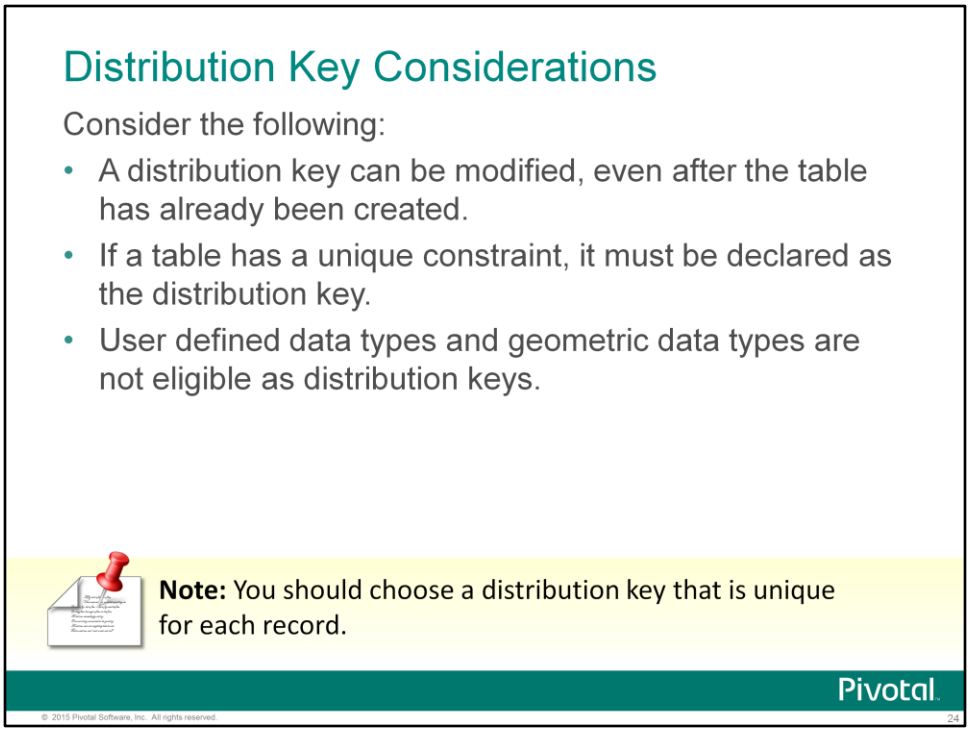
While distribution keys can uniquely identify rows, most times, they do not.

Primary keys are optional in Greenplum. If you do not specify a distribution key, and a primary key exists, it will, by default, be used as the distribution key. It can ensure the most even distribution of data.



Every table in a Greenplum database has a data distribution method. The `DISTRIBUTED` clause specifies the distribution method for a table. There are two distribution methods:

- Column distribution is designated by specifying a column name in the `DISTRIBUTED BY` clause. The `DISTRIBUTED BY` (column name) clause will distribute the data across all segment instances using a hash algorithm.
- Random distribution which is designated by specifying the `DISTRIBUTED RANDOMLY` clause. The `DISTRIBUTED RANDOMLY` clause will distribute the data across all segment instances using a random algorithm.

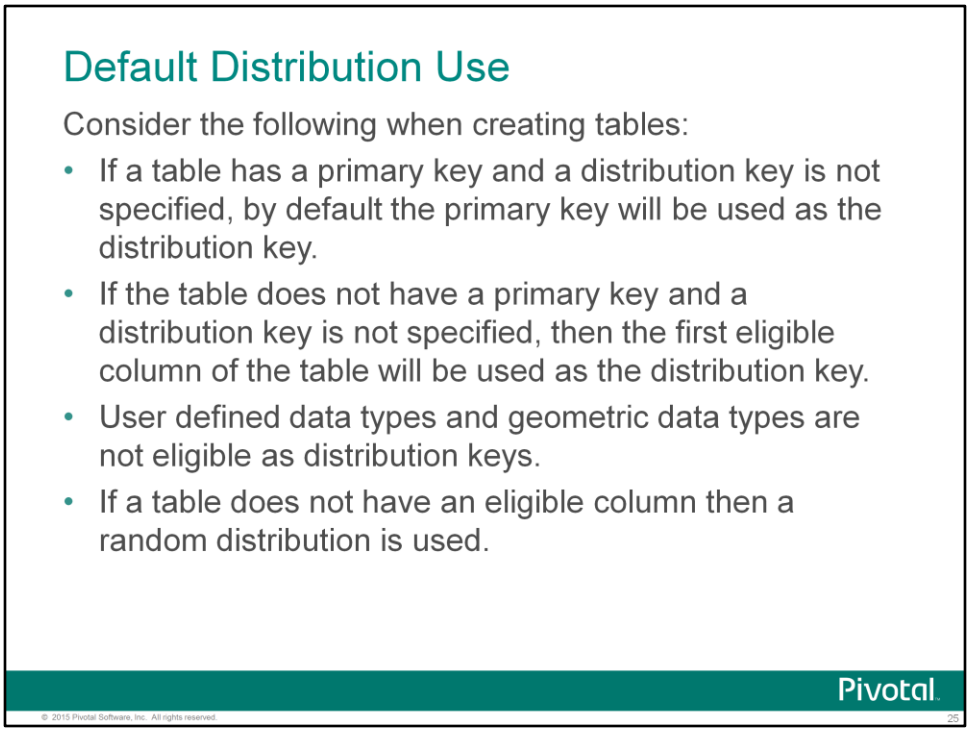


While the distribution policy on a column can be changed, care should be taken to do so. This action redistributes data recursively through child tables.

A table can have one unique constraint and the unique column (or set of columns). If a table has a unique constraint it must be specified as the distribution key for the table.

Columns that contain a user defined data type or geometric data type are not eligible to be used as the distribution key for a table.

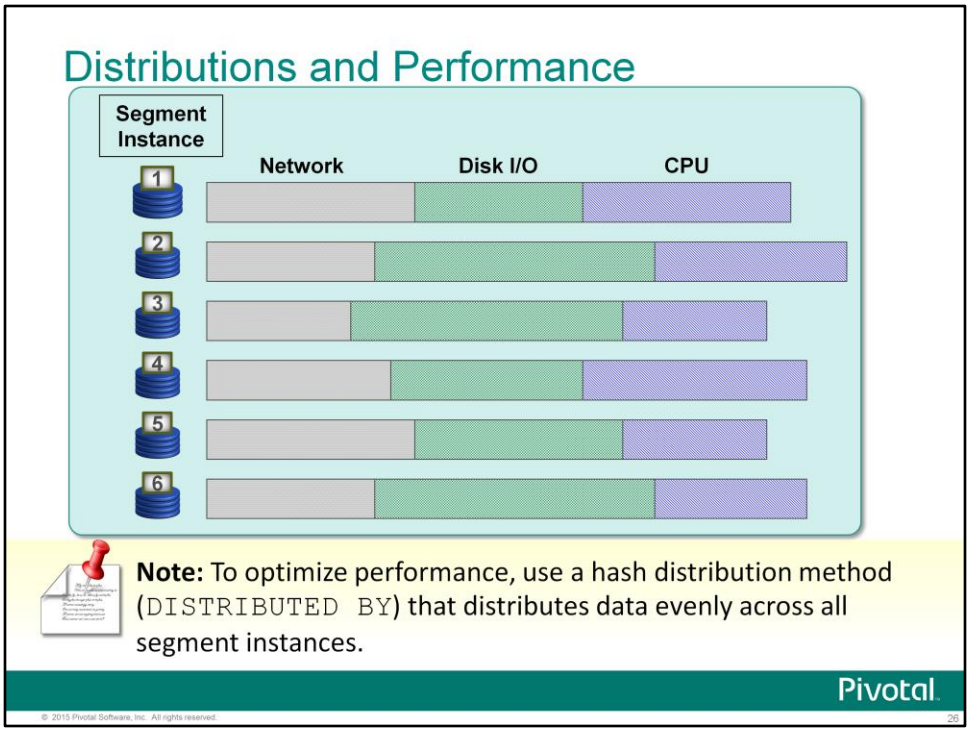
If your primary key consists of 13 columns, your distribution key should include those 13 columns at the very least.



Every table in a Greenplum database has a data distribution method. You should consider not using the system default distribution as it is may not result in an even distribution method. You should be fully aware of the data that will be loaded or inserted so that you make an informed decision on the distribution policy.

When creating tables, the following actions should be accounted for:

- If a distribution method is not specified and the table has a primary key defined then the primary key will be used as the distribution key. If the primary key is chosen well, this can ensure even distribution. However, you should determine for yourself whether the primary key makes a good distribution key.
- If a distribution method is not specified and the table does not have a primary key defined then the first eligible column will be used as the distribution key. Allowing the system to choose the first column as the distribution key can lead to uneven distribution of data. Once again, you must be responsible for selecting a distribution key based on known data and queries.
- User defined data types and geometric data types are not eligible distribution keys.
- If the table does not have a column of an eligible data type, the rows by default are distributed using a random distribution. While this will more evenly distribute your data, it may not be the right selection for your type of data and queries.

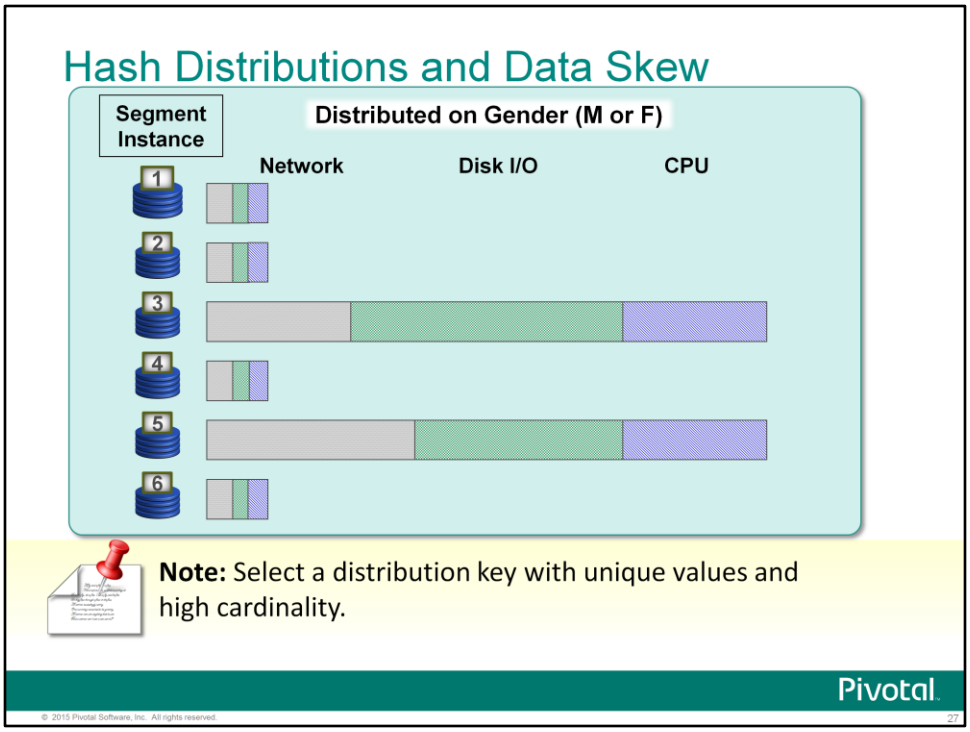


For the best possible performance, use a distribution method that distributes data evenly across all segment instances. In an MPP environment, overall response time for a query is measured by the completion time for all segment instances. If the data is skewed, then the segment instances with more data will have a longer completion time. The optimal goal is for each segment instance to have a comparable number of rows and to perform approximately the same amount of processing. Remember, your database is only as fast as its slowest segment.

For large tables a hash distribution that evenly distributes table rows across all segment instances is desired. While a random distribution will create a more even distribution across all segment instances, the significant performance gains of co-located joins is lost. Co-located joins are discussed in detail later.

Do not choose the random distribution policy or a default policy because it is the easy choice. The intent is to distribute the data to reduce movement, which can increase query times. You can introduce issues where one segment may be working harder if your data is skewed.

Use random distribution if an appropriately chosen single or multi-column distribution key does not distribute data evenly across the segments.

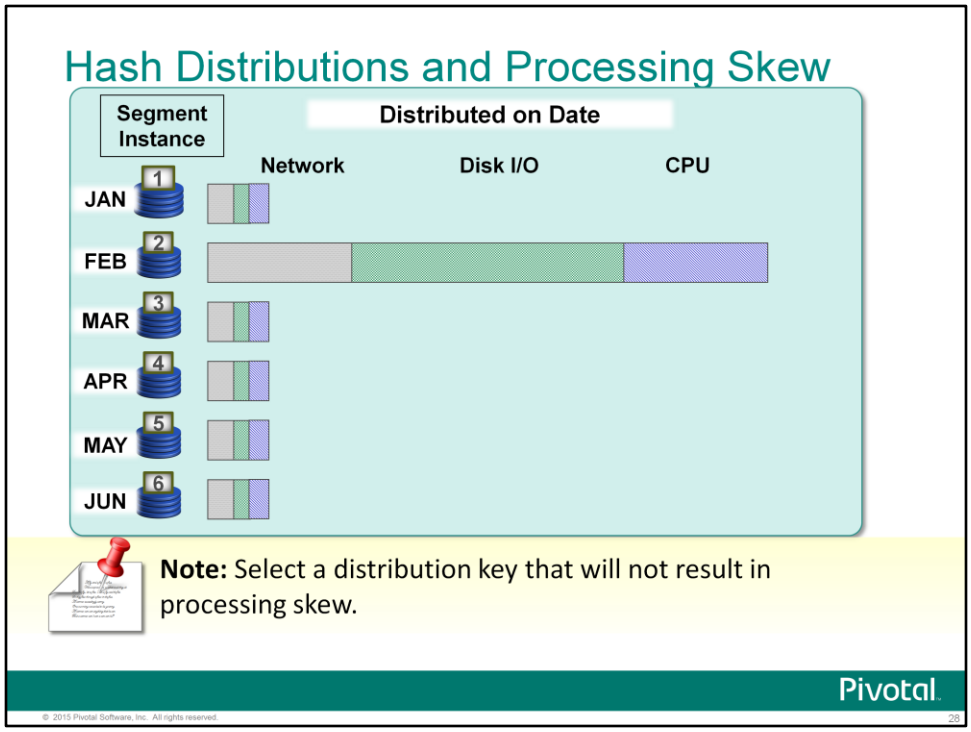


Use a distribution key with unique values and high cardinality to distribute the data evenly across all segment instances.

Boolean keys, such as True/False or 1/0, are not candidates for a distribution key as they will be distributed to two segment instances. Keys with low cardinality, such as Inactive/Active or Male/Female are not candidates for a distribution key.

It is important to remember in an MPP environment, overall response time for a query is measured by the completion time for all segment instances.

In the example displayed a distribution method on a column that contains the value M(ale) and F(emale) is hashed to two hash values. The system distributes rows with the same hash value to the same segment instance therefore resulting in the data being located on only two segment instances.



It is important to select a distribution key to distribute the data evenly across all segment instances but it is equally important to consider processing skew when selecting a distribution key.

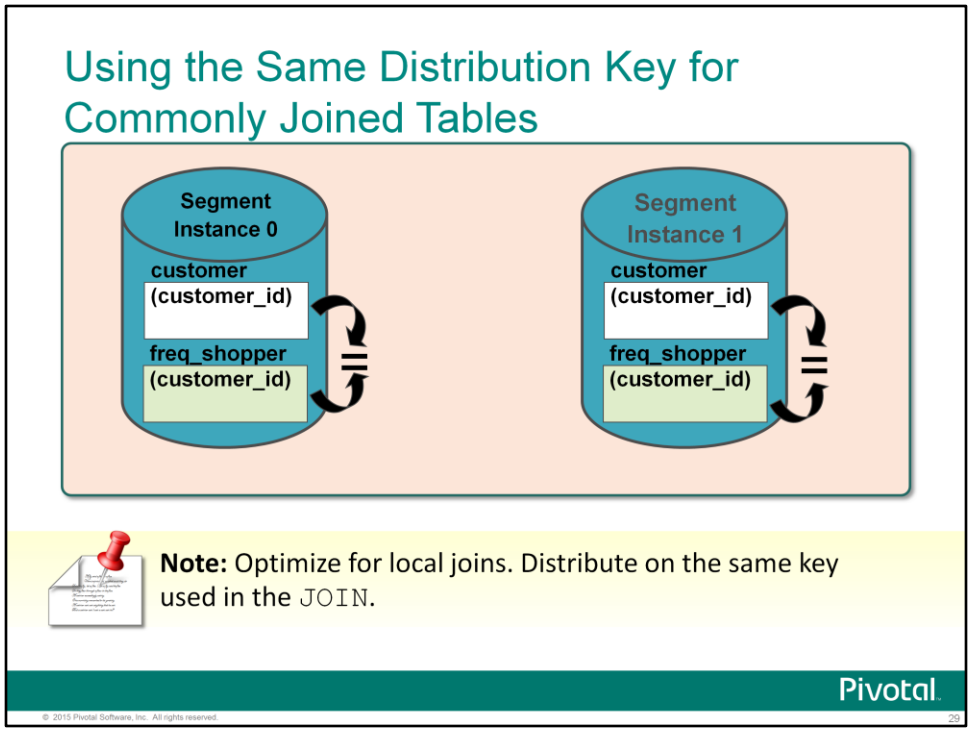
For example, using a `DATE` column as the distribution key may distribute rows evenly across segment instances but may result in processing skew if most of the analysis (queries) is performed by date. Massively parallel processing won't be achieved when all of the records to be processed for a given date are located on a single segment instance.

In the example displayed, the `DATE` column of a table provided an even distribution across all segment instances. Data records are distributed evenly by month with only slight variations in the number of rows in a month. While this resulted in an even distribution, analysis is performed by month with queries analyzing a month of data at a time (`SELECT … FROM table_name WHERE month = "FEB"`). In this example a query `WHERE month = "FEB"` will result in processing skew with only one segment instance working on behalf of the query.

Remember in an MPP environment, response time for a query is measured by the completion time for all segment instances.

Use `iostat` to check for processing skew. Use `gpssh` for `ssh` access to run system commands on multiple segment hosts at the same time.

If you distribute on a good column where the data is laid out well, but some segments end up working harder than other segments, you introduce processing skew. In such a case, you may be better served distributing on a unique key to distribute the data more efficiently.



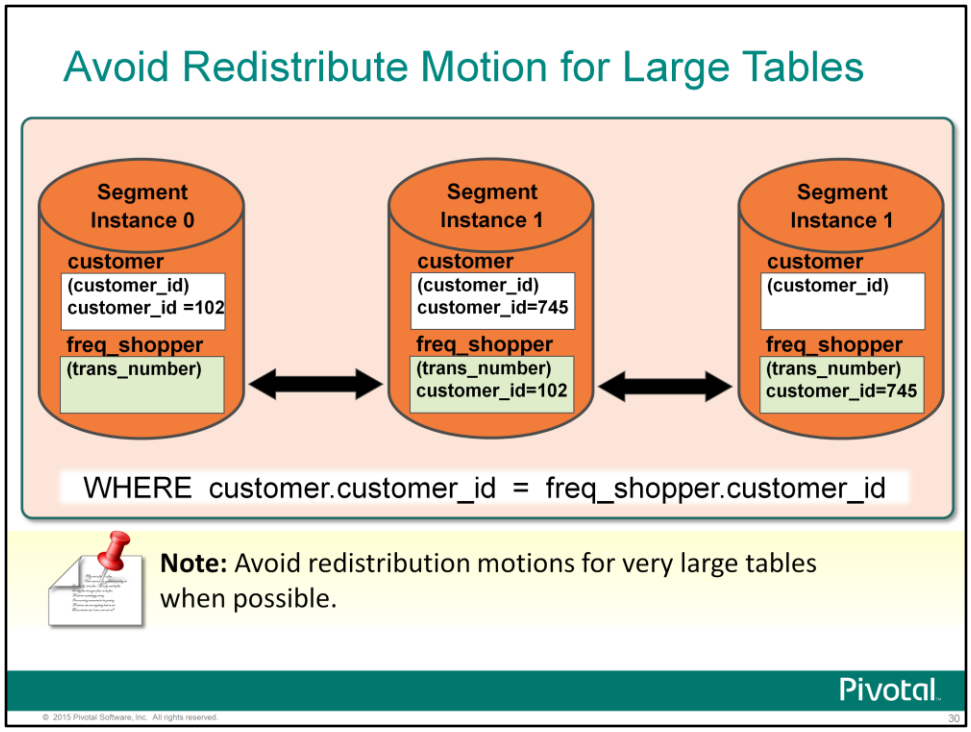
For large tables that are commonly joined together use the same distributed key to obtain local joins. A local join that is performed within the segment instance minimizes data movement and provides tremendous performance gains. Distribute on the same key (column) used in the join (`WHERE` clause).

When a local join is performed a segment instance operates independently of the other segment instances without network traffic or communication between segment instances.

In the point-of-sale example displayed the customer and `freq_shopper` table are very large tables and are commonly joined together on the `customer_id` in queries. The `customer_id` is also unique and provides for an even distribution of data. Therefore the same distribution key (`customer_id`) is used for both the customer and `freq_shopper` table to obtain the performance benefits of local joins.

To join the customer and `freq_shopper` tables each segment instance simply performs the following:

1. Scan the customer table that is already distributed on the join key and hashing it.
2. Scan the `freq_shopper` table that is already distributed on the join key and hashing it.
3. Perform a hash join operation between the two tables all locally within the segment instance without network activity.



To perform a local join matching rows must be located together on the same segment instance. In the case where data was not distributed on the join key, a dynamic redistribution of the needed rows from one of the tables to another segment instance will be performed.

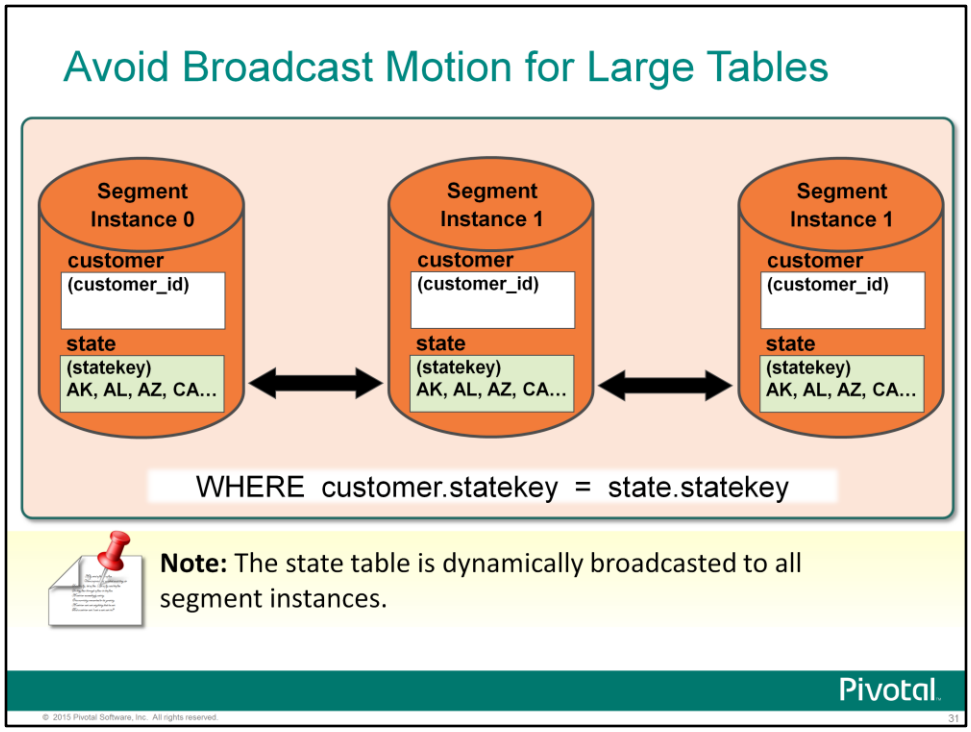
There is a performance cost to redistributing data that must be performed every time the join is required for a query. Though the performance impact is minimal for small tables, avoid redistribution motion for very large tables when possible.

When a redistribution is required the basic steps must still be performed:

1. Scan one table.
2. Scan the second table.
3. Perform the join with an additional step of redistributing the data before it can be joined.

In the example displayed, the `customer` table was distributed on the `customer.customer_id` and the `freq_shopper` table was distributed on `freq_shopper.trans_number`.

To perform the join between the `customer` and `freq_shopper` table `WHERE` the `customer.customer_id = freq_shopper.customer_id`, the `freq_shopper` table must be dynamically redistributed on the `freq_shopper.customer_id`.



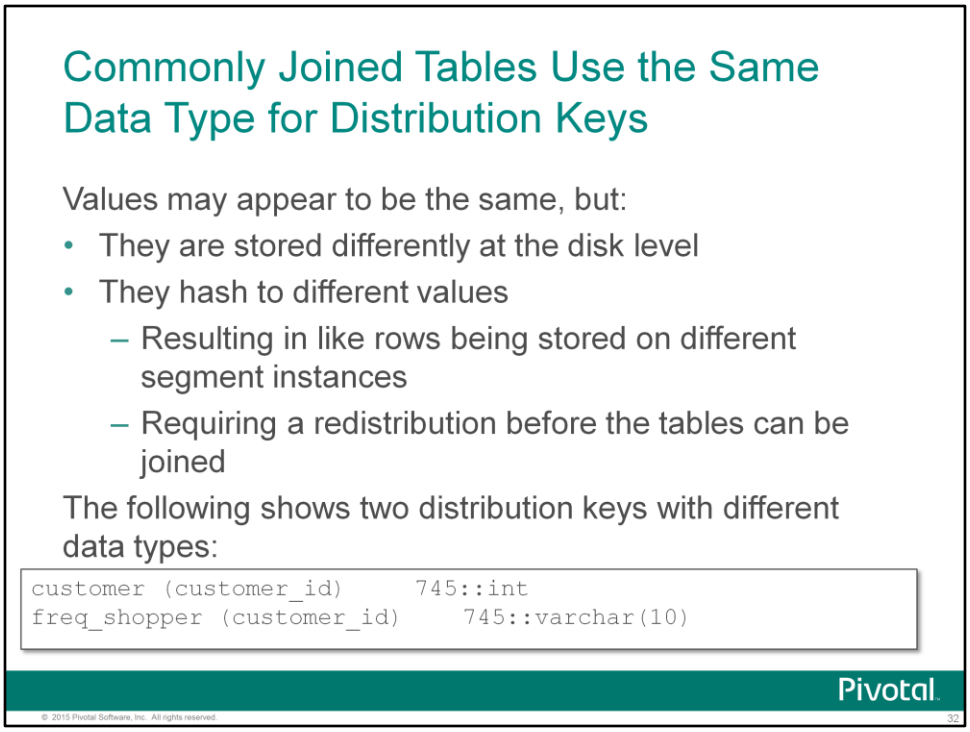
In some cases, a broadcast motion will be performed rather than a redistribute motion. In a broadcast motion every segment instance performs a broadcast, or sends its own individual rows to all other segment instances. This results in every segment instance having its own complete and local copy of the entire table.

A broadcast motion may not be as optimal as a redistribute motion therefore the optimizer typically only selects a broadcast motion for very small tables. A broadcast motion is not acceptable for large tables.

In the example displayed, the state table was distributed on the `state_key` and the `customer` table was distributed on the `customer_id`. It was not feasible to distribute the `customer` table on the `state_key` because it would result in data skew and the customer table is commonly joined using the `customer_id` with other tables.

The `customer` table is very large and the state table is very small relative to the number of rows. The optimizer rather than redistributing the customer table decides to broadcast the state table.

Each segment instance will send their individual rows from the state table to all other segment instances. Each segment instance will then have its own complete and local copy of the `state` table, to join with the `customer` table.



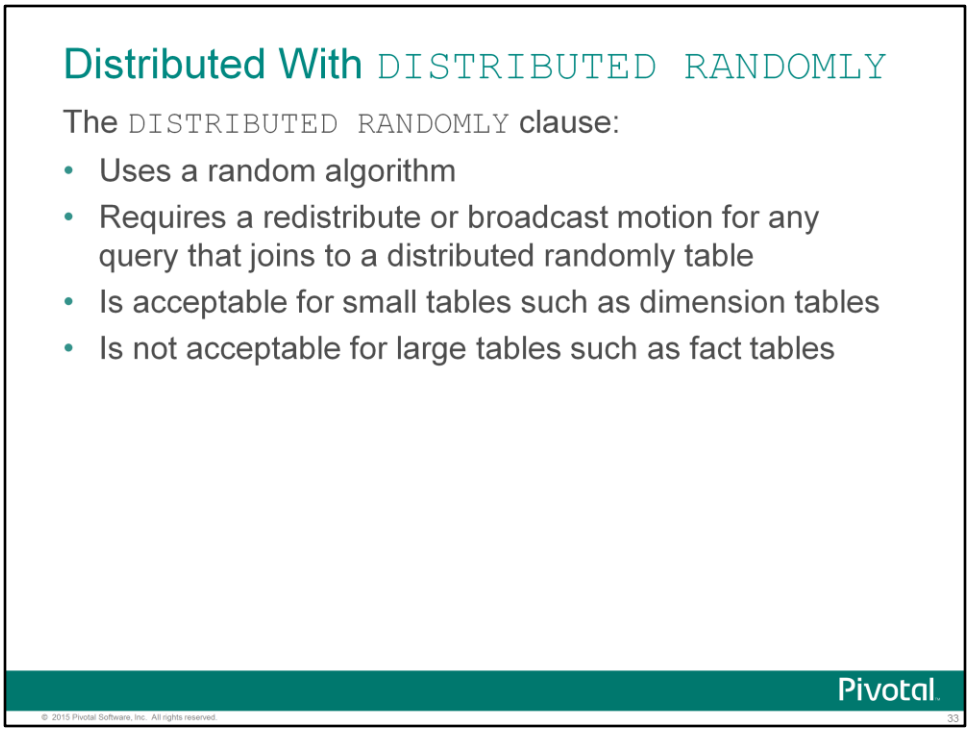
To perform a local join, matching rows must be located together on the same segment instance. It is important that the distribution keys are the same data type to obtain a local join.

While the values might appear to be the same representatively, different data types are stored differently at the disk level. The data types hash to different values resulting in like rows being stored on different segment instances.

In the example displayed:

- The `customer` table is distributed on the `customer.customer_id` which is defined as type `INT`.
- The `freq_shopper` table also distributed on the `freq_shopper.customer_id` is defined as type `VARCHAR`.
- This results in like rows, with the same `customer_id` respectively, being stored on different segment instances, requiring a redistribution to perform the join.

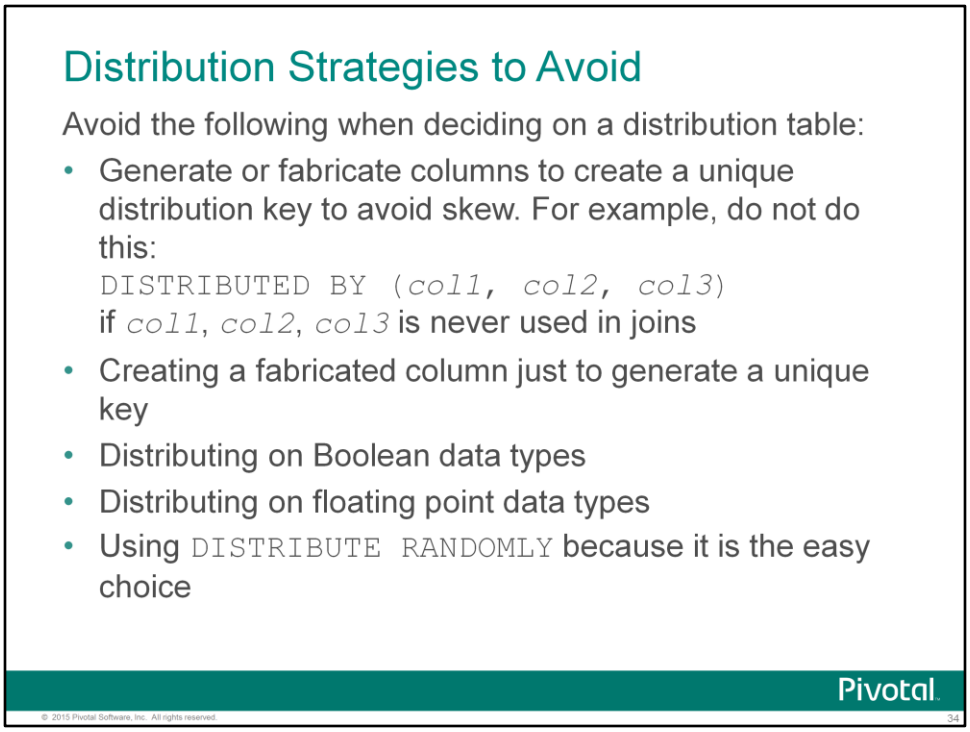
Instead of a local join, the optimizer redistributes the customer table based on the `VARCHAR` representation of the `customer.customer_id`. A hash join is then performed using the same data type for both `customer_id` columns (`customer.customer_id = varchar (freq_shopper.customer_id)`).



The `DISTRIBUTED RANDOM` clause distributes the data across all segment instances using a random distribution algorithm. For any table that uses a random distribution either a redistribution or broadcast operation will be required to perform the table join.

There are performance implications when performing a redistribution or broadcast of very large tables.

Random distribution should be used for small tables and when a hash distribution method is not feasible due to significant data skew.



Two of the most influential factors in optimizing the performance in a Greenplum data warehouse include:

- A distribution strategy that distributes rows evenly across all segment instances.
- A distribution strategy that achieves local joins.

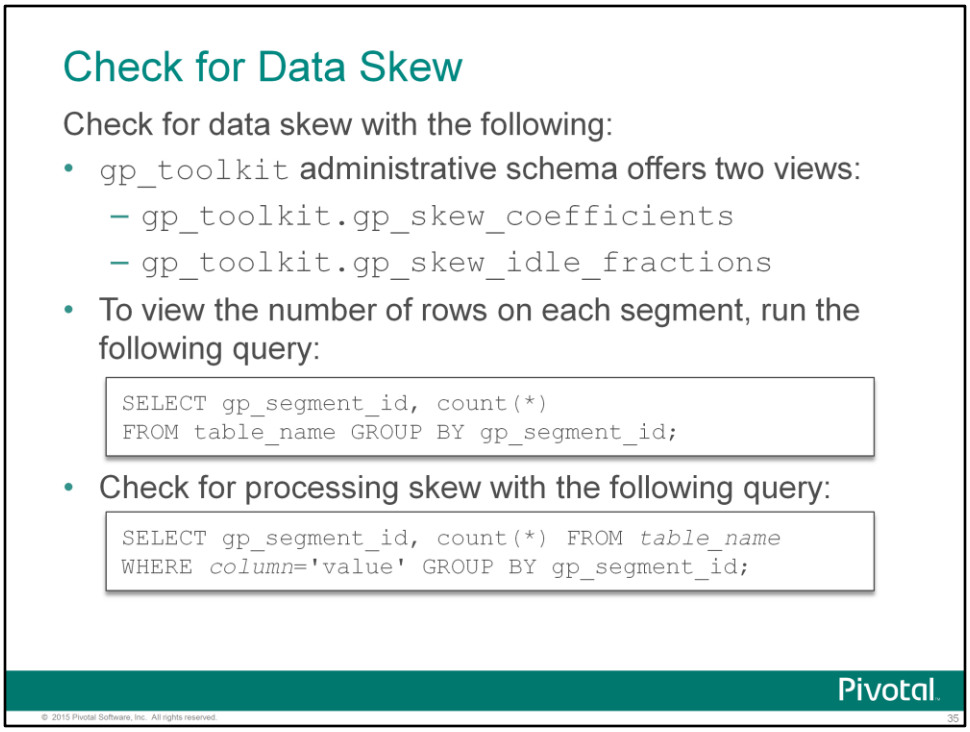
It is important to keep in mind it is possible to negatively impact performance through bad distribution decisions and overly complex strategies.

Do not combine columns in a multi-column distribution simply to generate a unique key to avoid data skew if the multi-column distribution key is never used in joins. In this scenario either a redistribution or broadcast operation will be required to perform the table join.

Never create fabricated or concocted columns to a table to generate a unique key (for example a homegrown serial key generation algorithm) to provide for an even distribution.

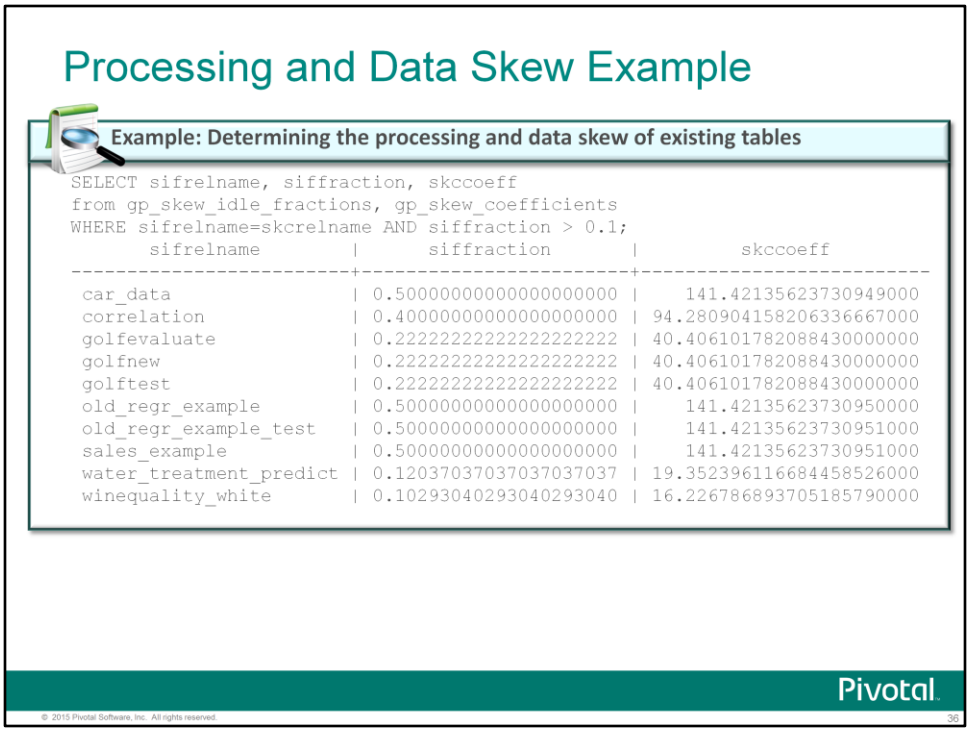
In both of these scenarios if a good distribution key is not available distribute the data using the random distribution method.

Do not distribute on Boolean data types or floating point data types. Boolean distributions will result in data skew. Floating point data types result in lossy arithmetic and diminished performance as hash join can not be performed on floating point columns. Favor integer column types for distribution columns.



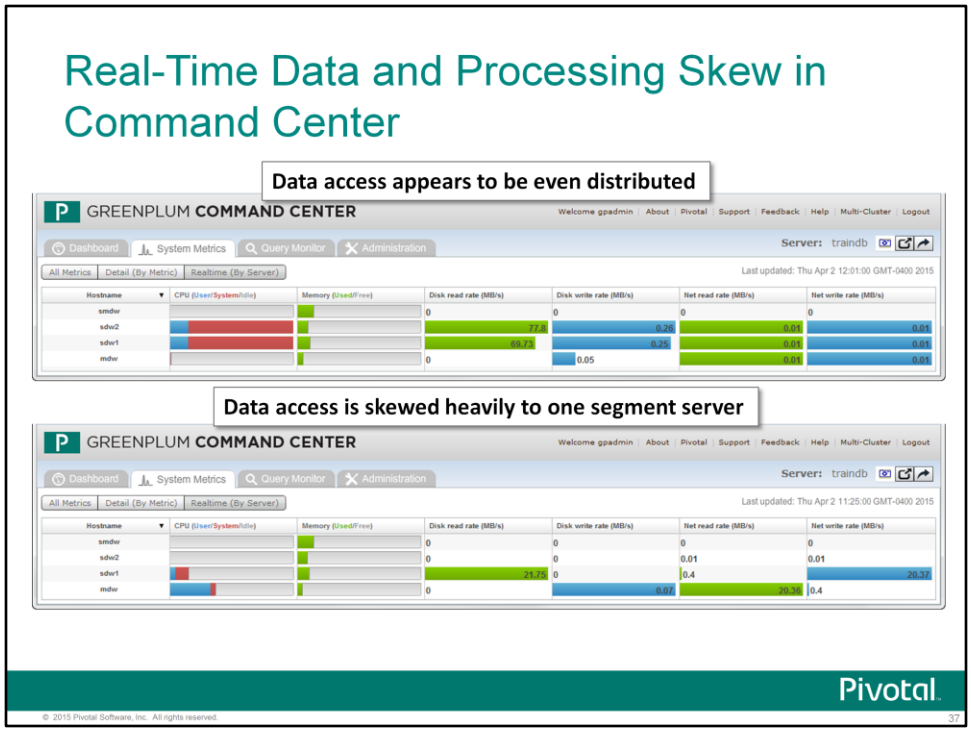
It is important to validate the table distributions and to check for data skew. Use the `gp_toolkit` administrative schema to check for uneven table distributions.

Individual segment host performance may also be viewed during query processing using the Greenplum Command Center.



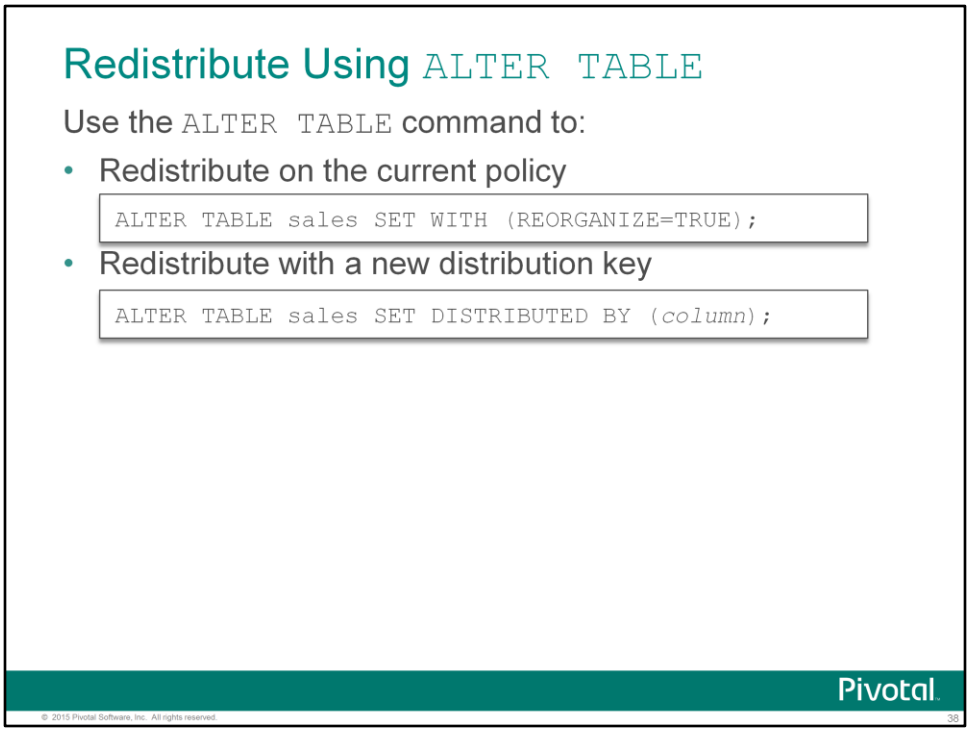
In this example, an SQL query is submitted against the `gp_skew_idle_fractions` view and the `gp_skew_coefficients` view to identify any tables with a processing skew greater than 10%. The related coefficient variable (`skccoeff`) is also displayed.

Note, the greater the processing skew, the greater the coefficient variable.



Greenplum Command Center displays real time data reads and writes, CPU processing, and network reads and writes.

The chart can be used to determine if one or more segments display greater processing or computational skew than others. Data access may change during the execution period of the in-flight SQL statement, so it is important to not look just at a single moment but to watch the update during the entire time the query is being executed.

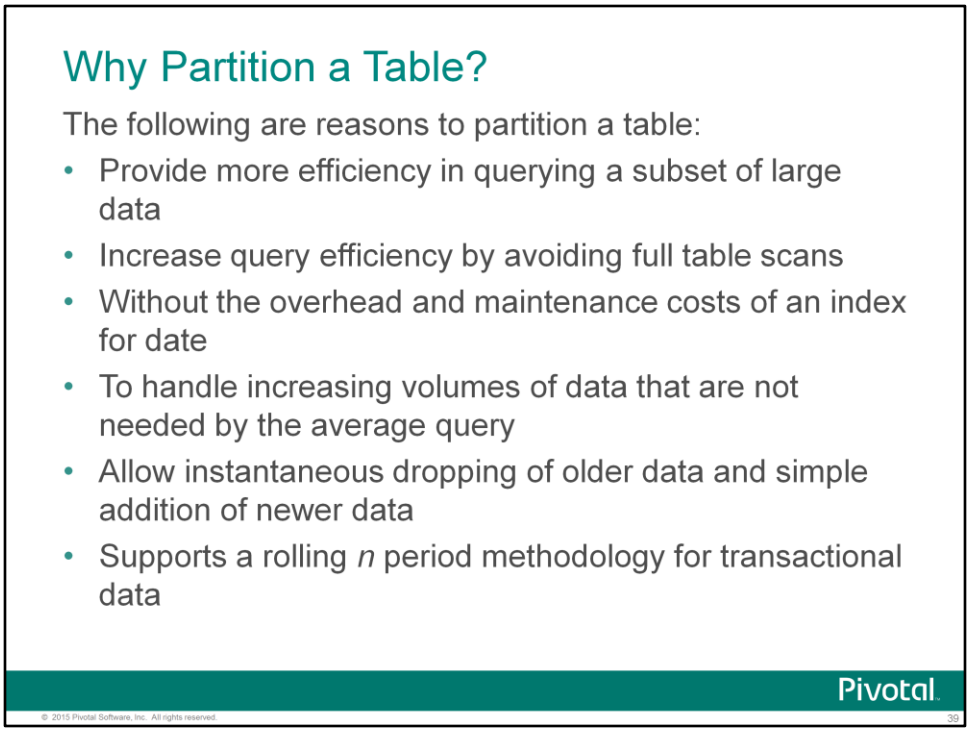


Use the `ALTER TABLE` SQL command to redistribute a table either on the current distribution policy to rebalance data, or by specifying a new distribution policy to distribute on the newly chosen distribution key.

To redistribute:

- On the current policy, specify the `SET WITH (REORGANIZE=TRUE)` clause.
- On a different distribution key, use the `SET DISTRIBUTION BY (`*`column`*`)` clause, where *`column`* is the column name for the new distribution key.

Data will be redistributed automatically and will redistribute recursively on child tables.



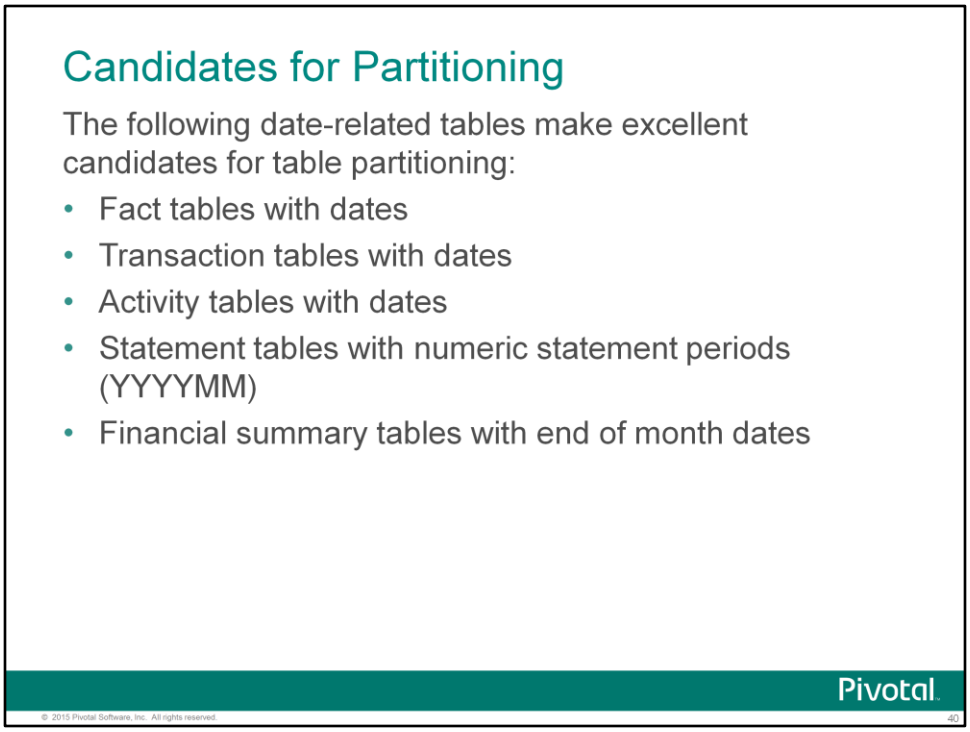
While Greenplum is not afraid of full table scans, it is a good idea to partition your table to help Greenplum process the table as efficiently as possible. However, your business rules should dictate if and how you partition tables.

The following are some of the reasons for considering table partition as part of the design:

- Provide more efficiency in querying against a subset of large volumes of transactional detail data as well as to manage this data more effectively. Businesses have recognized the analytic value of detailed transactions and are storing larger and larger volumes of this data.
- Increase query efficiency by avoiding full table scans.
- Without the overhead and maintenance costs of an index for date.
- As the retention volume of detailed transactions increases, the percent of transactions that the "average" query requires for execution decreases.
- Allow the instantaneous dropping of old data and the simple addition of newer data that may be queried more often. One of the practical uses of partition tables is partition swapping or partition exchanging. This type of practice allows you to load data to a table and replace an existing table with the same data plus the updated information with little impact to the original table.

  Support a *rolling n periods* methodology for transactional data.

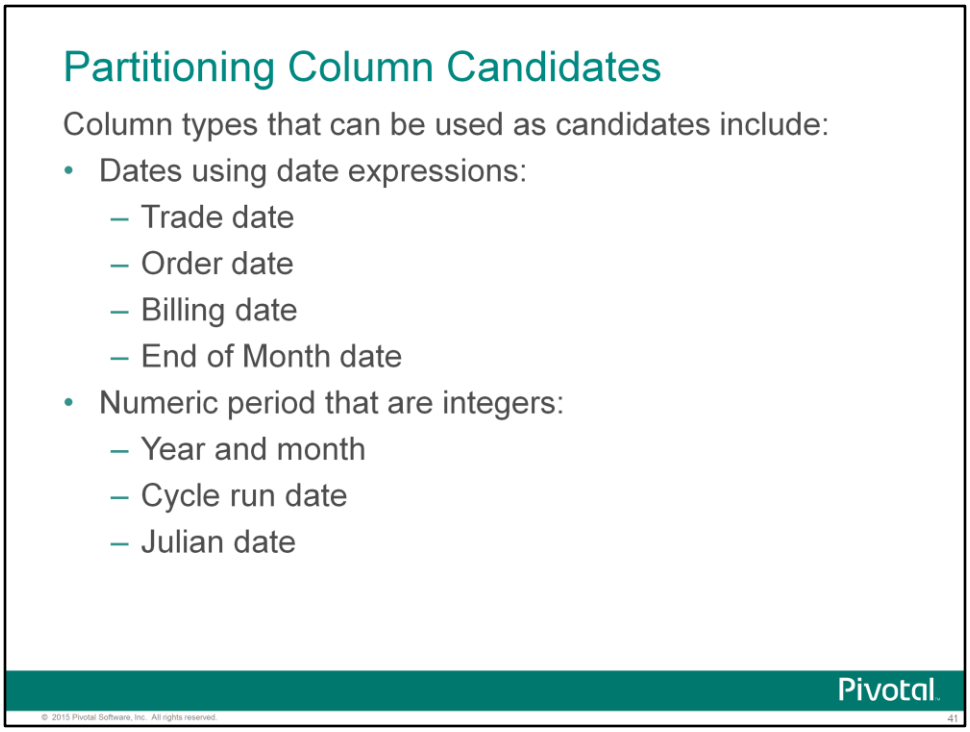
You can have default partitions that act as a catchall. This allows you to insert rows without having to worry about a partition. Later, you can alter the table to split the partition with the values that you want to go to the partitioned table.



Any table with large data volumes where queries commonly select for data using a date range or where the table has roll-off requirements makes an excellent candidate for table partitioning. Most of your partitions will be partitioned on dates. This allows you to implement temperature-sensitive data storage.

Candidates for partitioning include:

- Fact tables with dates
- Transaction tables with dates
- Activity tables with dates
- Statement tables with numeric statement periods
- Financial summary tables with end of month dates



Columns used for partitioning should be simple and meet your business requirements. Examples for dates include:

- Dates with date expressions:
    - Trade date
    - Order date
    - Billing date
    - End of month date
- Numeric representations for dates, including:
    - Year and month
    - Cycle run date
    - Julian date

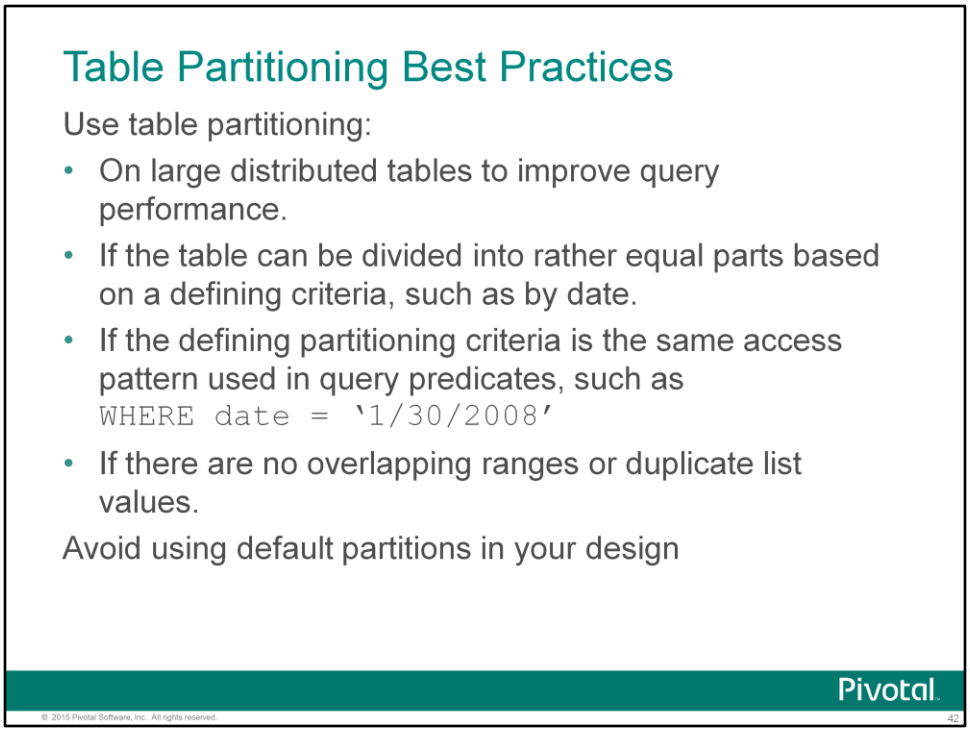


Table partitioning is not a substitute for table distributions. Therefore, it is important to create optimal data distributions for each table.

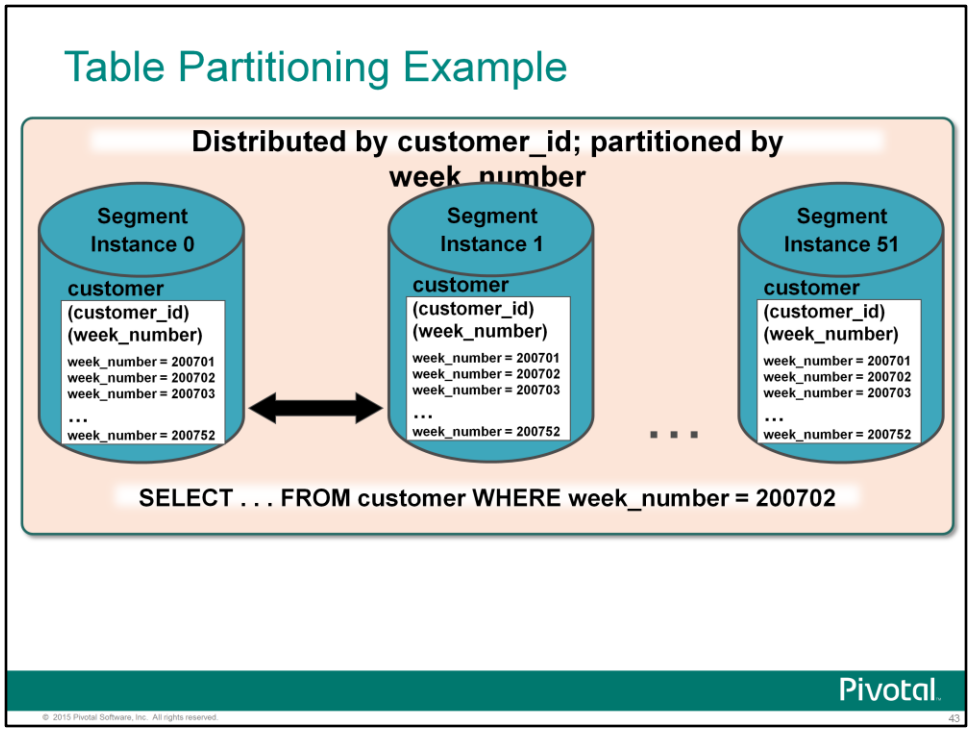
The primary goal of table partitioning is to eliminate scanning partitions that contain data that is not needed to satisfy a query.

Consider table partitioning on large tables that can be divided into somewhat equal parts based on a defining criteria and the defining criteria is used in query predicates, the `WHERE` clause. The `WHERE` clause is your biggest identifier for partitioning.

If the query access pattern (`SELECT….WHERE`) does not match the partitioning definition the benefit of partition elimination is not achieved.

When defining partitioning criteria it is important to not have overlapping ranges if using range partitioning and to ensure list values are unique if using list partitioning. The query planner will have problems identifying which segment to put the data into.

While default partitions provide a method for catching all data that does not fit other partitions, the use of a default partition can negatively impact performance. Default partitions are always scanned. If the partition contains a large amount of data, this can negatively impact performance during table scans.



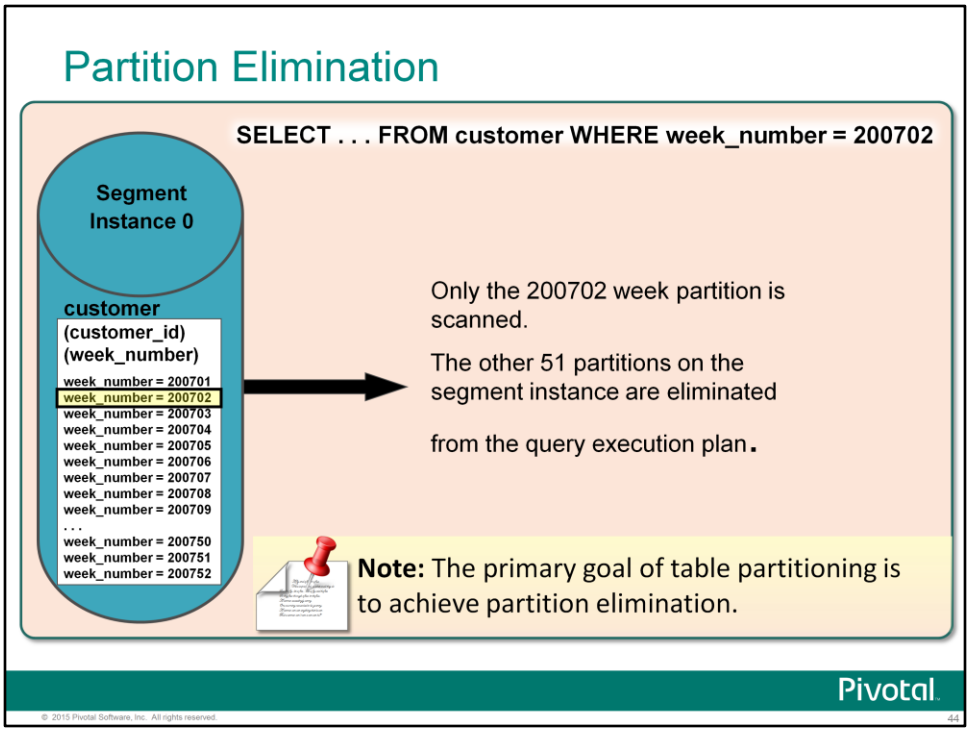
In the example displayed on the slide, the `customer` table was distributed on `customer_id` providing for an even distribution across all segment instances.

In this data warehouse environment example, a large percentage of the queries are against a small subset of history. For example:

- One week (`WHERE week_number = 200702`)
- One month (`WHERE week_number = 200701 AND week_number = 200702 AND week_number = 200703 AND week_number = 200704`)
- One quarter

In this scenario, query performance can be improved by eliminating partitions to be scanned that contain the other weeks that are not needed to satisfy the query.

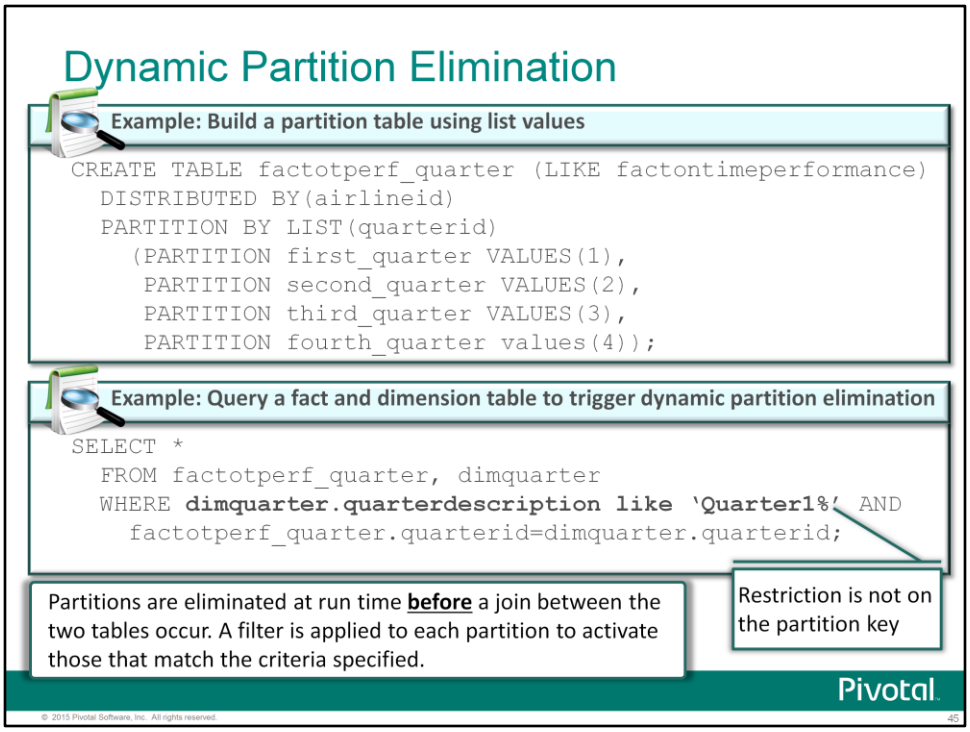
It was also determined that `week_number` can be easily divided into equal parts with no overlapping ranges. Therefore the `customer` table is partitioned on `week_number` with each segment instance containing fifty-two partitions or child tables based on week number.



Partition elimination is the act of filtering and eliminating data irrelevant to a query. By reducing table scans, query processing significantly improves, allowing for more concurrency.

For a query that selects a single week, for example `WHERE week_number = 200702`, the remaining fifty-one partitions within the segment instance are eliminated from the query execution plan and are not scanned. This results in improved query performance.

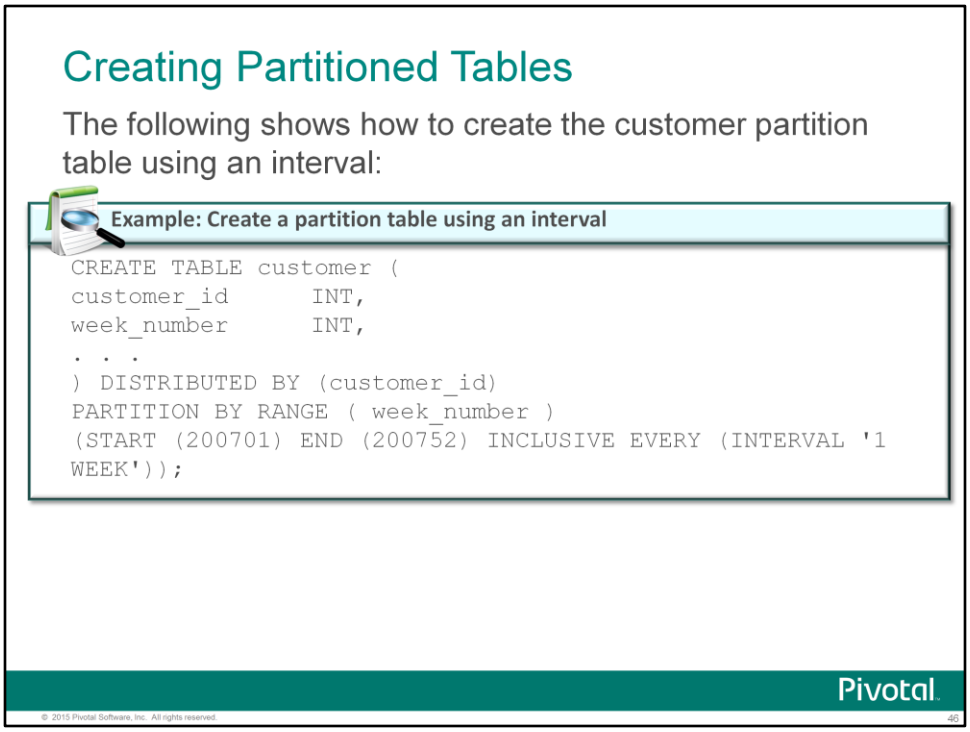
The overall goal of table partitioning is to achieve partition elimination to improve query performance. This can occur either as part of the query plan or during query run-time.



Partition elimination that occurs during run-time is called dynamic partition elimination. Dynamic partition elimination is used whenever the values are only available at query run-time. These values are used to prune partitions dynamically, once again, improving the speed of query processing.

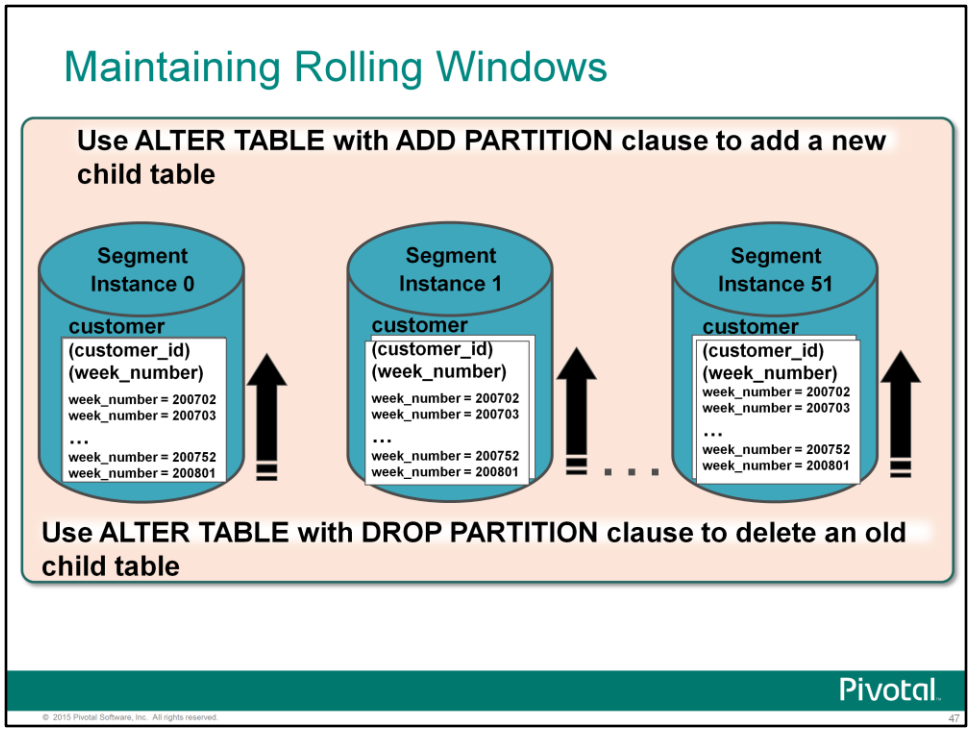
This can occur whenever the restriction defined as part of the `WHERE` clause is not on the partitioning key. While the feature was available in earlier releases of Greenplum 4.x, it applied to constant values known to the partition table at planning time. The feature was extended to eliminate partitions at run-time.

Dynamic partition elimination is on by default, but can be disabled by setting the server configuration parameter, `gp_dynamic_partition_pruning`, to `off`.



Once a partition strategy is determined that provides for partition elimination, the partitions or child tables must be created.

Use the `CREATE TABLE` statement with the `INHERITS` clause to create the parent-child relationship and inheritance from the parent table. Use the CHECK constraints clause to limit the data a child table can contain based on some defining criteria.



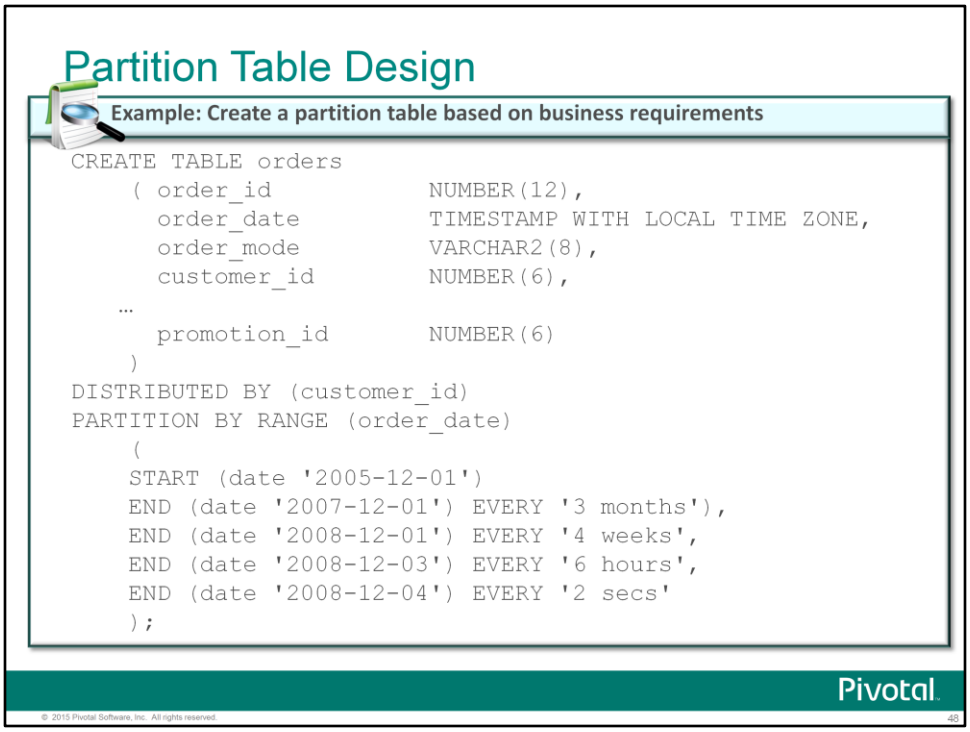
Many data warehouse environments may contain rolling history data that represents many weeks, months or years of activity. A rolling history table typically contains a very large number of records. For these tables a rolling window of data is stored and maintained based on a defining criteria.

Continuing with the example:

- Fifty-two weeks of data are stored and analyzed in the customer table.
- Each week, a new week's worth of data is inserted and the oldest week's worth of data is deleted, maintaining fifty-two rolling weeks of data at any given time.
- As these rows are historical in nature, they are rarely, if ever, modified after insertion.

To maintain the rolling window of data:

- Use `ALTER TABLE` command with the `ADD PARTITION` clause to a new partition or subpartition containing the new data.
- Use `ALTER TABLE` command with the `DROP PARTITION` clause to a remove a containing older data.

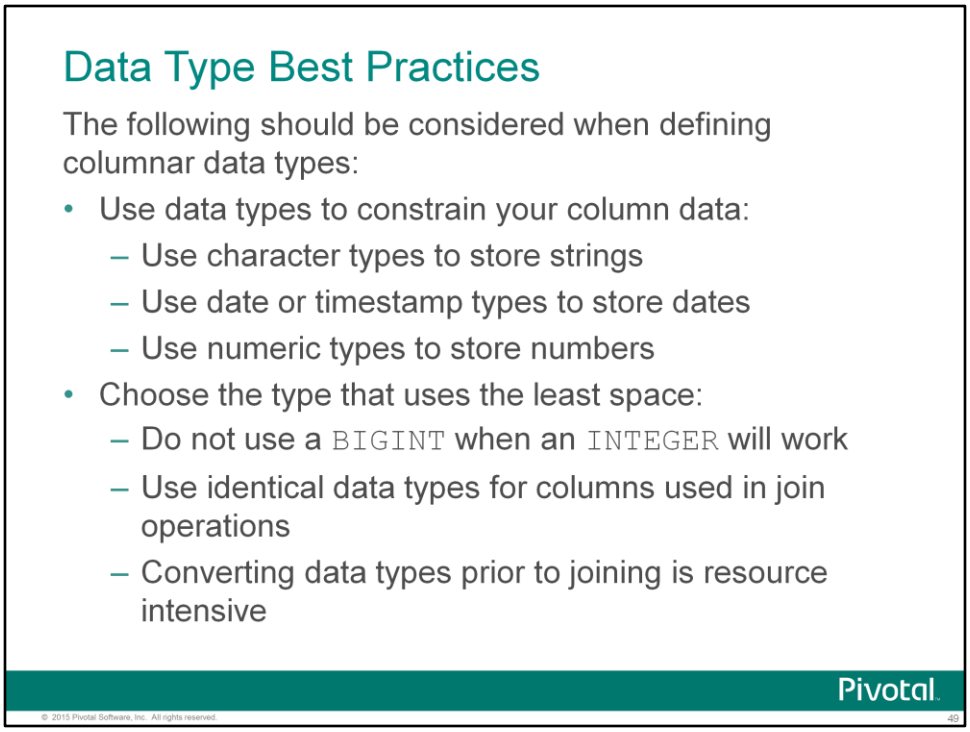


The business requirements dictate the design you choose when partitioning a table.

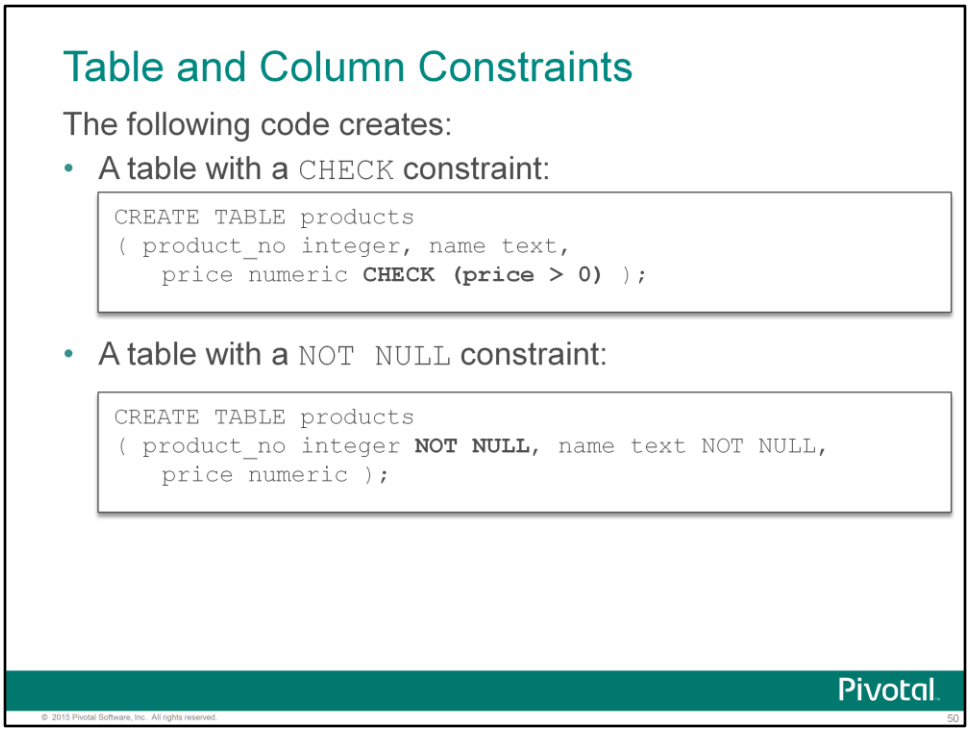
In this example, the SQL syntax to create a single-level partition for the `orders` table, where there no partitions are defined, is displayed.

The table is partitioned by a range on the `order_date` column, with data being created:

- Every 3 months for a specific date range
- Every 4 weeks for another date range
- Every 6 hours for a third date range
- Every 2 seconds for the last date range shown



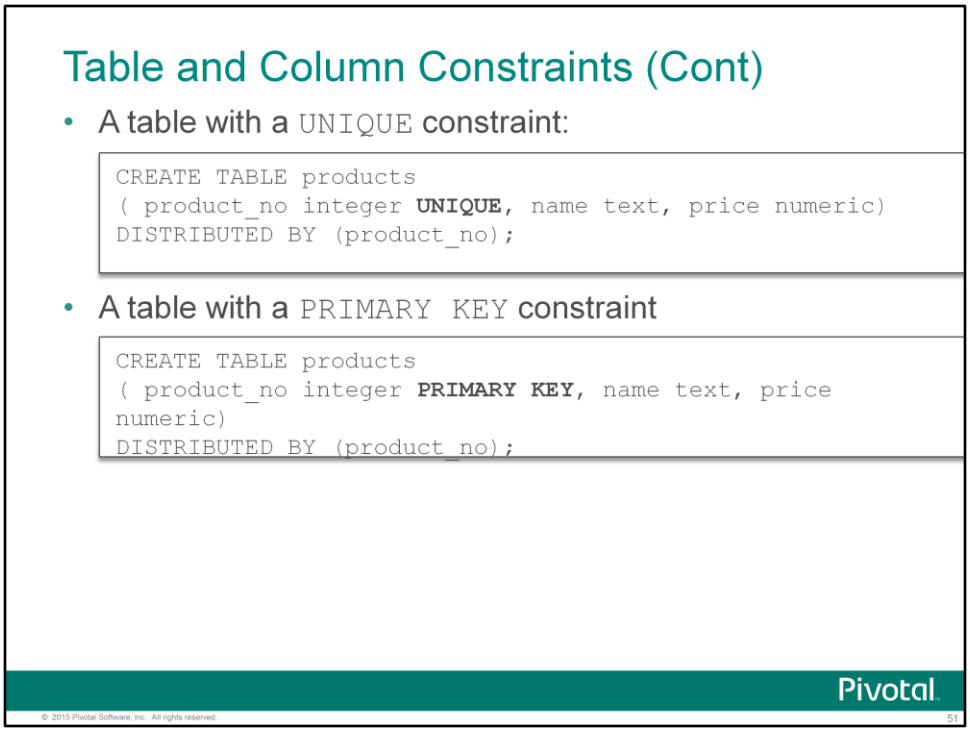
Use the correct data types to store your values. Verify that the data types match on JOIN operations. You must analyze your table design to ensure they can be matched on joins. Joins work more efficiently if the data types of the columns used in the join predicate have identical data types. When the data types differ, the database must convert one of them so that the data types can be compared correctly, amounting to unnecessary overhead.



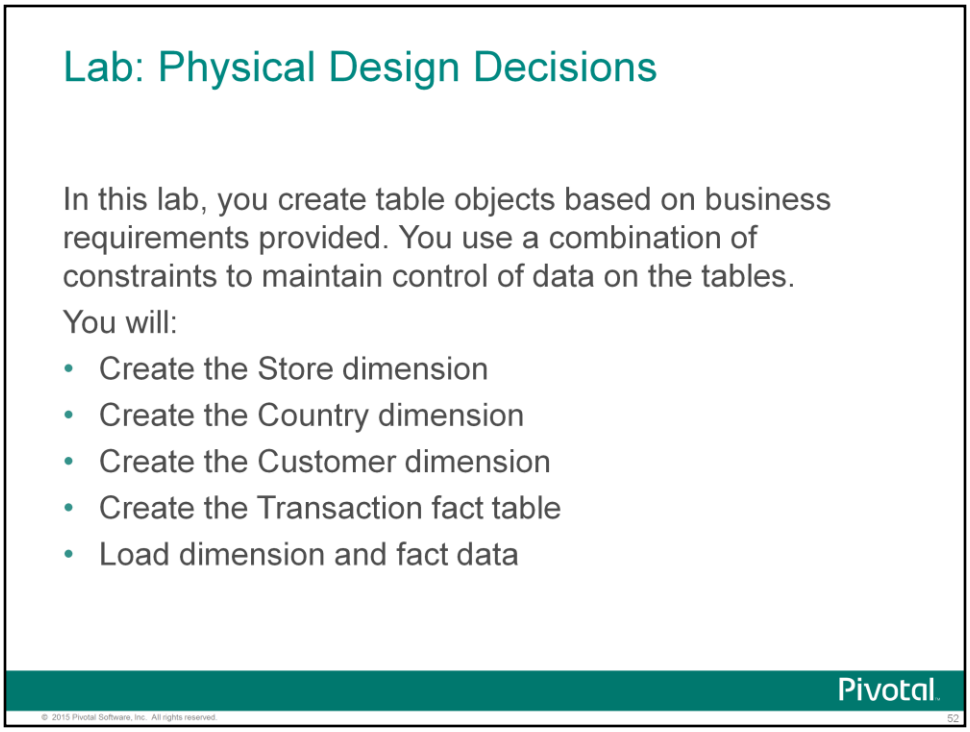
Constraints give you more control over the data in table, where data types may not provide enough of control over the data defined. If a user tries to enter data that violates the constraint you defined for a column, an error is raised.

The slide shows how to create supported types of constraints on the table and columns:

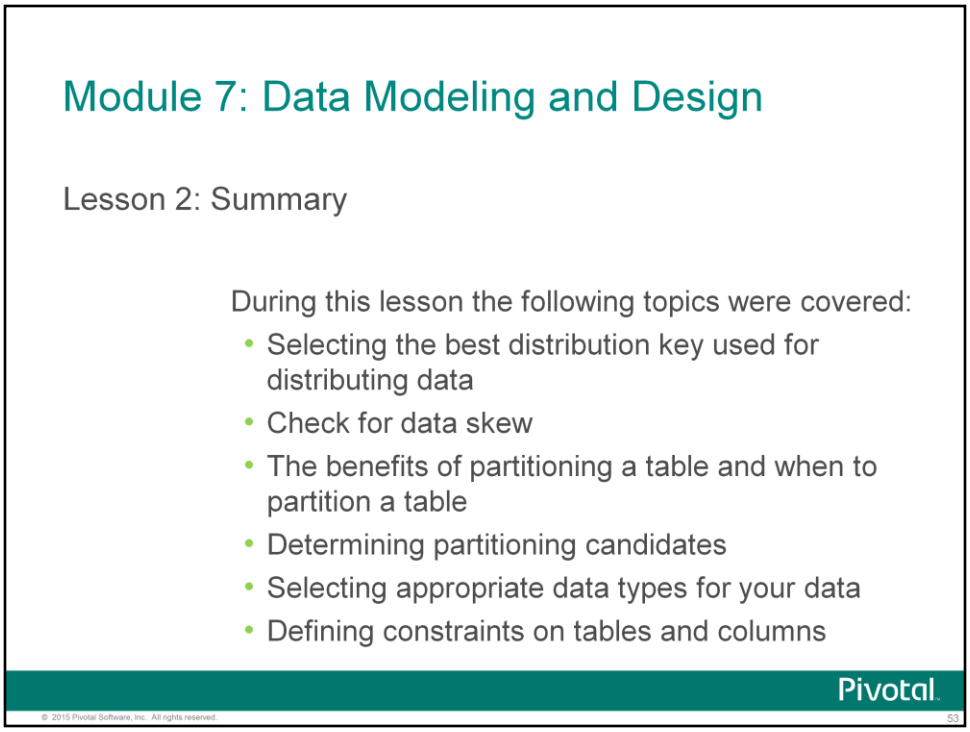
- In the first example, a `CHECK` constraint is placed on the price column of the products table. This ensures that for any price values entered must be greater than 0.
- The second example shows a `NOT NULL` constraint, where the `product_no` and `name` columns cannot be `NULL` or empty.



- The first example on this slide shows a `UNIQUE` constraint on the `product_no` column, where each value is distinct. The table must be hash distributed and cannot be distributed randomly. The constraint column must either be the same, or a superset, of the table's distribution key columns.
- The second example on this slide shows the `PRIMARY KEY` constraint on the product_no column. The `PRIMARY KEY` constraint enforces `UNIQUE` and `NOT NULL` constraints on the column. The table must be hash distributed and cannot be distributed randomly. The constraint column must either be the same, or a superset, of the table's distribution key columns.



In this lab, you create table objects based on business requirements provided. You use a combination of constraints to maintain control of data on the tables.

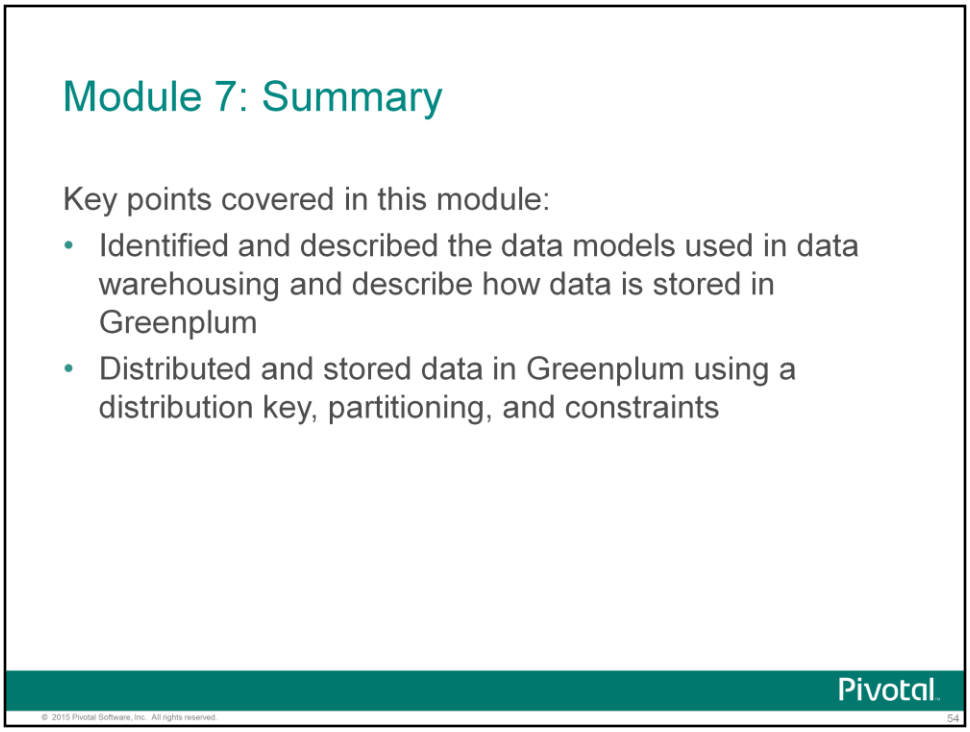


This lesson discussed how to design and define your tables to best suit the needs of its users, including selecting the best distribution key for distributed data. Distribution keys impact how the data is distributed across segments and therefore can impact scan rate times, particularly if data is skewed. Steps to check for skew are provided in the lesson.

When to partition a table and why you should do so is also reviewed in this lesson. Selecting ideal partitioning candidates is based on how your users access the data.

Selecting appropriate data types can impact physical storage size as well as join performance for queries, so data types should also be a consideration when defining tables and fields.

Lastly, defining constraints on tables and columns can help you to control the type of data within a table or partition and should be considered when your data needs to follow strict rules.



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

- Identify and describe the data models used in data warehousing and describe how data is stored in Greenplum.
- Distribute and store data in Greenplum using a distribution key, partitioning, and constraints.