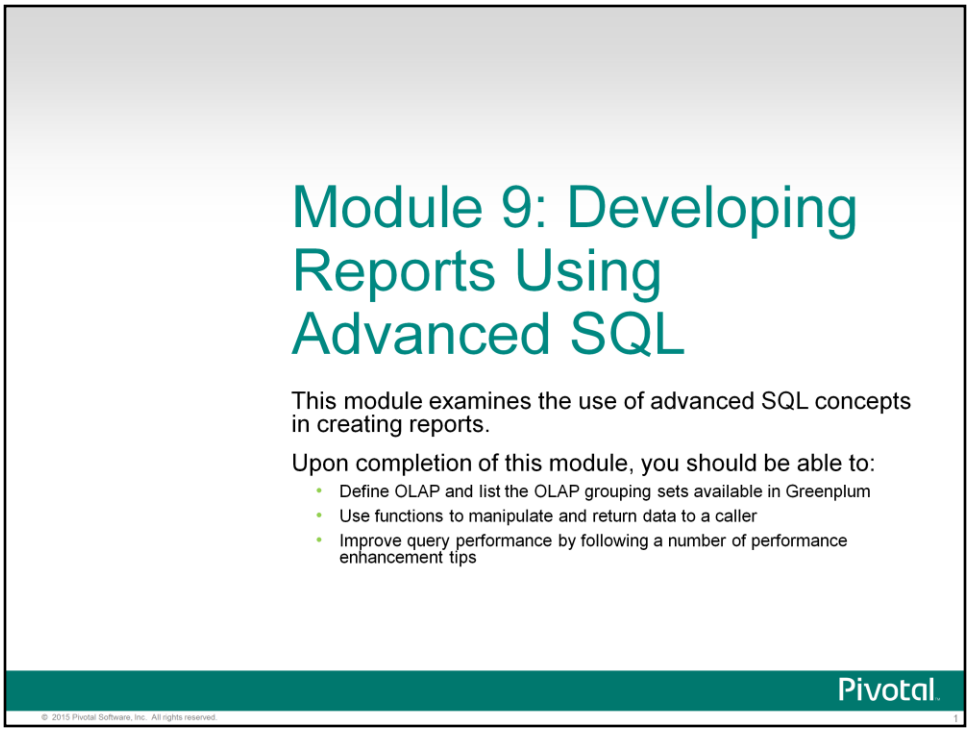
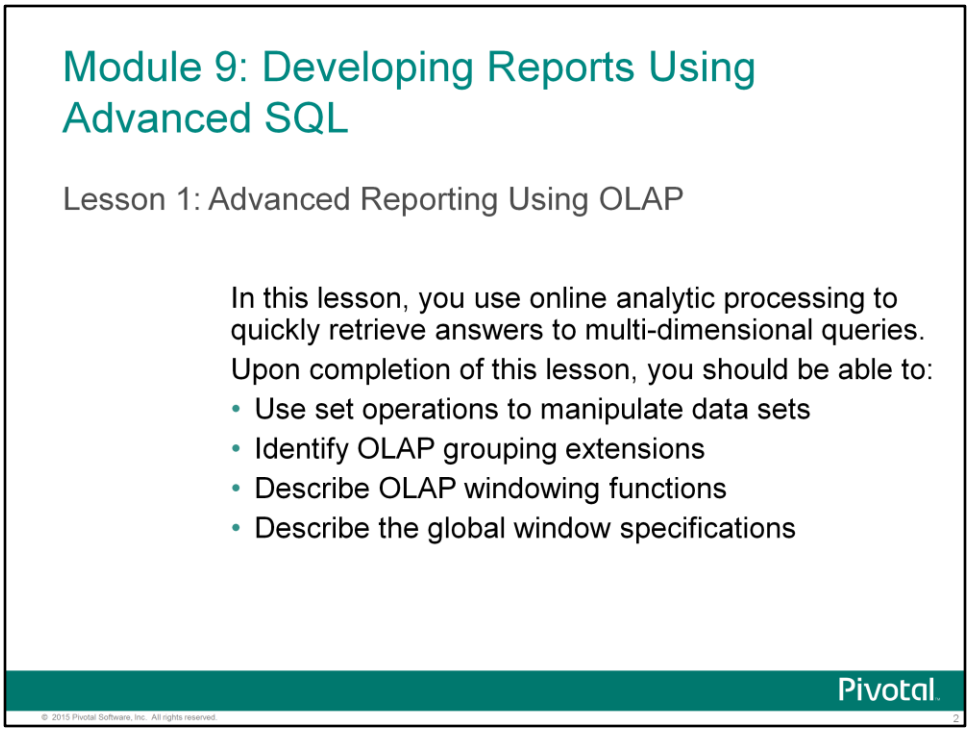



This module examines advanced reporting techniques, including OLAP and functions. OLAP groups and windowing functions offers benefits not only for coding, but also to improve performance.

In this module, you will:

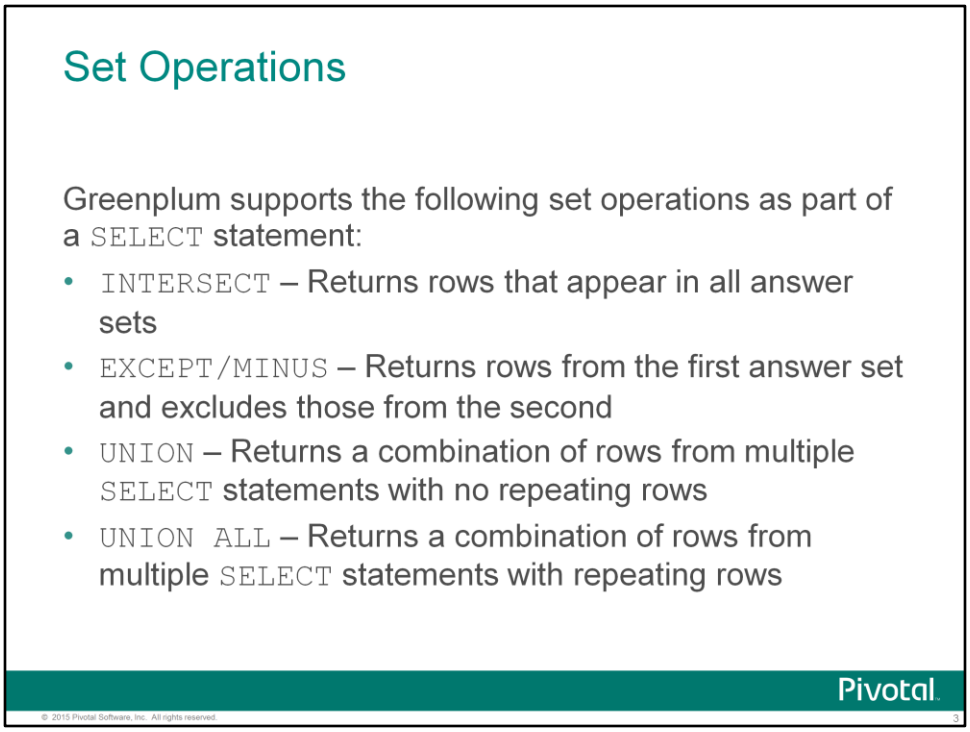
- Define OLAP and list the OLAP grouping sets available in Greenplum
- Use functions to manipulate and return data to a caller
- Improve query performance by following a number of performance enhancement tips




Online analytic process queries are used to answer multidimensional questions asked by users.

In this lesson, you will:

- Use set operations to manipulate data sets
- Identify OLAP grouping extensions
- Describe OLAP windowing functions
- Describe the global window specifications




Set operators:

- Manipulate the results sets of two or more queries by combining the results of individual queries into a single results set.
- Do not perform row level filtering.

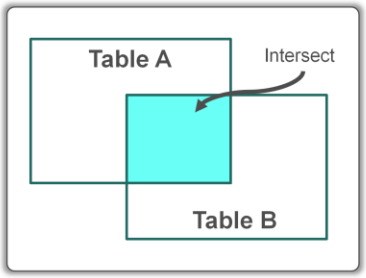
Set operations supported by Greenplum are:

- **`INTERSECT`** which returns rows that appear in all answer sets generated by individual `SELECT` statements.
- **`EXCEPT`** returns all rows from the first `SELECT` except for those which also selected by the second `SELECT`. This operation is the same as the `MINUS` operation.
- **`UNION`** combines the results of two or more `SELECT` statements. There will be no repeating rows.
- **`UNION ALL`** combines all the results of two or more `SELECT` statements. There may be repeating rows.

## Set Operations – INTERSECT

INTERSECT:

- Returns only the rows that appear in both SQL queries
- Removes duplicate rows




```
SELECT   t.transid
         c.custname
FROM     facts.transaction t
    JOIN dimensions.customer c
      ON c.customerid = t.customerid

INTERSECT

SELECT t.transid
       c.custname
FROM   facts.transaction t
  JOIN dimensions.customer c
    ON c.customerid = t.customerid
WHERE  t.transdate BETWEEN
       '2008-01-01' AND '2008-01-21'
```

Pivotal



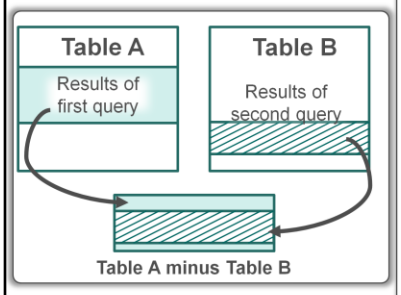
4

A set operation takes the results of two queries and returns only the results that appear in both result sets. Duplicate rows are removed from the final set returned.

## Set Operations – EXCEPT

EXCEPT:

- Returns all rows from the first SELECT statement
- Omits all rows that appear in the second SELECT statement




```
SELECT   t.transid
         c.custname
FROM     facts.transaction t
    JOIN dimensions.customer c
      ON c.customerid = t.customerid

  EXCEPT

SELECT t.transid
       c.custname
FROM   facts.transaction t
  JOIN dimensions.customer c
    ON c.customerid = t.customerid
WHERE  t.transdate BETWEEN
       '2008-01-01' AND '2008-01-21'
```

Pivotal



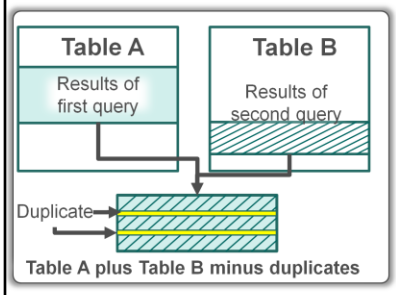
5

The EXCEPT set operation takes the distinct rows of the first query and returns all of the rows that do not appear in the result set of the second query.

## Set Operations – UNION

UNION:

- Combines rows from the first query with rows from the second query
- Removes duplicates or repeating rows




```
SELECT t.transid
       c.custname
FROM   facts.transaction t
  JOIN dimensions.customer c
    ON c.customerid = t.customerid
WHERE  t.transdate BETWEEN
       '2008-01-01' AND '2008-05-17'

  UNION

SELECT t.transid
       c.custname
FROM   facts.transaction t
  JOIN dimensions.customer c
    ON c.customerid = t.customerid
WHERE  t.transdate BETWEEN
       '2008-01-01' AND '2008-01-21'
```

Pivotal

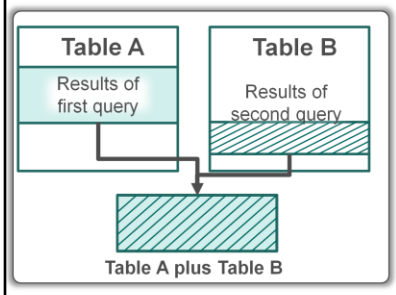


A union operation combines the results of the SELECT statement from the first table with the results from the query on the second table. The result set does not contain any repeating rows.

## Set Operations – UNION ALL

UNION ALL:

- Combines rows from the first query with rows from the second query
- Does not remove duplicate rows

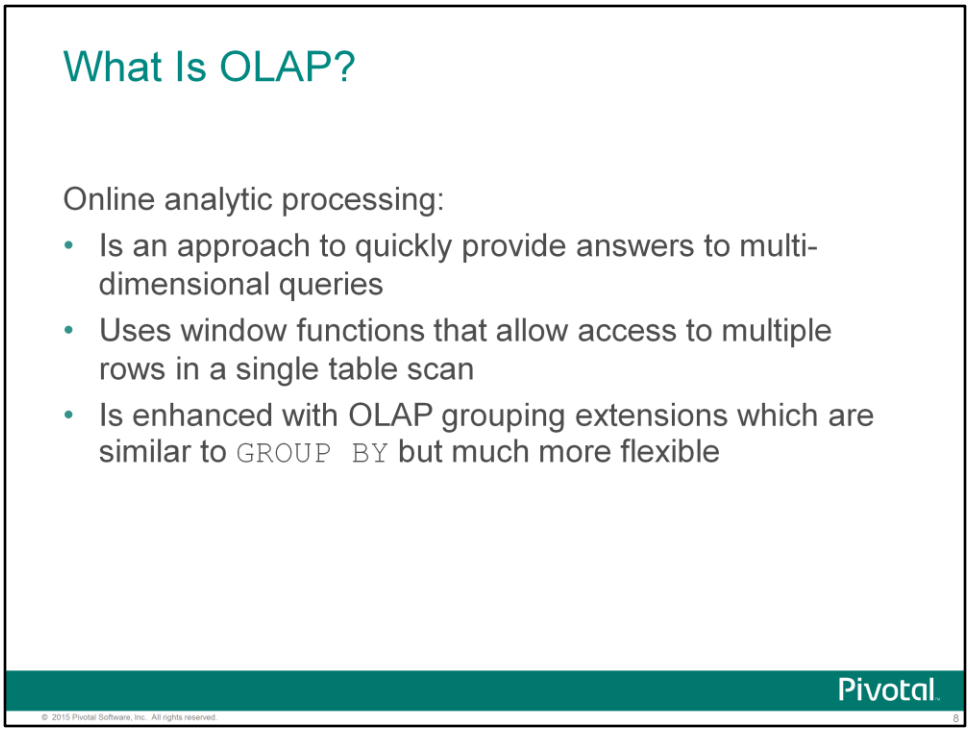



```
SELECT t.transid
       c.custname
FROM   facts.transaction t
  JOIN dimensions.customer c
    ON c.customerid = t.customerid
WHERE  t.transdate BETWEEN
       '2008-01-01' AND '2008-05-17'

  UNION ALL

SELECT t.transid
       c.custname
FROM   facts.transaction t
  JOIN dimensions.customer c
    ON c.customerid = t.customerid
WHERE  t.transdate BETWEEN
       '2008-01-01' AND '2008-01-21'
```

Pivotal

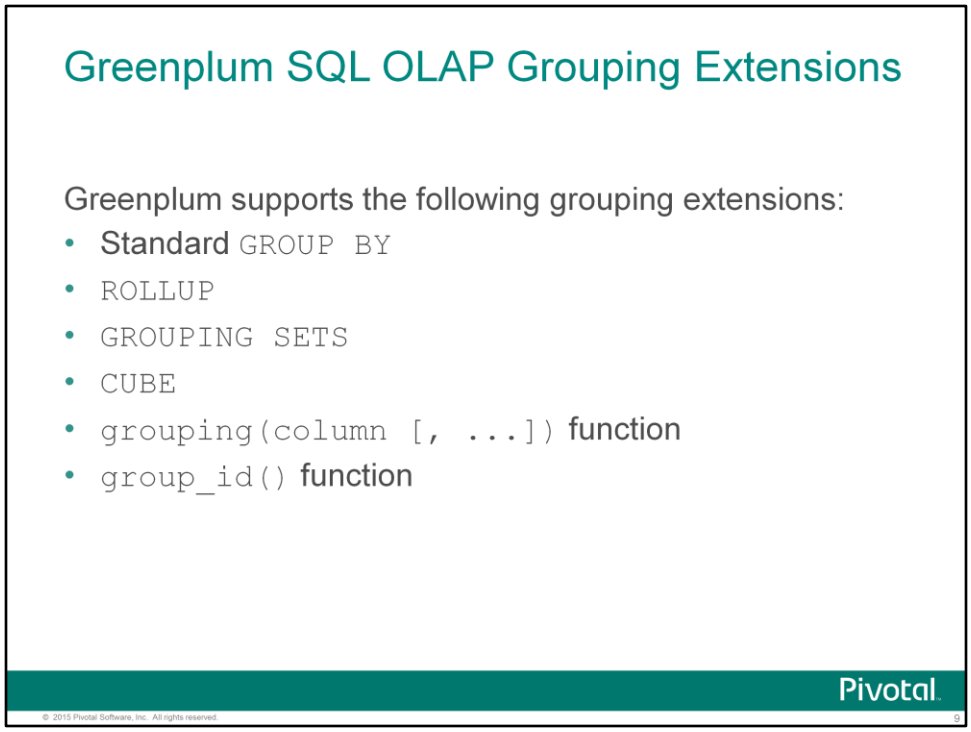


The UNION ALL set operation is like the UNION operation but it does not remove duplicate or repeating rows.



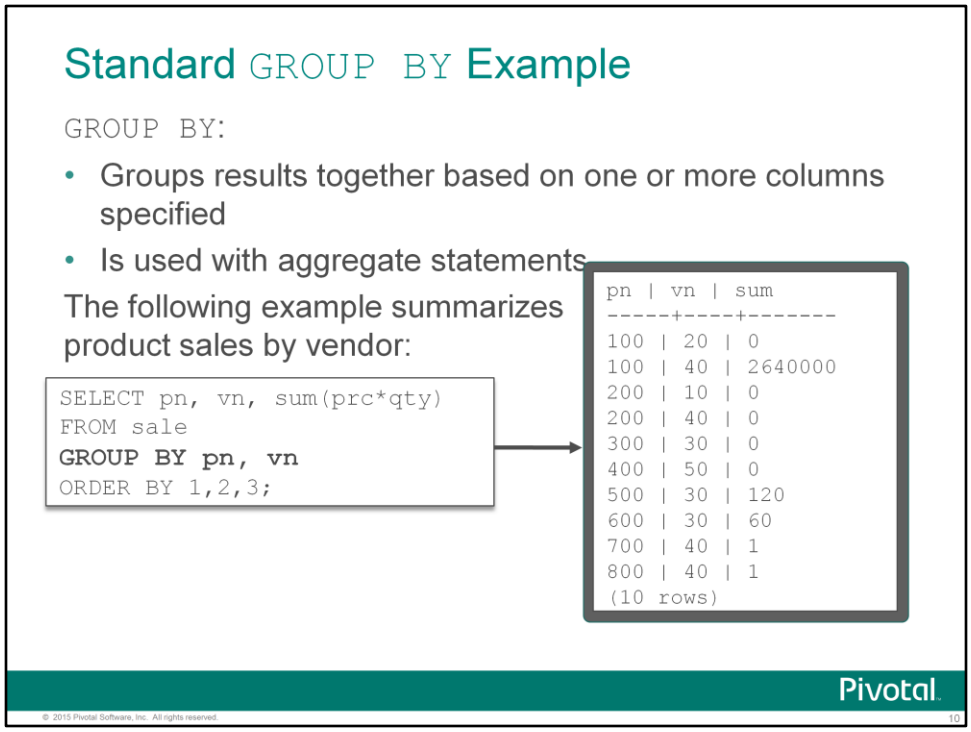

Part of broader category of BI, online analytic process is an approach to answer multi-dimensional queries. Databases configured for OLAP use a multidimensional data model that allows for complex analytical and ad-hoc queries.

Greenplum supports OLAP with window functions that provide access to multiple rows in a single table scan and grouping extensions, which provide flexibility when grouping result sets using the `GROUP BY` clause.



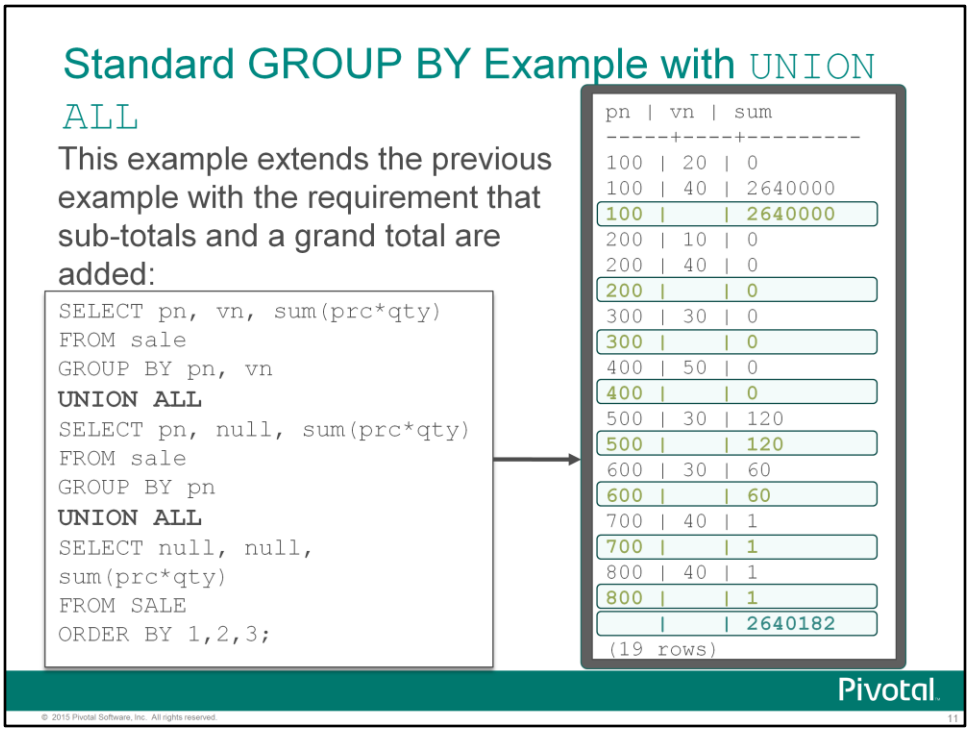

Greenplum introduced support for extensions to the standard `GROUP BY` clause, which is fully supported. These clauses can simplify the expression of complex groupings:

- **`ROLLUP`** – This extension provides hierarchical grouping.
- **`CUBE`** – Complete cross-tabular grouping, or all possible grouping combinations, is provided with this extension.
- **`GROUPING SETS`** – Generalized grouping is provided with the `GROUPING SETS` clause.
- **`grouping` function** – This clause helps identify super-aggregated rows from regular grouped rows.
- **`group_id` function** – This clause is used to identify duplicate rows in grouped output.

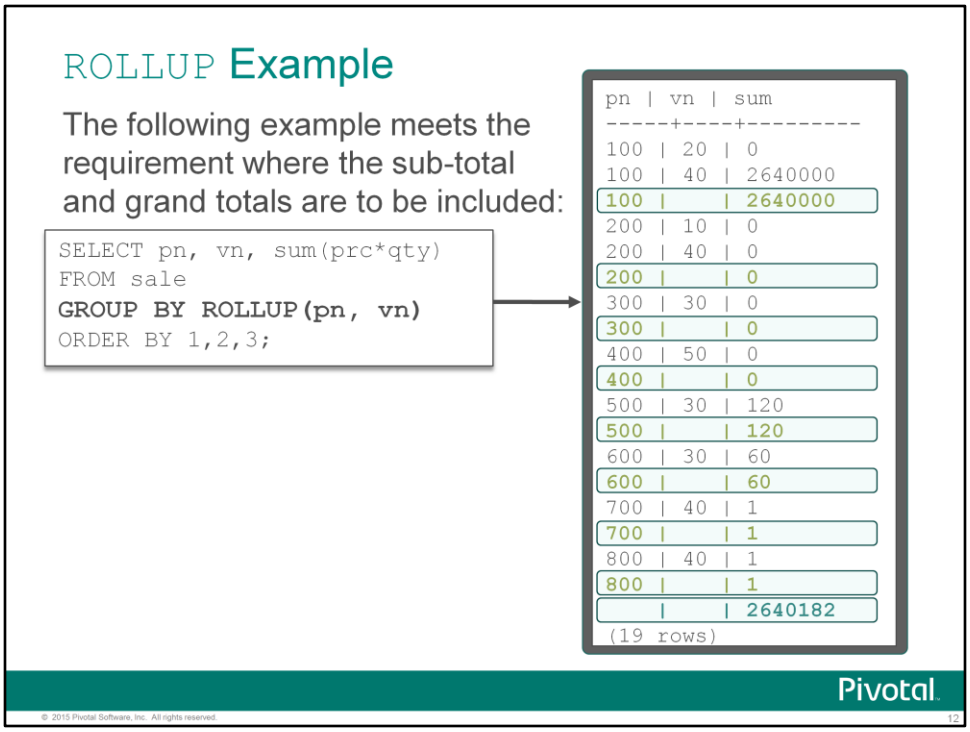



The standard `GROUP BY` clause groups results together based on one or more columns specified. It is used in conjunction with aggregate statements, such as `SUM`, `MIN`, or `MAX`. This helps to make the resulting data set much more readable to a user.

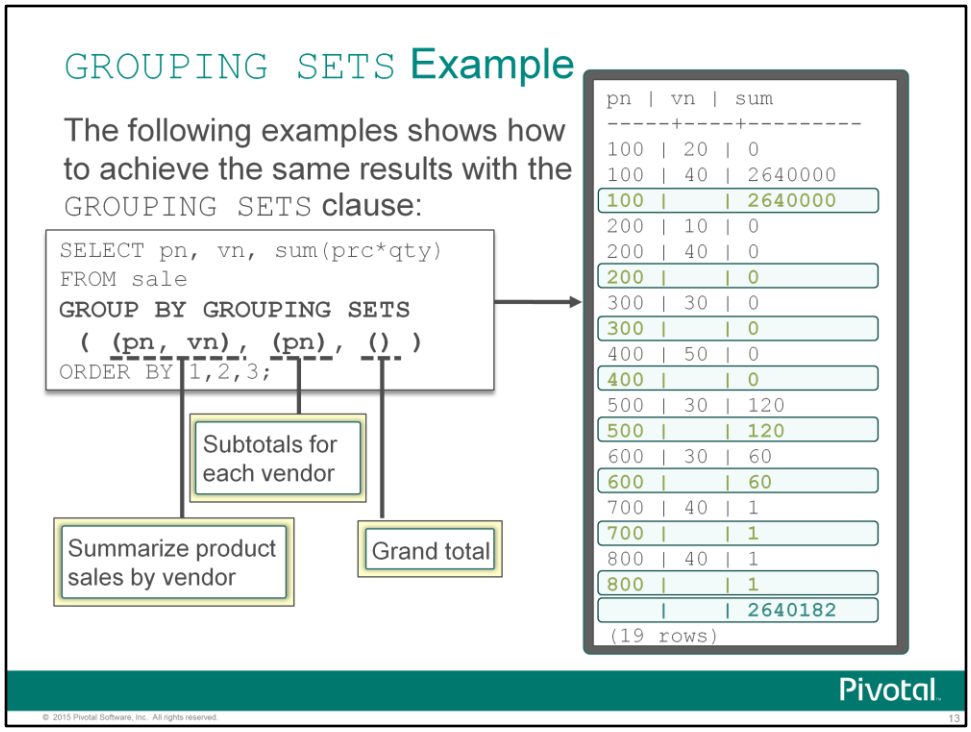
The slide shows an example of a standard `GROUP BY` clause used to summarize product sales by vendor.




In this follow-on example, the requirements for the query have been extended to include sub-totals and a grand total. You would need to use a `UNION ALL` to continue the grouping and provide for the additional requirements.

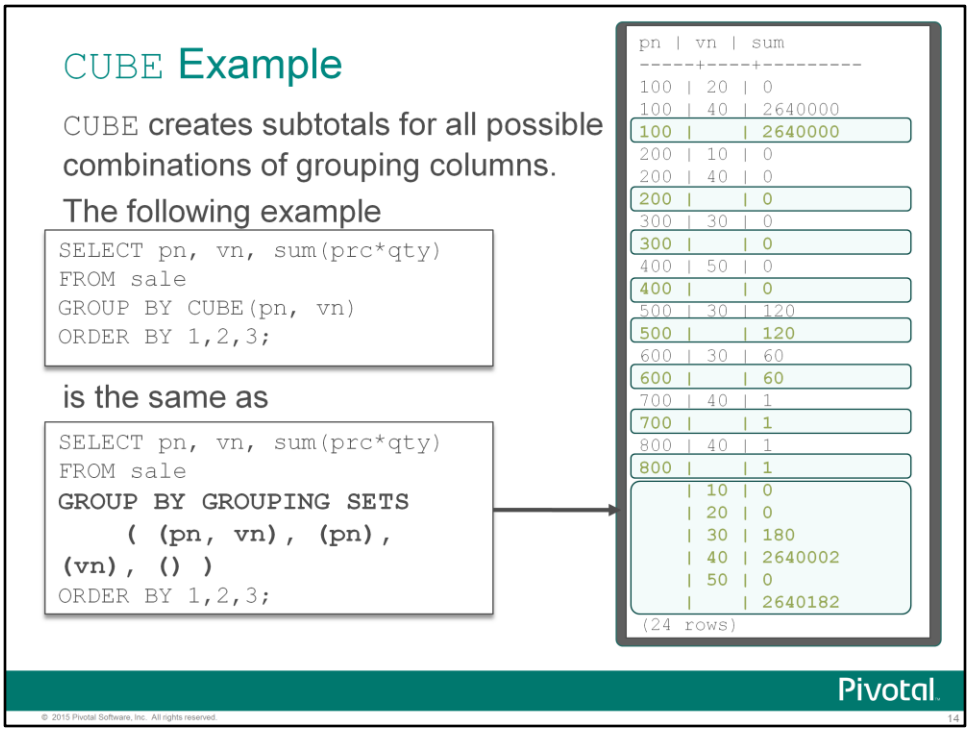



This slide meets the requirements provided in the previous slide, but uses the `ROLLUP` grouping extension. `ROLLUP` allows you to perform hierarchical grouping and helps to reduce the code.




The `GROUPING SETS` extension allows you to specify grouping sets. If you use the GROUPING SETS clause to meet the earlier requirements so that it produced the same output as `ROLLUP`, it would use the following groups:

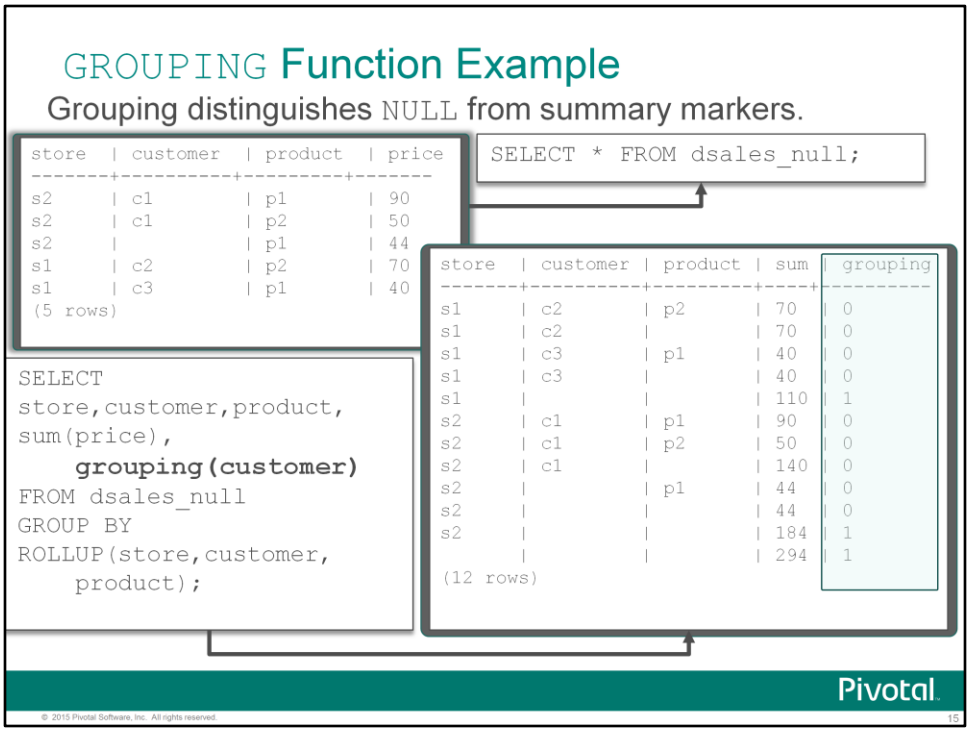
- `(pn,vn)` – This grouping summarizes product sales by vendor.
- `(pn)` – This grouping provides subtotal sales for each vendor.
- `()` – This grouping provides the grand total for all sales for all vendors.




A `CUBE` grouping creates subtotals for all of the possible combinations of the given list of grouping columns, or expressions. In terms of multidimensional analysis, CUBE generates all the subtotals that could be calculated for a data cube with the specified dimensions.

In the example shown on the slide, the additional grouping set of `(vn)` - subtotaling the sales by vendor, is included as part of the cube.

Note that *n* elements of a CUBE translate to 2*n* grouping sets. Consider using `CUBE` in any situation requiring cross-tabular reports. `CUBE` is typically most suitable in queries that use columns from multiple dimensions rather than columns representing different levels of a single dimension. For instance, a commonly requested cross-tabulation might need subtotals for all the combinations of month, state, and product.

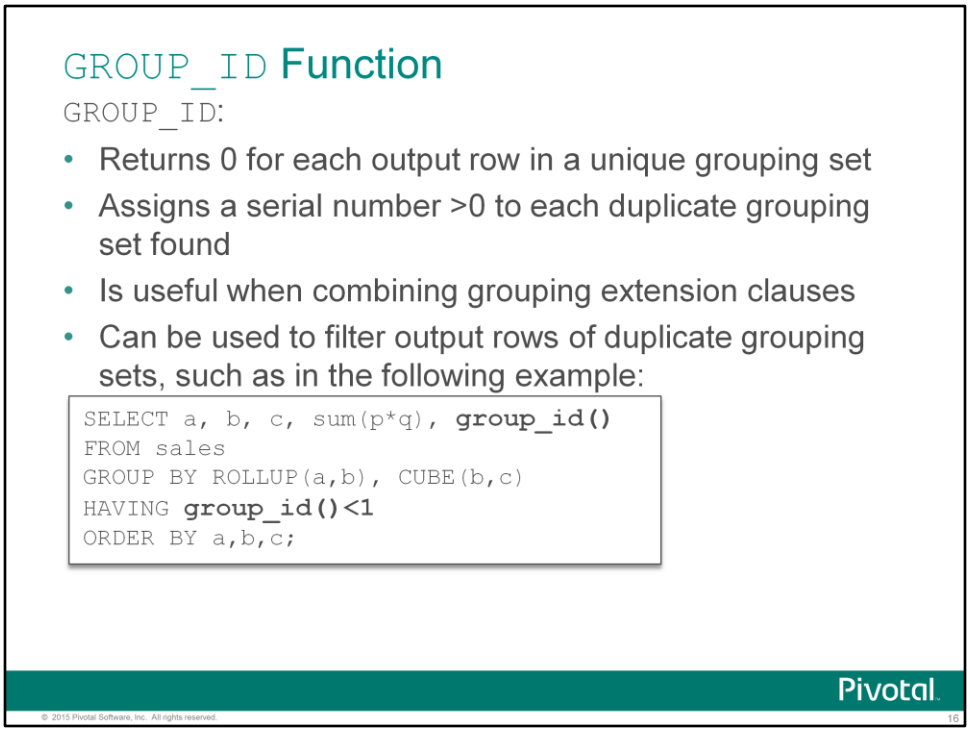



When you use grouping extensions to calculate summary rows, such as sub-totals and grand totals, the output can become confusing if the data in a grouping column contains `NULL` values. It is hard to tell if a row is supposed to be a subtotal row or a regular row containing a NULL.

In the example shown on the slide, one of the rows shown where the `customer` field is NULL. Without the grouping id, you could misinterpret the sum of 44 as a subtotal row for store 2.

The `GROUPING` function returns a result for each output row, where:

- 1 represents a summary row.
- 0 represents grouped rows.




In this example query, the combination of `ROLLUP` and `CUBE` produces:

- 11 grouping sets
- 7 `DISTINCT` grouping sets

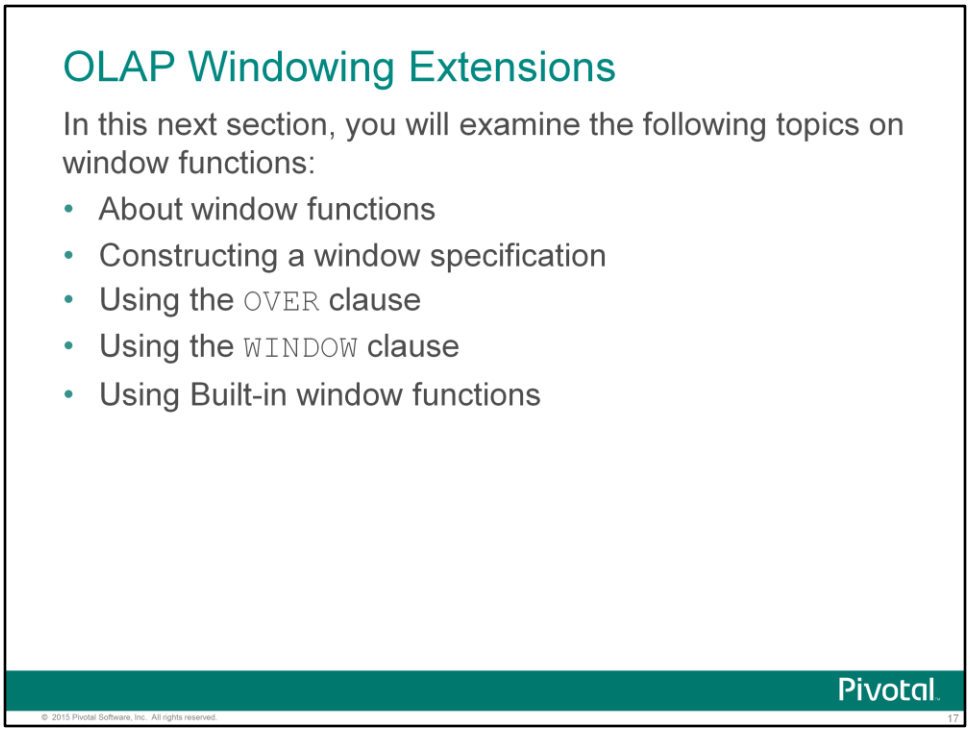
The `group_id` function can be used to filter out or identify duplicate grouping sets in the output.

`GROUP BY ROLLUP (a,b), CUBE (b,c)` is the same as:

- `GROUP BY GROUPING SETS ( (a,b), (a), () ), GROUPING SETS ( (b,c), (b), (c), () )`
- `GROUP BY GROUPING SETS ((a,b,b,c), (a,b,b), (a,b,c), (a,b), (a,b,c), (a,b), (a,c), (a), (b,c),(b),())`

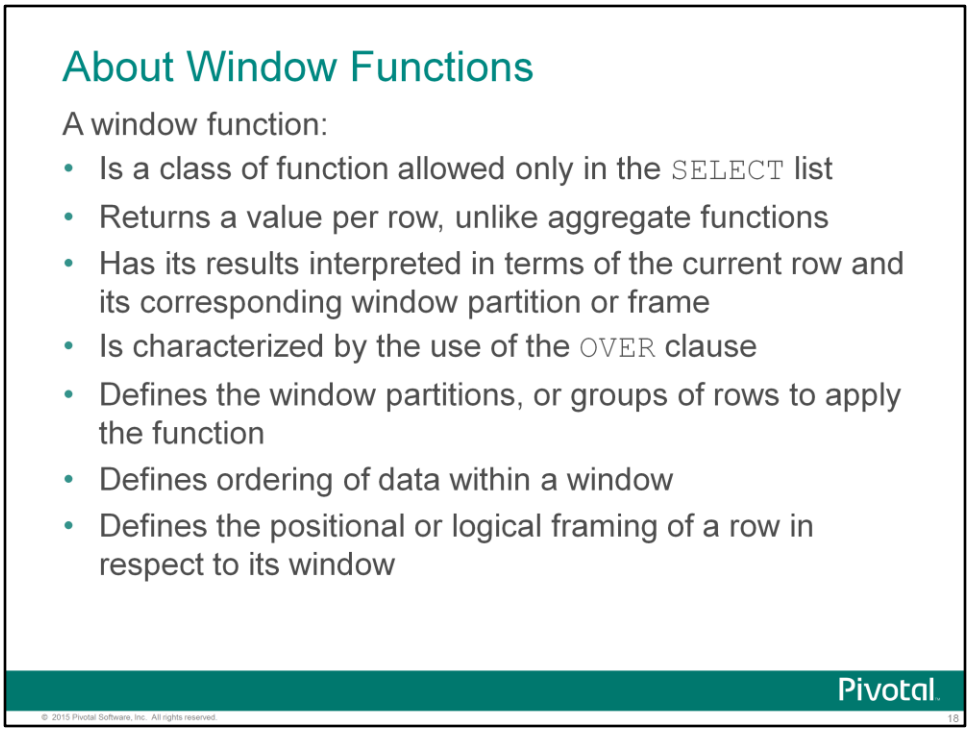
Where there are 11 total grouping sets but only 7 distinct grouping sets, where the groups are:

- `(a,b,b,c) = (a,b,c) = (a,b,c)`
- `(a,b,b) = (a,b) = (a,b)`
- `(a,c)`
- `(b,c)`
- `(a)`
- `(b)`
- `()`




Greenplum supports the concept of OLAP analytical functions, also known as window functions. In this next section, you will examine:

- Key concepts of window functions and how they are used.
- The SQL syntax used to construct the window specification used by window functions:
- The use of the `OVER` clause, which is a window function classified by the use of the special OVER clause, used to define the window specification.
- The use of the `WINDOW` clause, which is a convenient feature for defining and naming window specifications that can be used in one or more window function `OVER` clauses of a query.
- The built-in window functions provided in Greenplum Database.




Window functions are a new class of functions introduced in Greenplum.

Window functions allow application developers to more easily compose complex OLAP queries using standard SQL commands. For example:

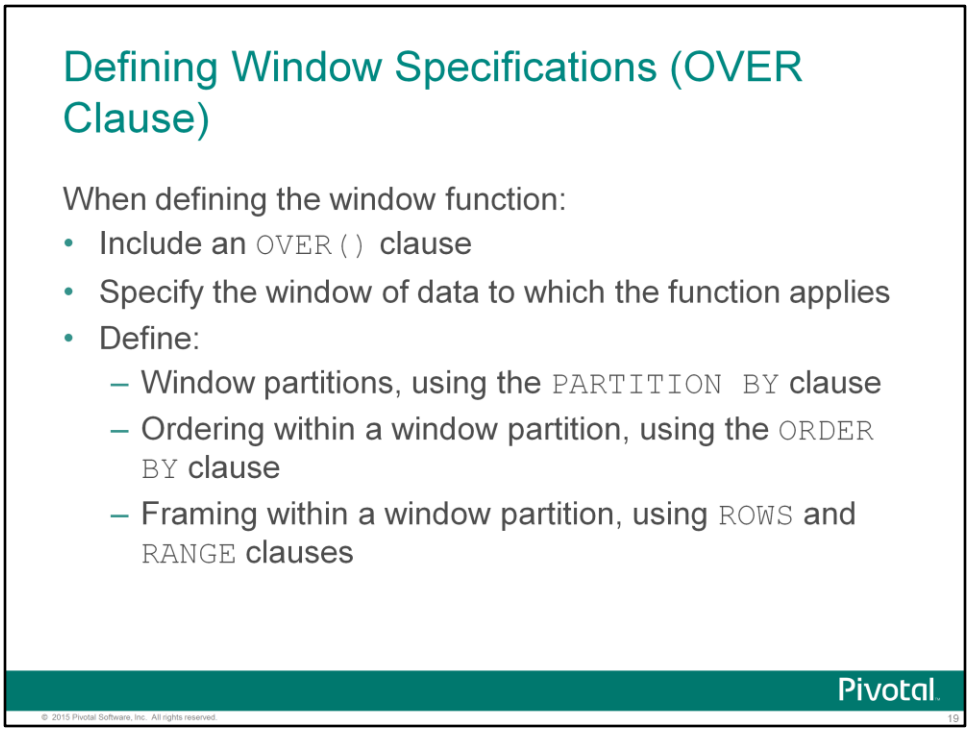
- Moving averages or sums can be calculated over various intervals.
- Aggregations and ranks can be reset as selected column values change.
- Complex ratios can be expressed in simple terms.

Window functions can only be used in the `SELECT` list, between the `SELECT` and `FROM` keywords of a query.

Unlike aggregate functions, which return a result value for each group of rows, window functions return a result value for every row, but that value is calculated with respect to the rows in a particular window partition (grouping) or window frame (row position within the window).

What classifies a function as a window function is the use of an `OVER` clause. The `OVER` clause defines the window of data to which the function will be applied. There are three characteristics of a window specification:

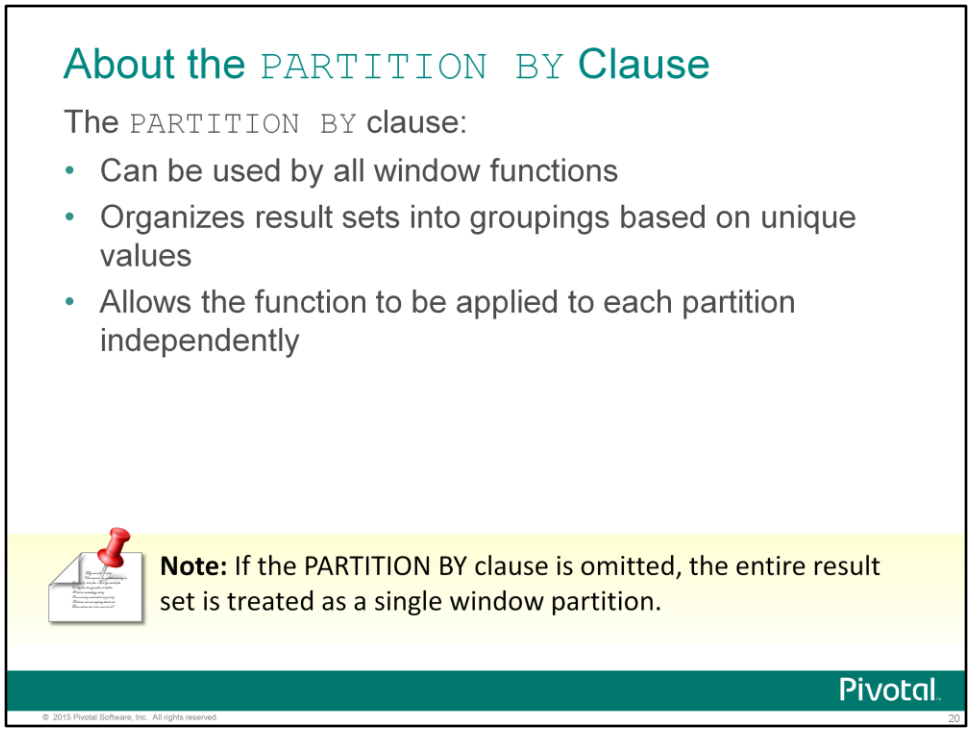
- Partitions (groupings) – a window function calculates the results for a row in respect to its partition.
- Ordering of rows within a window partition – some window function such as RANK require ordering.
- Framing – for ordered result sets, you can define a window frame that analyzes each row with respect to the rows directly above or below it.




All window functions must have an `OVER()` clause. The window function specifies the window of data to which the function applies.

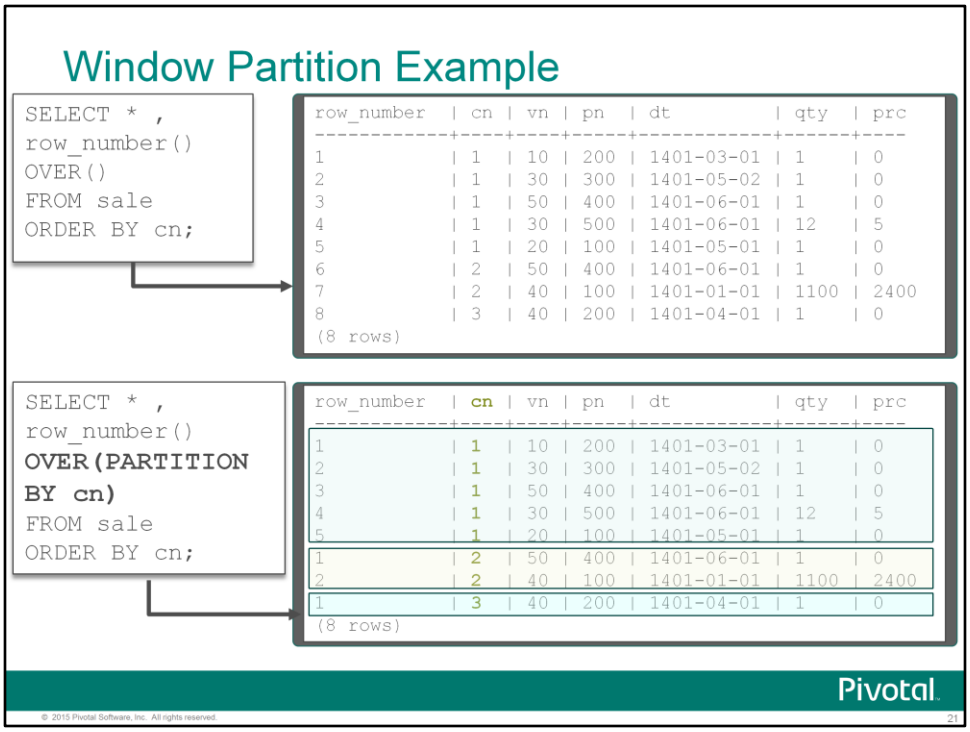
It defines:

- Window partitions using the `PARTITION BY` clause.
- Ordering within a window partition using the `ORDER BY` clause).
- Framing within a window partition (`ROWS/RANGE` clauses).




The `PARTITION BY` clause:

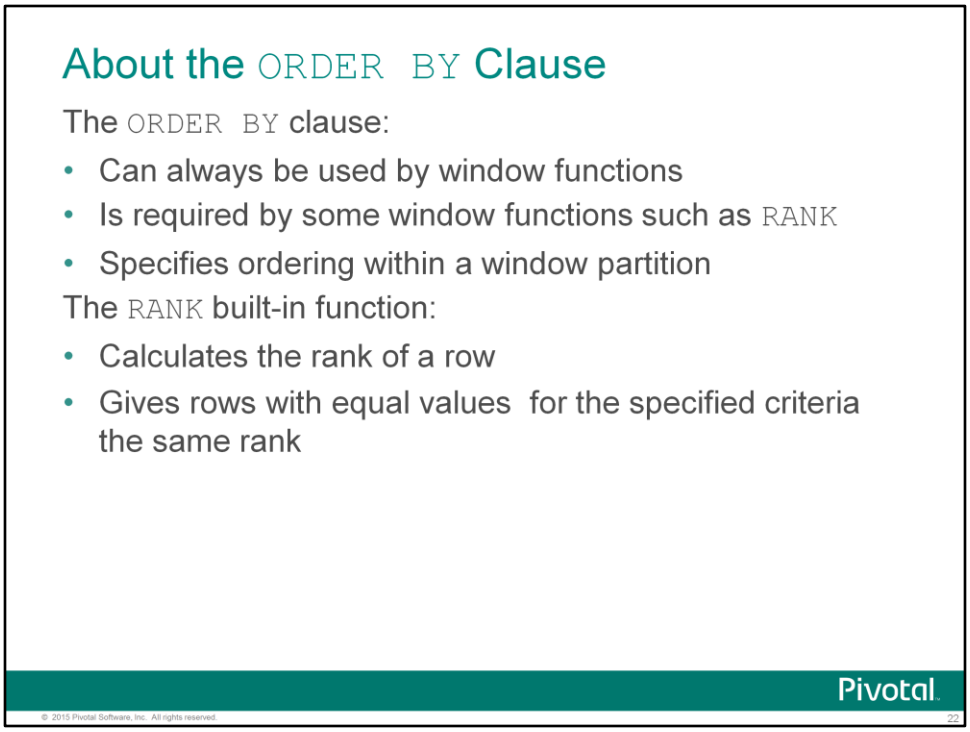
- Can be used by all window functions. However, it is not a required clause. Windows that do not use the `PARTITION BY` clause present the entire result set as a single window partition.
- Organizes the result set into groupings based on the unique values of the specified expression or column.
- Allows the function to be applied to each partition independently.




The example on the slide uses the `row_number` window function. This function returns a row number for each unique row in the result set.

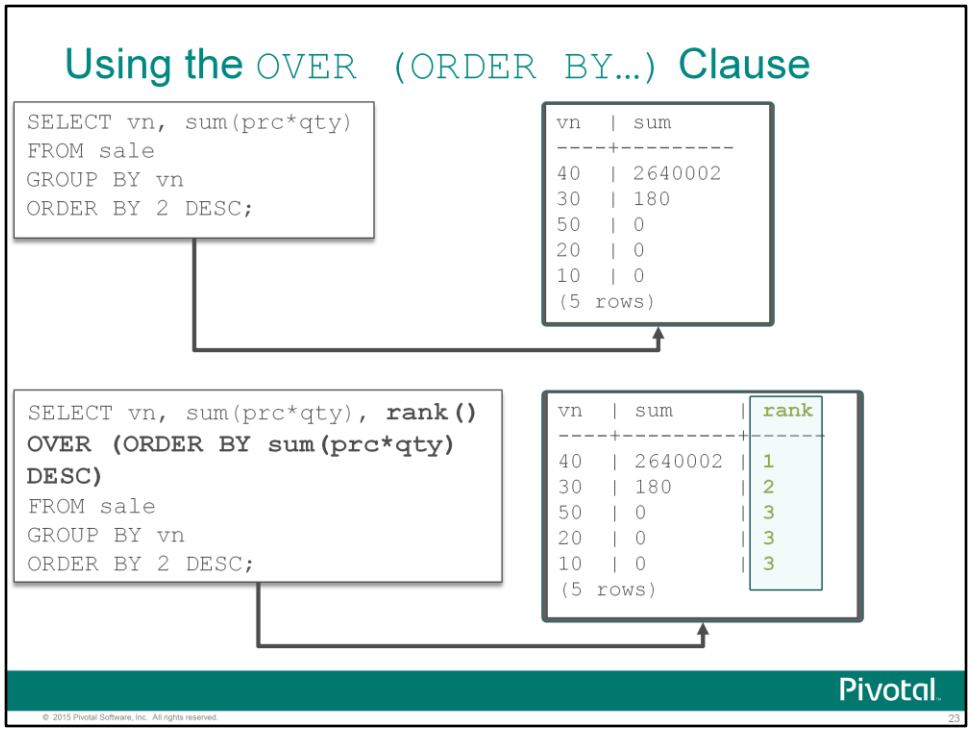
In the first example, the `OVER` clause does not have a `PARTITION BY`. The entire result set is treated as one window partition.

In the second example, the window is partitioned by the customer number. Note that the result of row number is calculated within each window partition.




The `ORDER BY` clause is used to order the resulting data set based on an expression or column. It is always allowed in windows functions and is required by some window functions, including `RANK`. The `ORDER BY` clause specifies ordering within a window partition.

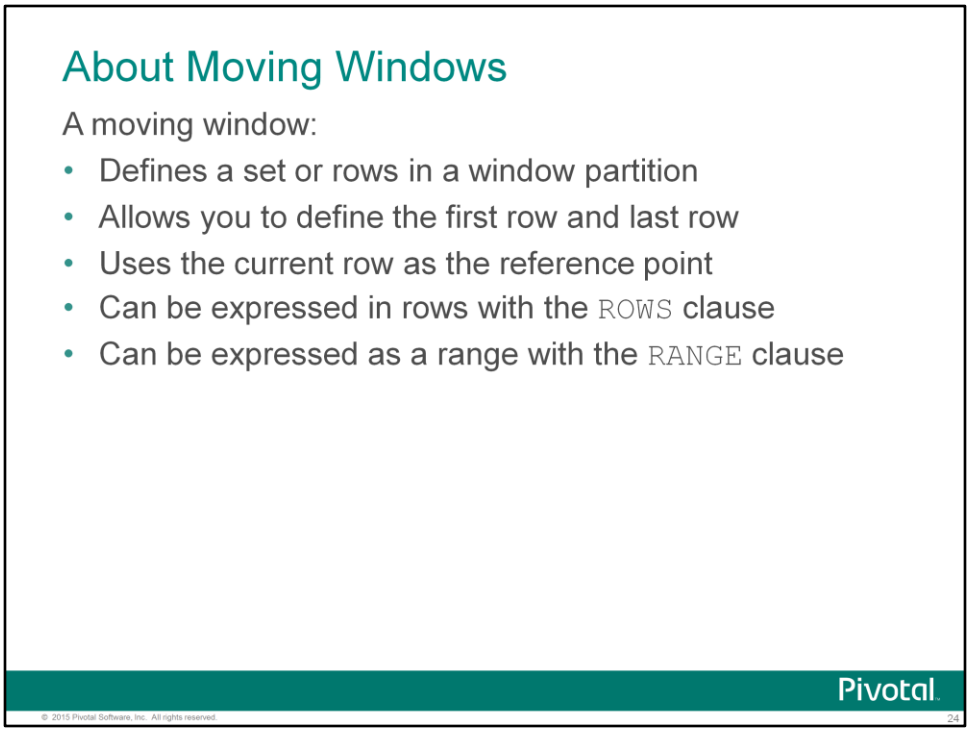
The RANK function is a built-in function that calculates the rank of a row in an ordered group of values. Rows with equal values for the ranking criteria receive the same rank. The number of tied rows are added to the rank number to calculate the next rank value. In this case, ranks may not be consecutive numbers.




The slide shows an example of two queries that rank vendors by sales totals.

The first query shows a window function grouped on the vendor column, `vn`.

The second query uses the `RANK` function to output a ranking number for each row. Note that the `PARTITION BY` clause is not used in this query. The entire result is one window partition. Also, do not confuse `ORDER BY` of a window specification with the `ORDER BY` of a query.




A moving or rolling window defines a set of rows within a window partition. When you define a window frame, the window function is computed with respect to the contents of this moving frame, rather than against the fixed contents of the entire window partition. Window frames can be row-based, represented by the `ROWS` clause, or value based, represented by a `RANGE`.

When the window frame is row-based, you define the number of rows offset from the current row. If the window frame is range-based, you define the bounds of the window frame in terms of data values offset from the value in the current row.

If you specify only a starting row for the window, the current row is used as the last row in the window.

## About Moving Windows (Cont)

A moving window:

- Is defined as part of a window with the ORDER BY clause as follows:

```
WINDOW window_name AS (window_specification)
```

where window_specification can be:

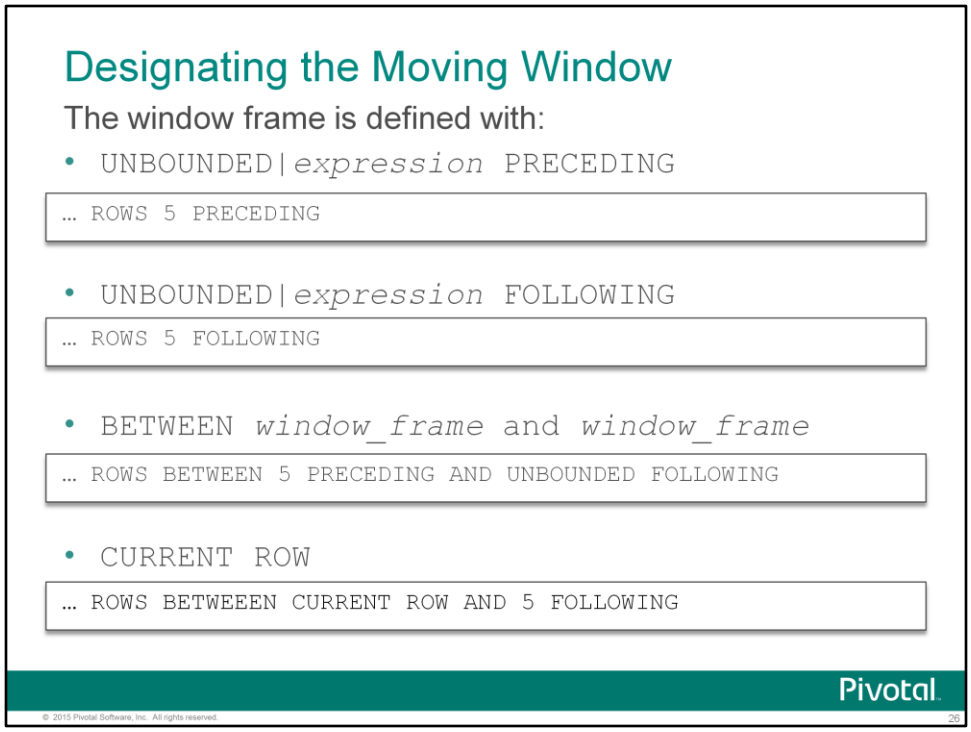
```
[window_name]
[PARTITION BY expression [, ...]]
[ORDER BY expression [ASC | DESC | USING operator] [, ...]
[{RANGE | ROWS}
{ UNBOUNDED PRECEDING
| expression PRECEDING
| CURRENT ROW
| BETWEEN window_frame_bound AND window_frame_bound }]]
```

Pivotal





About Moving Windows (Continued)

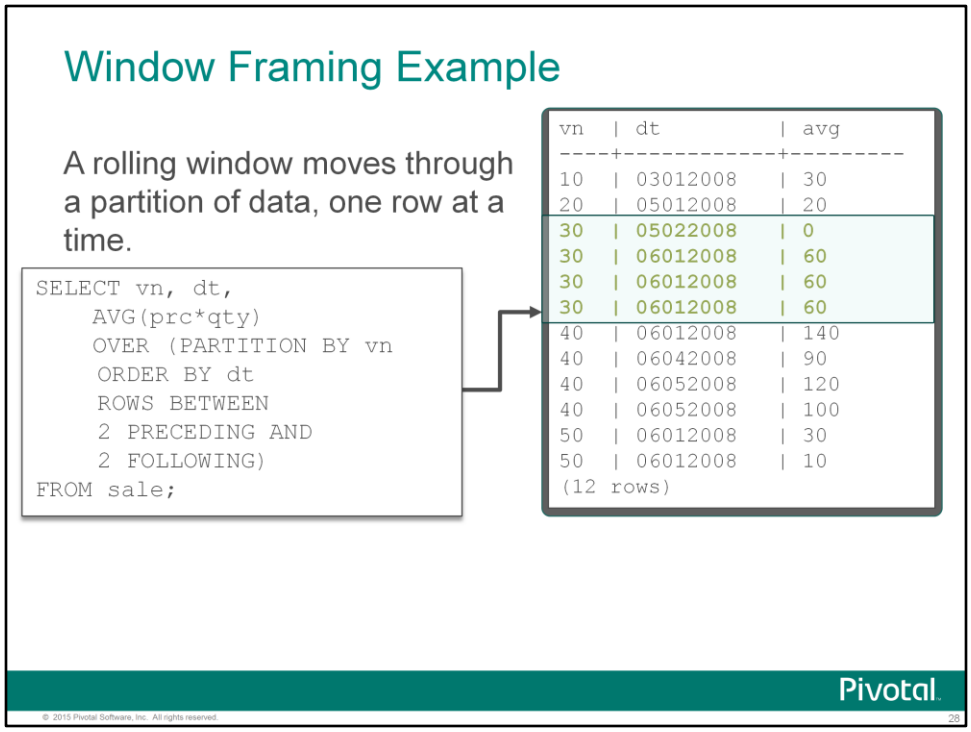



The window frame can be defined as:

- **`UNBOUNDED` or *`expression`* `PRECEDING`** – This clause defines the first row of the window using the current row as a reference point. The starting row is expressed in terms of the number of rows preceding the current row. If you define the window frame for a `ROWS` window frame as `5 PRECEDING`, the window frame starts at the fifth row preceding the current row. If the definition is for a RANGE window frame, the window starts with the first row whose ordering column value precedes that of the current row by 5. If the term `UNBOUNDED` is used, the first row of the partition acts as the first row of the window.
- **`UNBOUNDED` or *`expression`* `FOLLOWING`** – This clause defines the last rows of the window using the current row as a reference point. Similar to `PRECEDING`, the last row is expressed in terms of the number or rows following the current row. Either an expression or the term `UNBOUNDED` can be used to identify the last rows. If `UNBOUNDED` is used, the last row in the window is the last row in the partition.

**Designating the Moving Window (Continued)**

- **`BETWEEN` *`window_frame_bound`* `AND` *`window_frame_bound`*** – This clause defines the first and last rows of the window, using the current row as a reference point. The first and last rows are expressed in terms of the number or rows preceding and following the current row, respectively. You can use other window frame syntax to define the bound. For example, `BETWEEN 5 PRECEDING AND 5 FOLLOWING` defines a window frame where the previous 5 rows and the next 5 rows from the current row are included in the moving window.
- **`CURRENT ROW`** – This clause references the current row in the partition. If a window frame is defined as `BETWEEN CURRENT ROW AND 5 FOLLOWING`, the window is defined starting with the current row and ending with the next 5 rows.

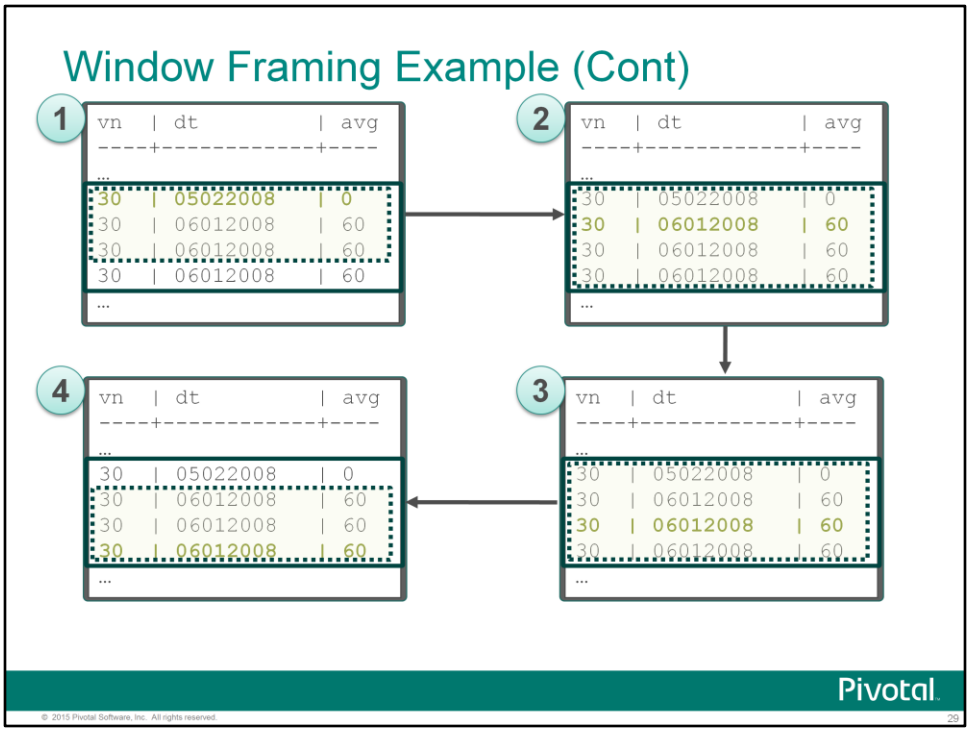



While window framing clauses require on `ORDER BY` clause, not all window functions allow framing.

The `ROWS` and `RANGE` clauses specify a positional or logical rolling window that moves through a window partition of data.

In the example shown on the slide, the rolling frame applies to its partition, in this case, `vendor`, and ordering within that partition, `date`.

The example shows positional framing using the `ROWS BETWEEN` clause where the result is interpreted with respect to the `CURRENT ROW` position in the partition. The focus of the window frame moves from row to row within its partition only.




The focus of the frame moves from the first selectable row in the window partition using the criteria, 2 preceding and 2 following, for the rolling window.

## Built-in Window Functions

| Built-in Function | Description |
|---|---|
| `cume_dist()` | Calculates the cumulative distribution of a value in a group of values. Rows with equal values always evaluate to the same cumulative distribution value. |
| `dense_rank()` | Computes the rank of a row in an ordered group of rows without skipping rank values. Rows with equal values are given the same rank value. |
| `first_value(`*expr*`)` | Returns the first value in an ordered set of values. |
| `lag(expr [,`*offset*`] [,default])` | Provides access to more than one row of the same table without doing a self join. Given a series of rows returned from a query and a position of the cursor, LAG provides access to a row at a given physical offset prior to that position. If offset is not specified, the default offset is 1. default sets the value that is returned if the offset goes beyond the scope of the window. If default is not specified, the default value is null. |

**Note:** Any aggregate function used with the `OVER` clause can also be used as a window function.

Pivotal





The slide shows built-in window functions supported within Greenplum. These built-in functions require an `OVER` clause.

For more detailed information on the functions, refer to the *Greenplum Database Administrator Guide*.

## Built-in Window Functions (Cont)

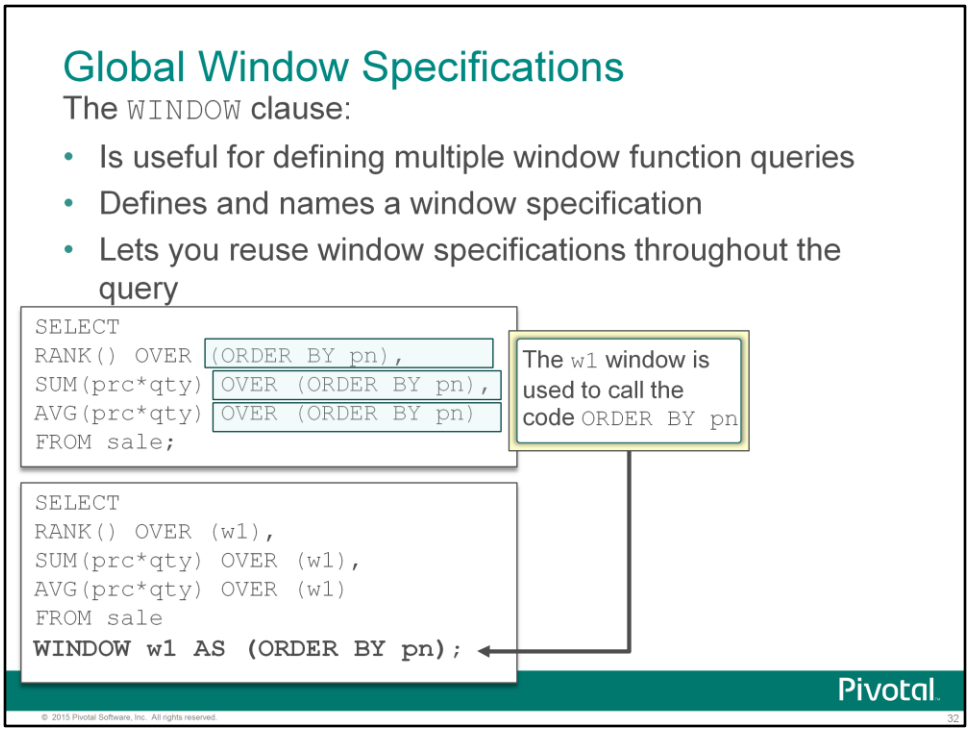
| Built-in Function | Description |
|---|---|
| `last_value(expr)` | Returns the last value in an ordered set of values. |
| `lead(expr [,offset] [,default])` | Provides access to more than one row of the same table without doing a self join. Given a series of rows returned from a query and a position of the cursor, LAG provides access to a row at a given physical offset after that position. If offset is not specified, the default offset is 1. default sets the value that is returned if the offset goes beyond the scope of the window. If default is not specified, the default value is null. |
| `ntile(expr)` | Divides an ordered dataset into a number of buckets (as defined by expr) and assigns a bucket number to each row. |
| `percent_rank()` | Calculates the rank of a hypothetical row R minus 1, divided by 1 less than the number of rows being evaluated (within a window partition). |
| `row_number()` | Assigns a unique number to each row to which it is applied (either each row in a window partition or each row of the query). |

Pivotal



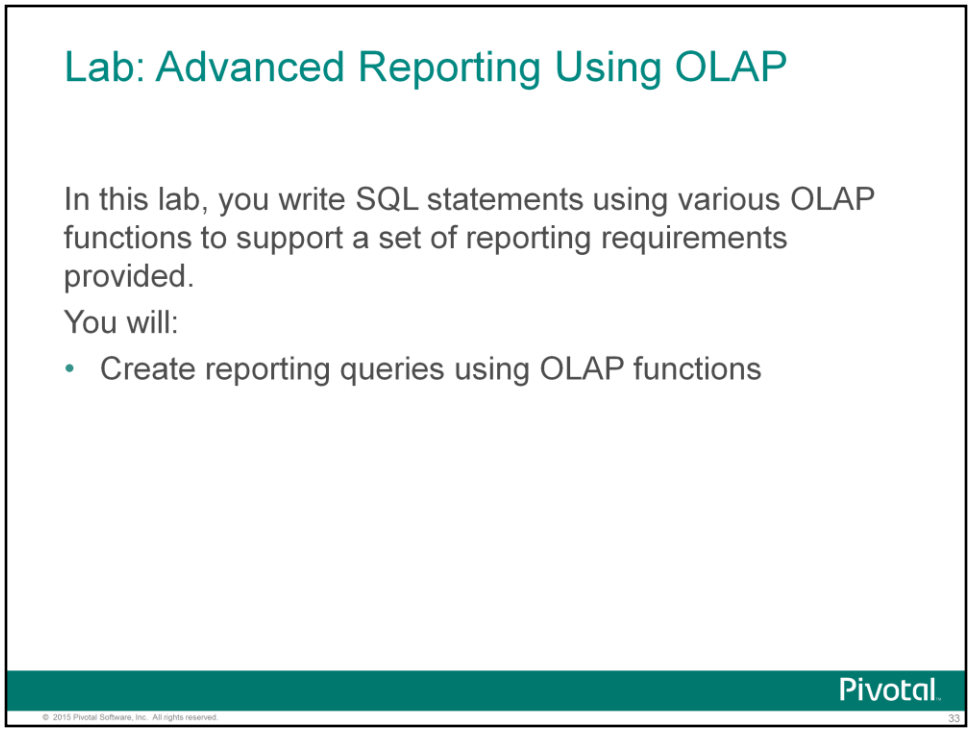


Built-in Window Functions (Continued)

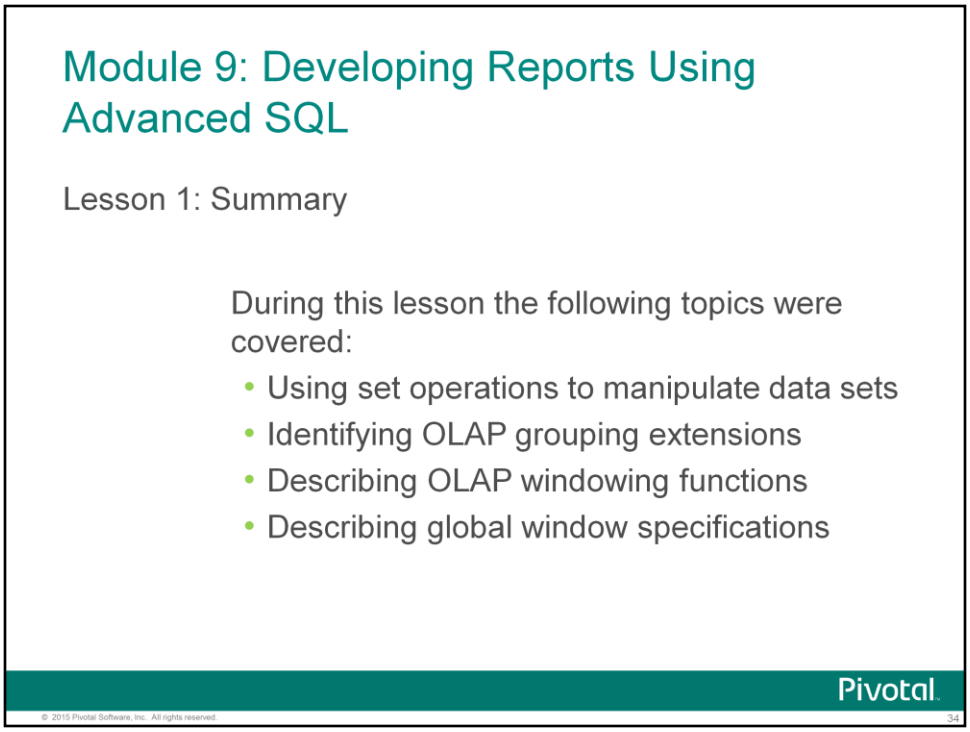



The `WINDOW` clause is a convenient feature that allows you to define and name a window specification once and then refer to it multiple times throughout the query.

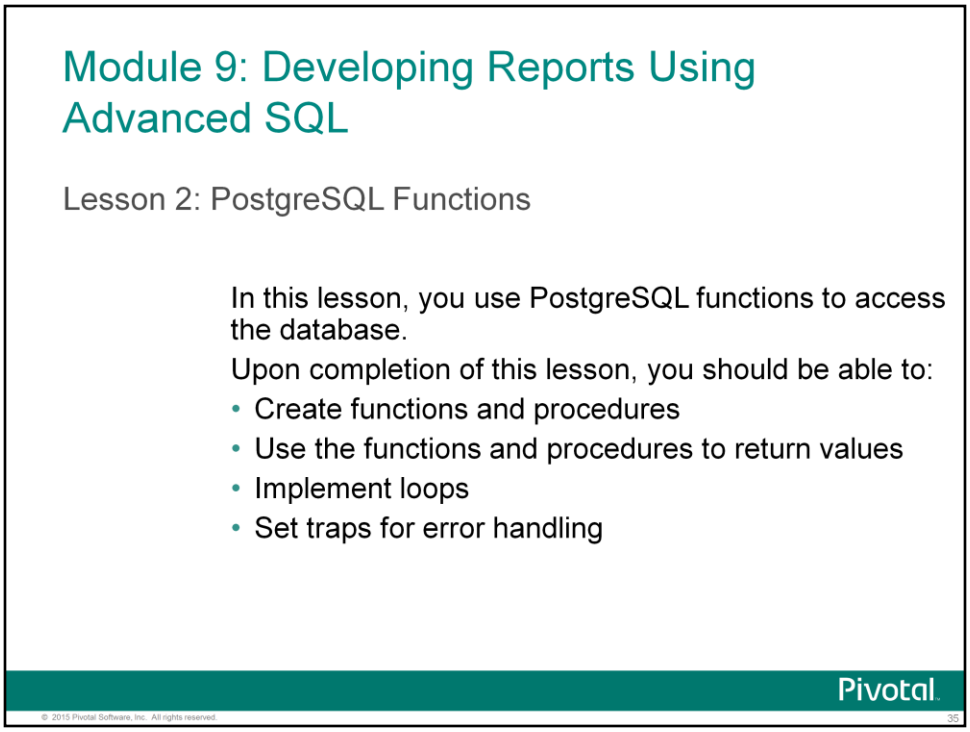
In the example shown on the slide, several window functions are using the same window specification in the `OVER` clause. You can use the `WINDOW` clause to define the window specification once, and then refer to it by name. In this example, the window specification is called `w1`.




In this lab, you write SQL statements using various OLAP functions to support a set of reporting requirements provided.



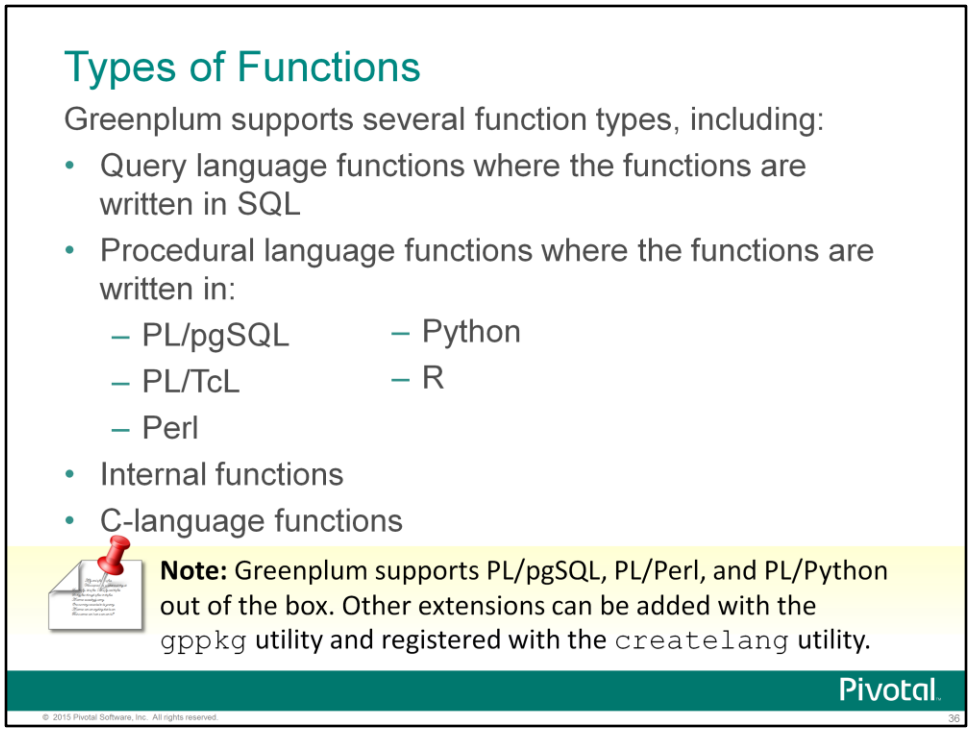

This lesson covered common set operations, OLAP grouping, and OLAP windowing functions that are used to present and summarize data for reporting purposes. These operations can greatly reduce the number of lines of SQL code required to return the same results and improve overall performance of the queries.




Functions allow you to define and name a block of code that you can call by name during your development effort. Functions can reduce development time and provide you with a means of returning a value to a caller.

In this lesson, you will:

- Create functions and procedures
- Use the functions and procedures to return values
- Implement loops
- Set traps for error handling




Greenplum supports a variety of methods for developing functions, including:

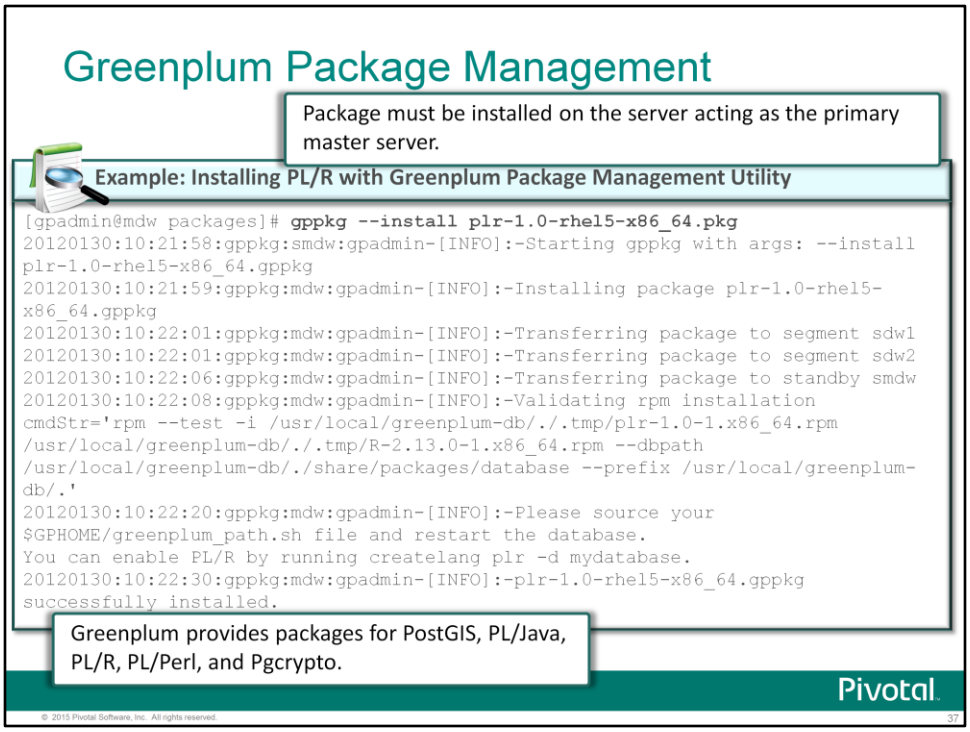
- Query language support for functions developed in SQL.
- Procedural language support for functions written in languages such as PL/PGSQL, which is a subset of PL/SQL, PL/TcL, Perl, Python, and R, a programming language for statistical computing and graphics.
- Internal functions
- C-language functions

On installing Greenplum, PL/pgSQL is installed. All other languages can be installed to work with Greenplum.

Languages can be added to Greenplum with the Greenplum Package Manager utility and registered with the language handler utility, `createlang`. Greenplum:

- Has installed packages for PL/pgSQL, PL/Perl, and PL/Python.
- Has already registered PL/pgSQL in all databases by default, so no further action is required to enable it.
- Includes a language handler for PL/R, but the package is not pre-installed.
- Includes a package for PL/TcL, but it is not enabled by default.

This lesson focuses primarily on query language and procedural language support.



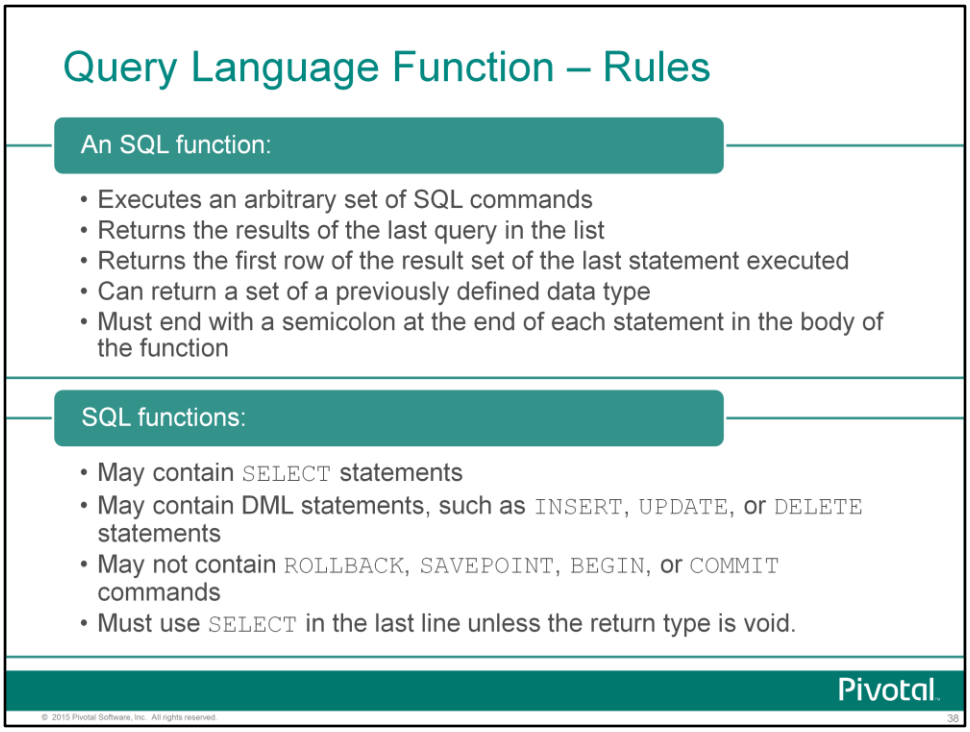

Before we discuss functions, let us examine the package management feature available starting with Greenplum Database 4.2.

The Greenplum Package Manager, `gppkg`, provides an interface for installing extensions, such as pl/R, pgcrypto, and PostGIS to name a few. You can install, update, or remove packages to and from the environment.

Greenplum provides the available packages from the EMC Download Center. It manages the installation and distribution of packages to all systems in the Greenplum cluster. In addition, it is automatically called to install the extensions during an upgrade or expansion of the Greenplum cluster. Some of these packages may require additional manual steps to make them available to the Greenplum Database. The list of supported extensions currently supported include:

- PostGIS
- PL/Java
- PL/R
- PL/Perl
- Pgcrypto

The Greenplum Package Management utility does not currently support user-custom modules.



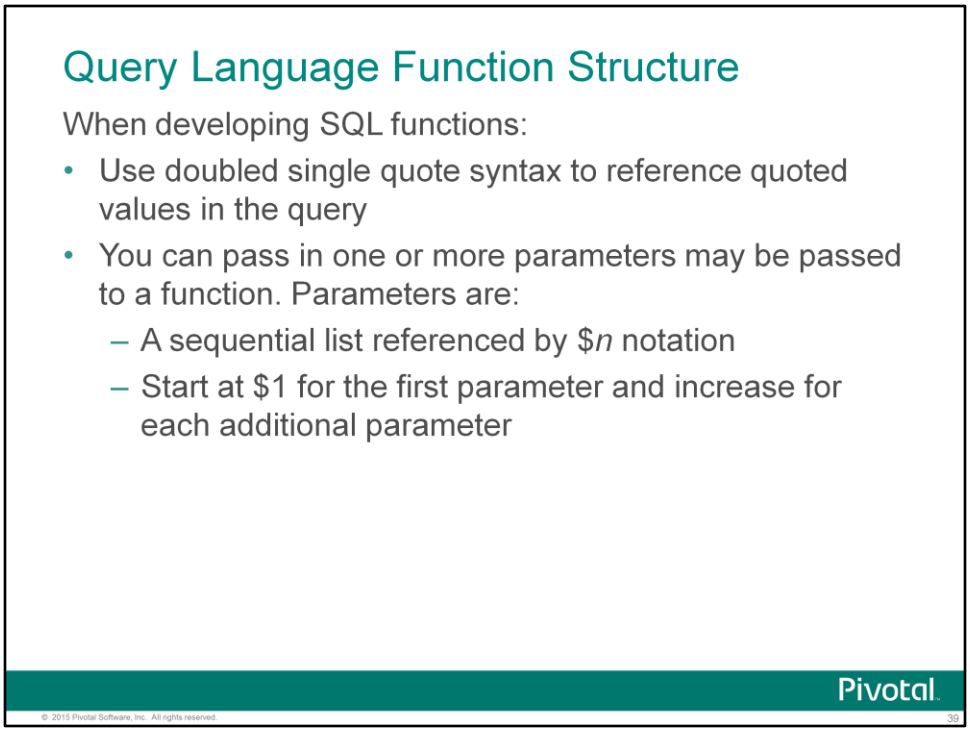

The query language function is a list of SQL statements that are wrapped in a function so that it is easier for you to call it when needed. An SQL function:

- Executes an arbitrary set of SQL statements within the body of the function.
- Returns the results of the last query in the list.
- Returns the first row of the result set of the last statement executed in the function body.
- Can return a set of previously defined data types.
- Must end with a semicolon at the end of each SQL statement.

You:

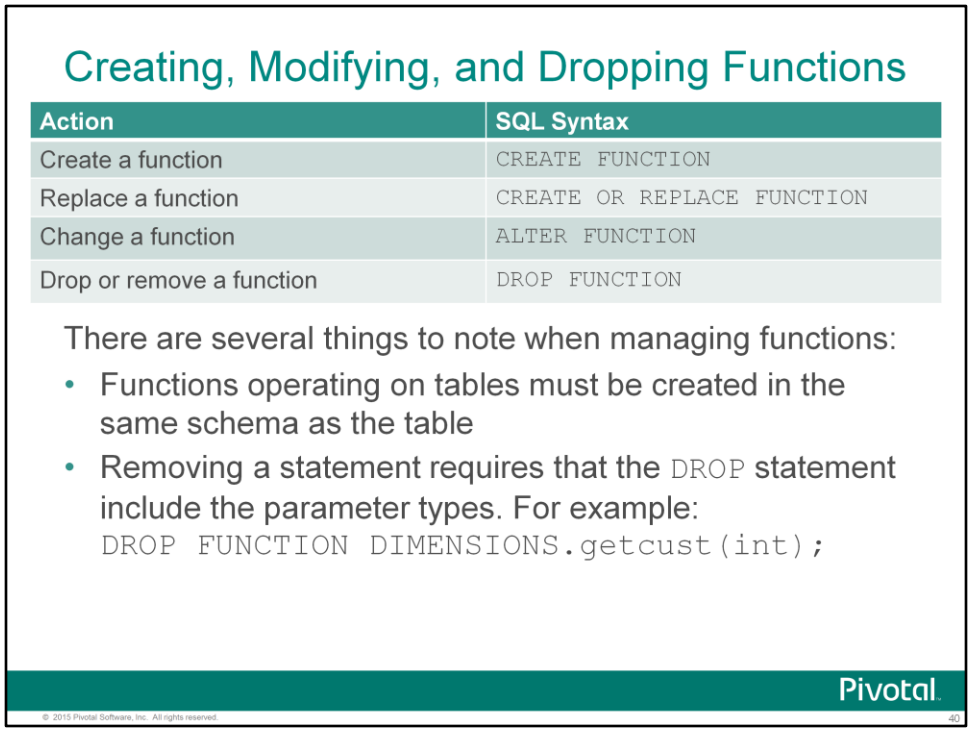
- Can include `SELECT` statements.
- Cannot include rollback, savepoint, begin, or commit commands.
- Support these end-user functions yourself.

One of the main strengths of a function is to return a value. When defining the function, the last statement must select something. It does not return a void by default.



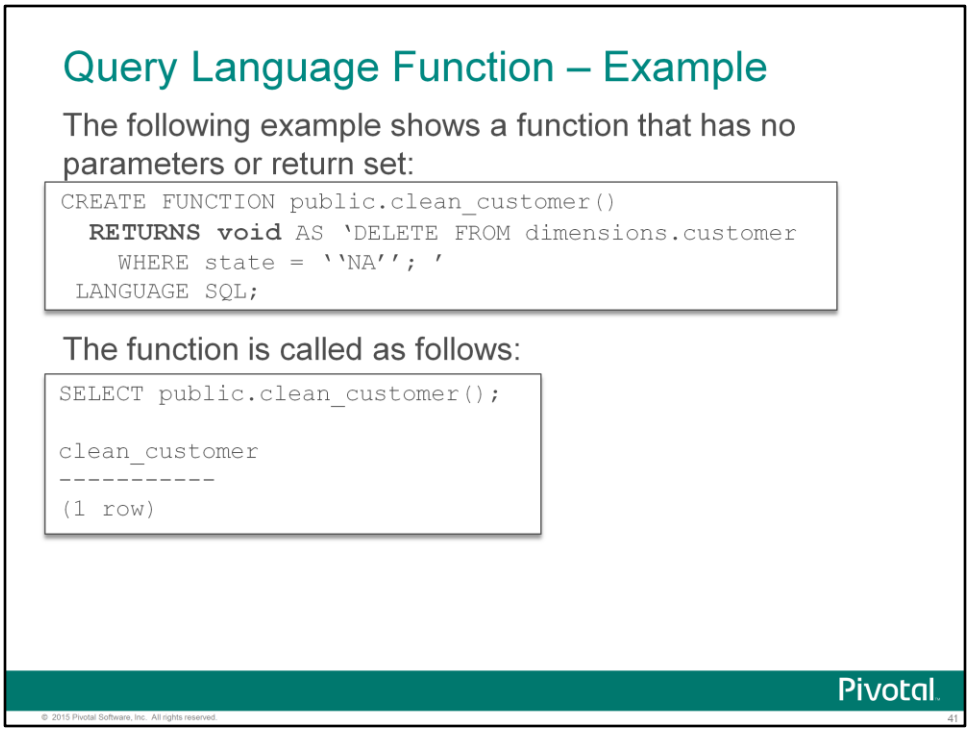

The query language requires that quoted values use the doubled single-quote syntax, where two single quotes are used.

If you are passing parameters into a function, the parameters will be referenced by $n, where n is a sequential list of numbers, starting with 1. The first parameter is referenced as $1, while the second is referenced as $2. Each additional parameter is assigned the next sequential number in the series.



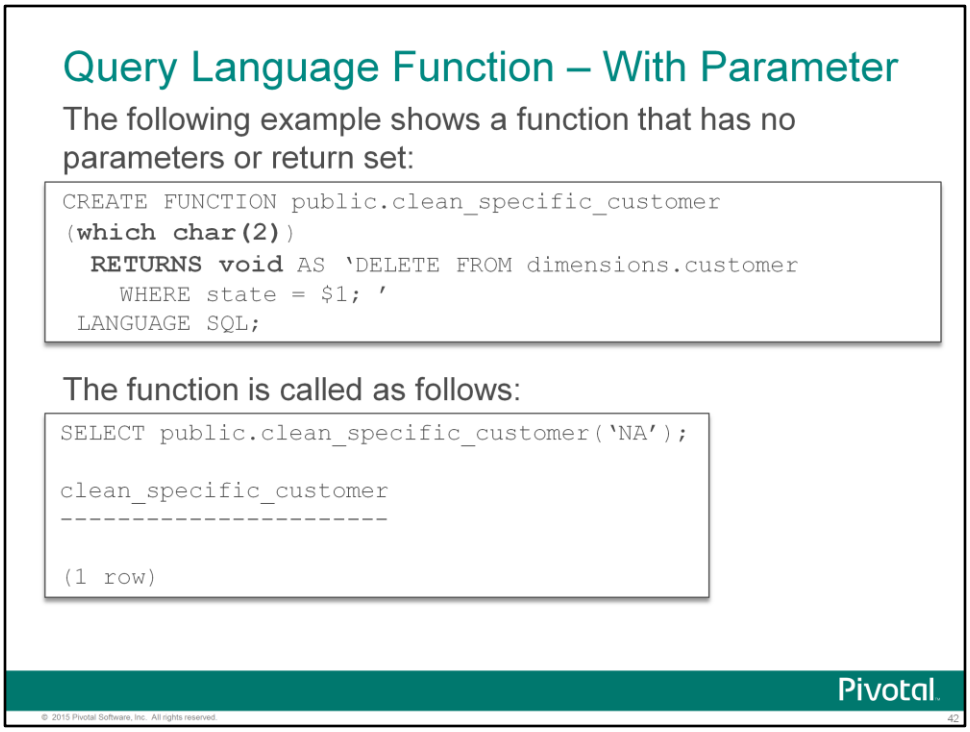

Functions that operate on tables must be created in the same schema. If you modify a table, you must have access to a schema. You:

- Create a function with the `CREATE FUNCTION` command. You must have `CREATE` access to the schema to create a function. A function can be created with or without parameters.
- Replace an existing function with the `CREATE OR REPLACE FUNCTION` command. This command either creates a function if one did not exist before, or replaces an existing function. If you are replacing an existing function, you must specify the same number of parameters and the same data types found in the original function. If not, you are actually creating a new function.
- Change a function with the `ALTER FUNCTION` command. You must own the function before you can modify it. If the function is to be created in another schema, you must have `CREATE` privilege on that schema.
- Drop or remove a function with the `DROP FUNCTION` command. Because you can have multiple functions with the same name but different number of parameters and/or parameter types, you must include the appropriate number of parameters and parameter types as part of the command. You must also be the owner of the function to remove the function from the schema.

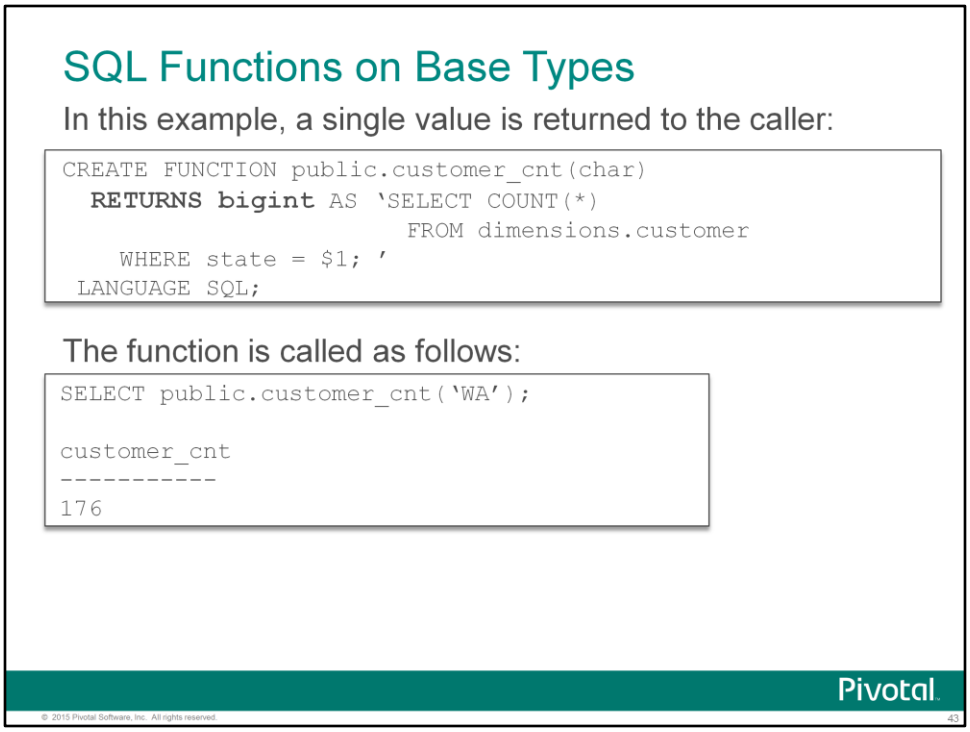



In this example, a function is created to remove invalid customers from the `dimensions.customer` table.

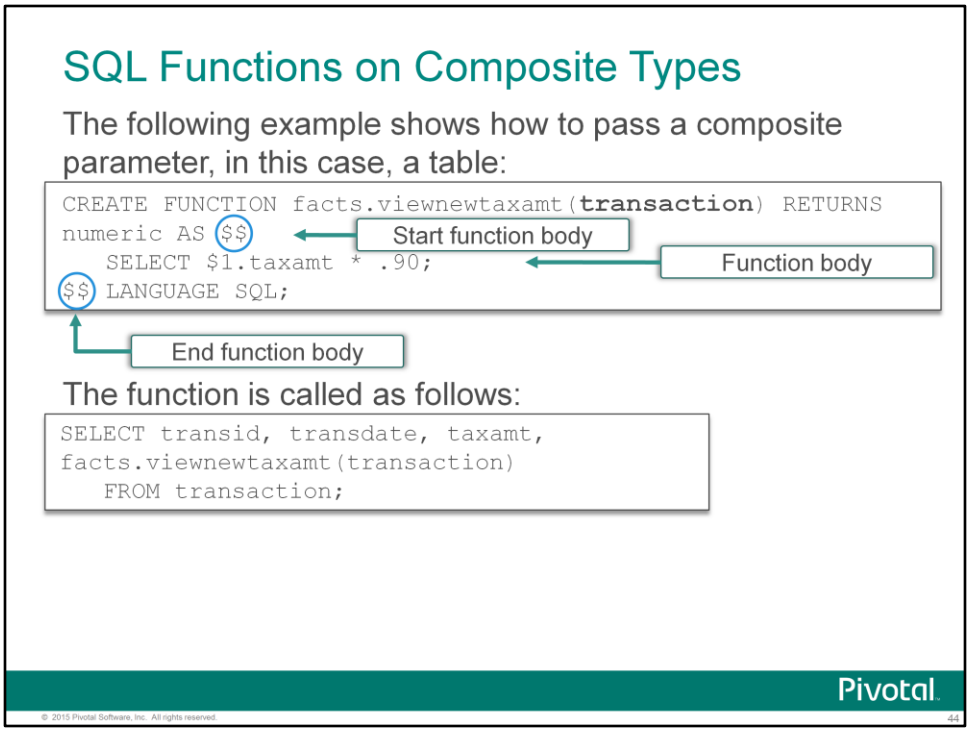
To use the function, issue a SELECT statement against the function. The function runs from the master, which evaluates and pushes all of the SQL in the statement down to the segments. The result is returned to the master.




In this example, a parameter is passed in. You can pass in a character as the data type. You do not need to specify a name for the parameter. The name `which` does not need to be in the code. The function definition will accept the data type, in this case, `char(2)`, without it being named, since it is not used anywhere else in the function.



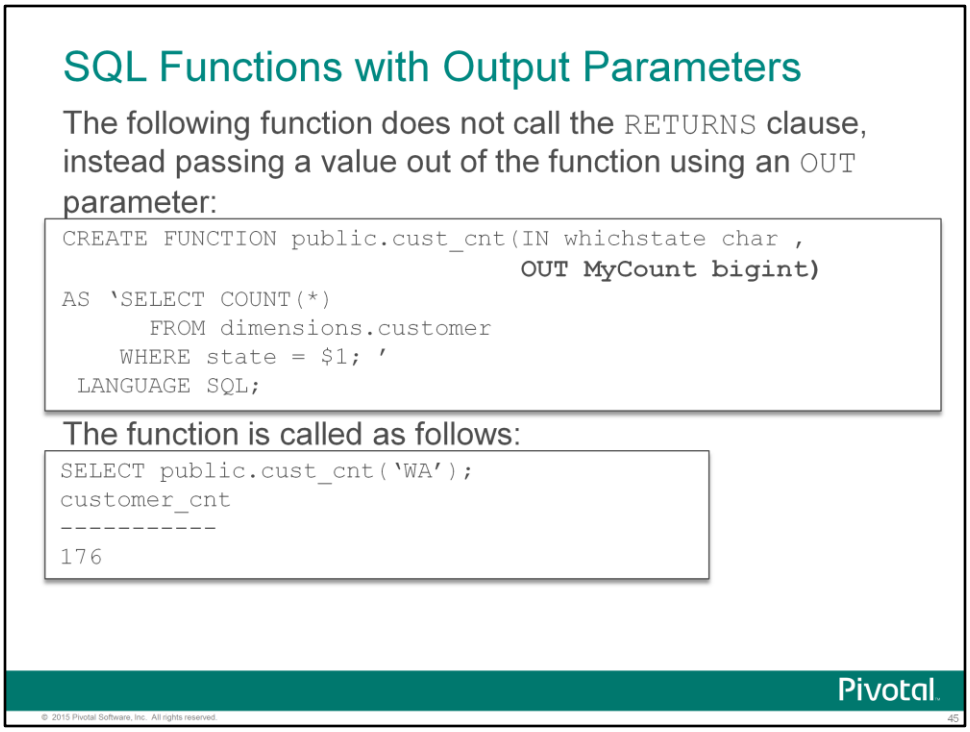

In this example, the char data type is being passed into the function, `public.customer_cnt`. Since it performing an aggregation with `COUNT(*)`, it evaluates on the master. The `FROM` predicate gets evaluated on the segments.




Whenever you pass in a parameter, you can identify it as:

- A base or primitive type, such as `integer`, `char`, or `varchar`.
- A composite parameter which is a relation-based parameter.
- A domain parameter.

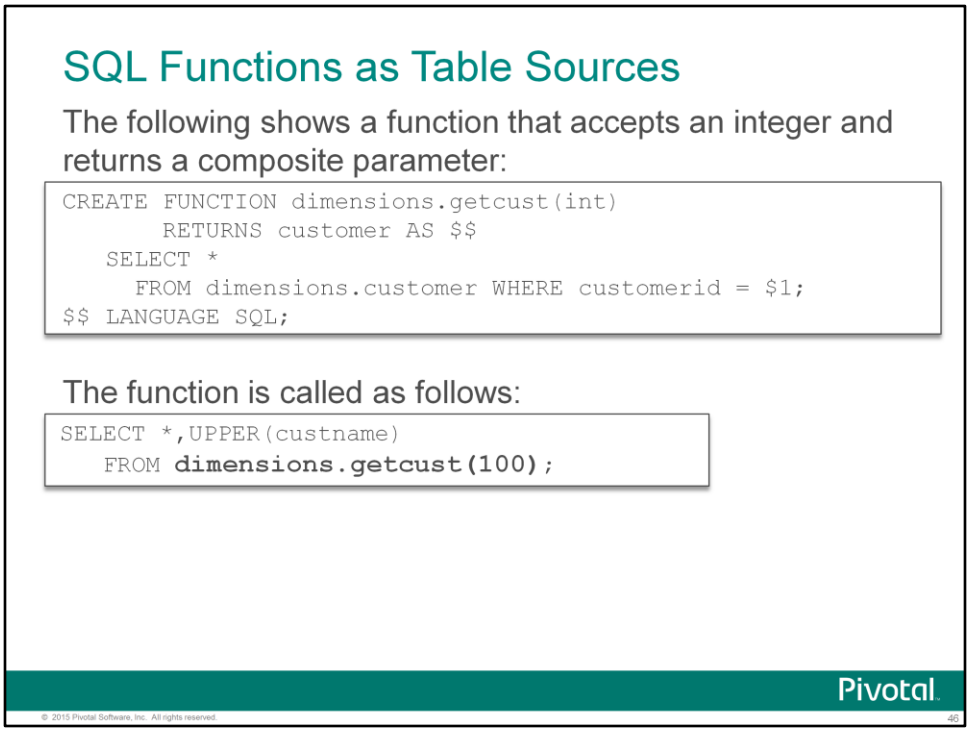
In this example, you pass the parameter `transaction`, which is relation-based. When you call this function, you select the various columns in the table. The value is multiplied by .9 and returned as numeric.




This example does not have a return clause. Within the parameter block, you define variables as:

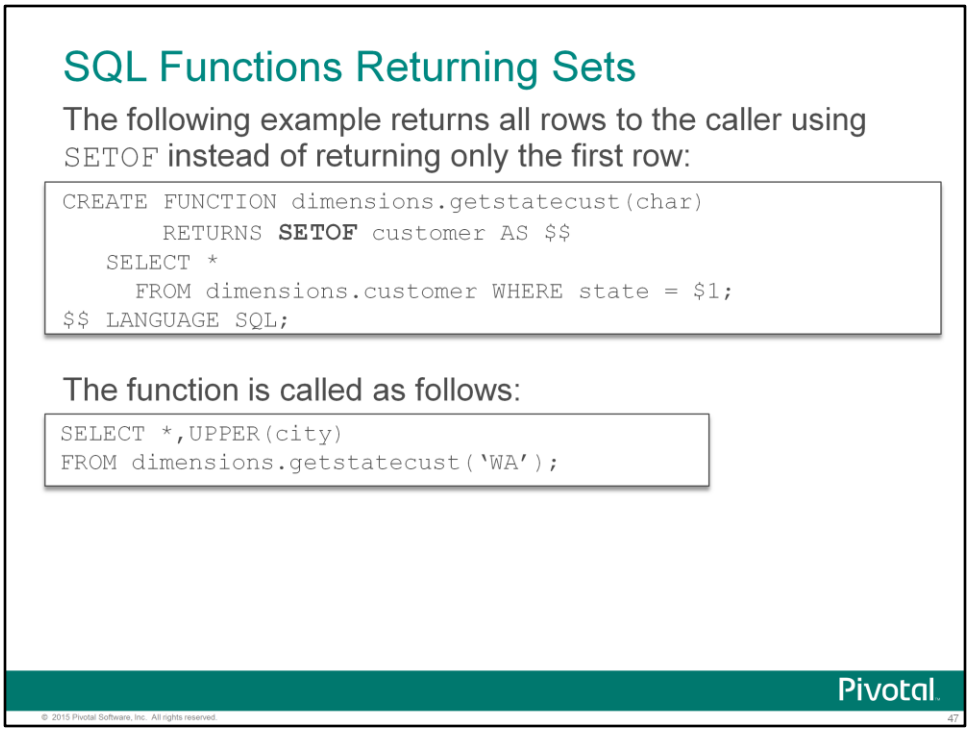
- `IN` to show which variables are being passed into the function.
- `OUT` to specify the variables that are passed out of the function. This is a shortcut for not having a return clause.

Specifically using the `RETURNS` clause or defining `IN` and `OUT` parameters is a matter of your coding preference and style.



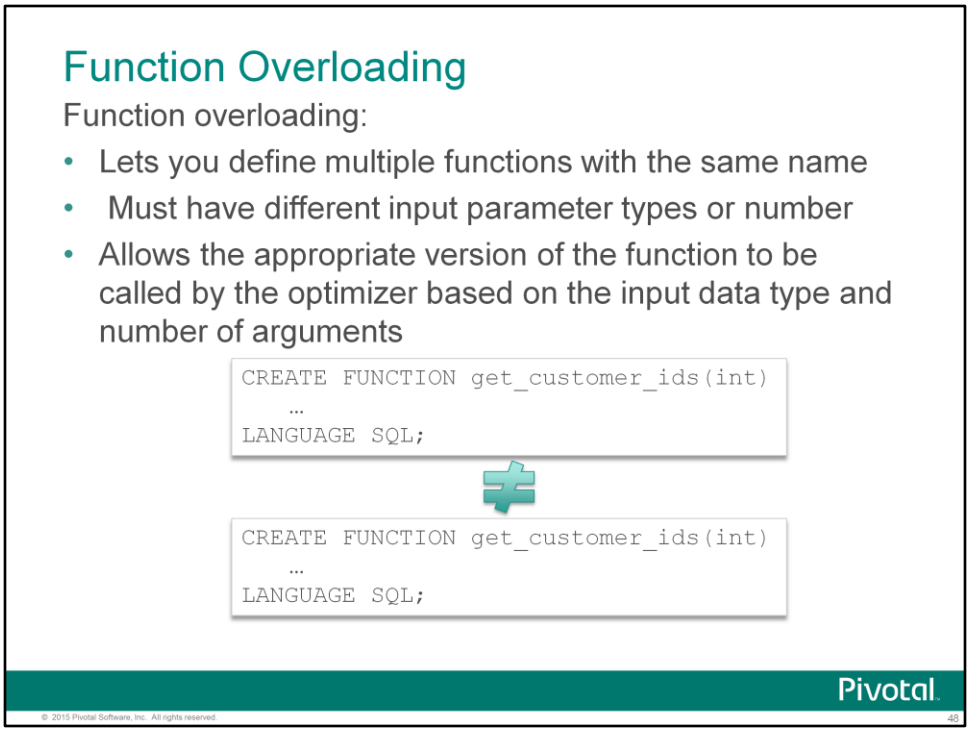

A function can be used as a table source. This lets you apply a filter or security check, wrapped around a table. Whenever a user queries the table, the results are returned in the correct format.

In this example, the `dimensions.getcust(int)` function returns a result set where the customer is equal to customer ID. This makes it much easier for a user to issue a call to the table without needing to specify the syntax provided in the function.

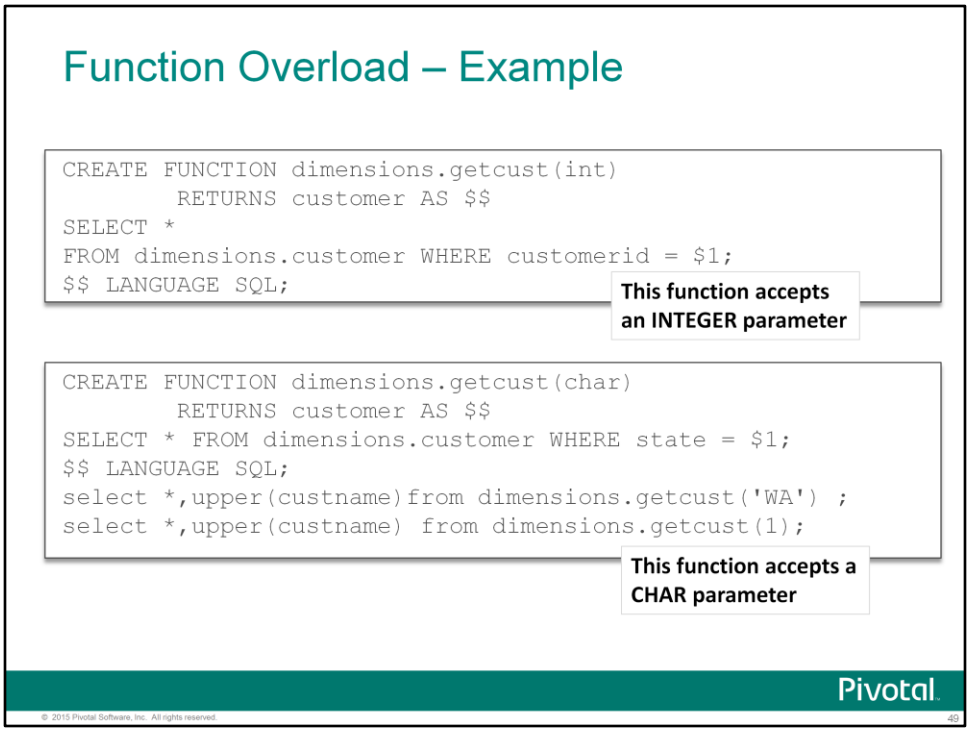



When an SQL function is declared as returning `SETOF` *`data_type`*, the final `SELECT` query in the function is executed to completion. Each row it outputs is returned as an element of the result set. If you do not specify a return as a `SETOF`, only the first row is returned.

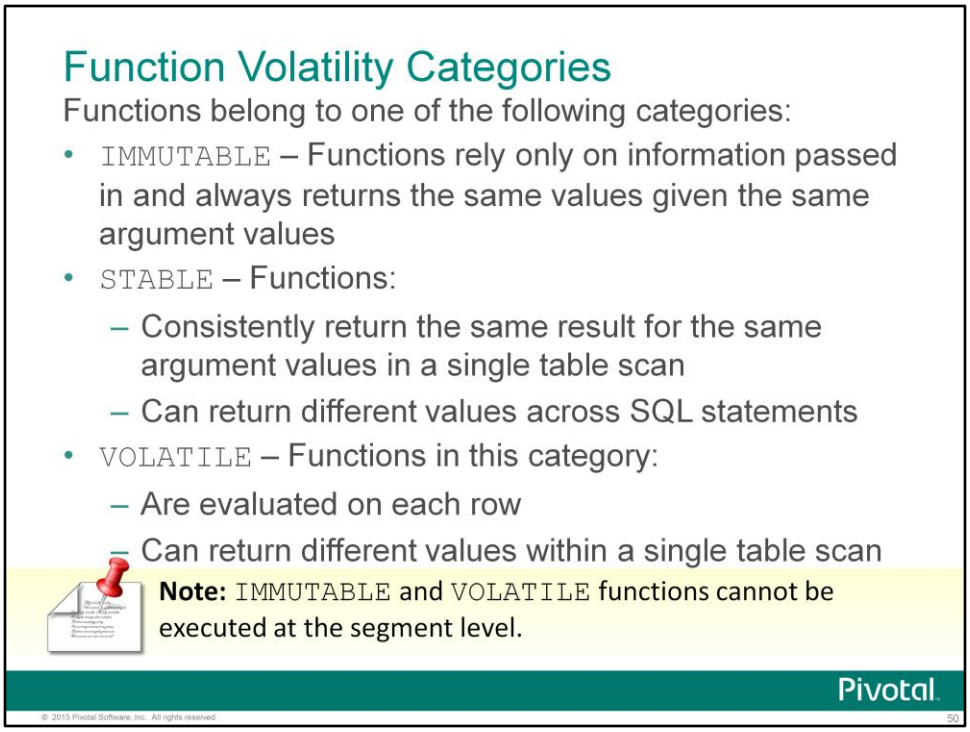
Now that you have an understanding of the structure of a function, let us examine function overloading.




Function overloading allows you to build several functions with the same name, but with different types or number of parameters. For example, creating a function called `get_customer_ids(int)` is different from the function called `get_customer_ids (char)`. If you pass in a character value to the function, the second function is called. Two functions are considered the same if they have the same names and input argument types.

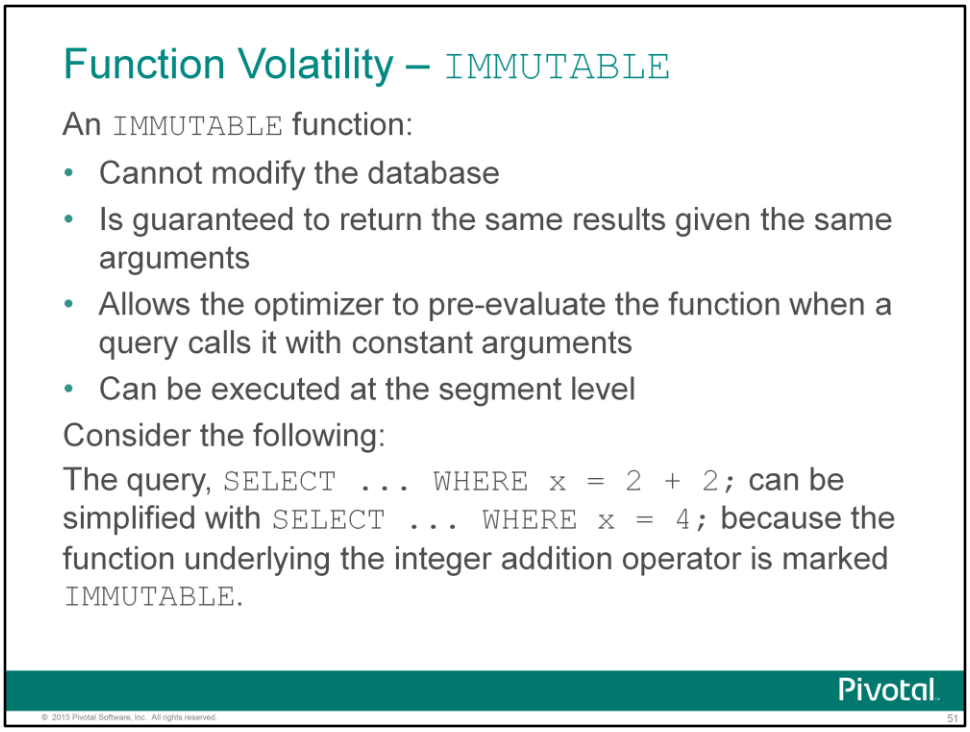



In this example, you have two functions defined with the same name. The first passes in an integer whereas the second passes in a character value. These functions have two different behaviors, but the same name. The optimizer will decide which function to call based on the parameters you pass in.




The volatility category gives the optimizer a general idea of what the function will do and whether it falls into one of the following areas:

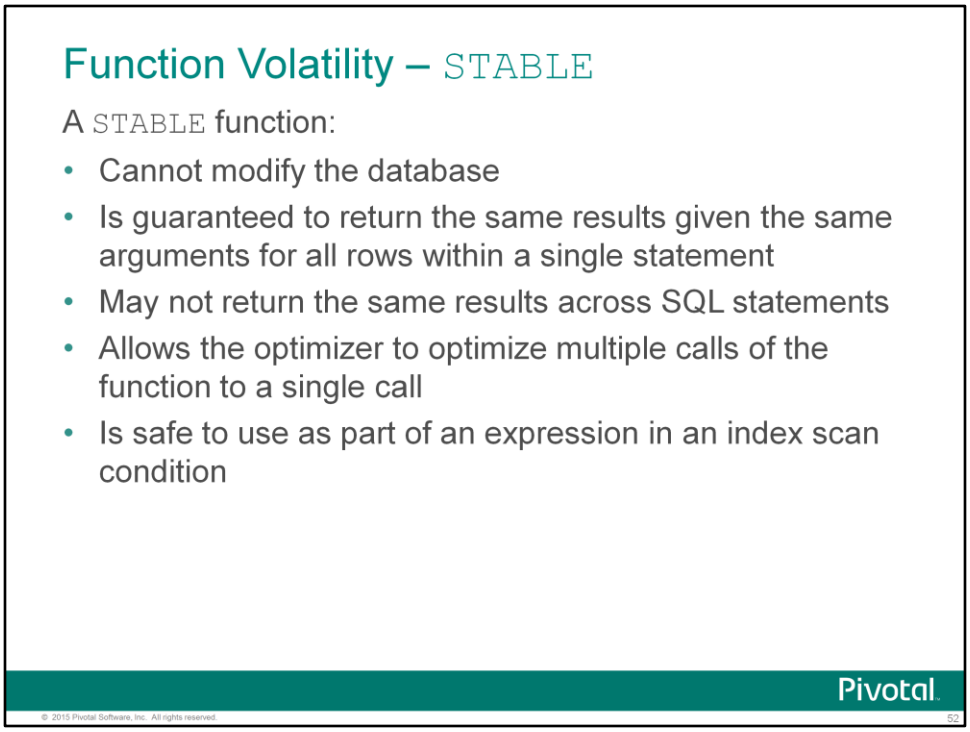
- **`IMMUTABLE`** – In an immutable function, the value is not going to change. This function relies on information provided in the argument list and will always return the same values for the same argument values.
- **`STABLE`** – Each table scan means that the function is evaluated once and uses the updated value. Another table scan will yield different results. Functions whose parameters depend on database lookups or parameter variables are classified as `STABLE`. Functions in the `current_timestamp` family qualify as stable.
- **VOLATILE** – The function is evaluated on each row. In this case, the function value can change within a single table scan. Examples of volatile functions include `random()`, `currval()`, and `timeofday()`.




**IMMUTABLE** functions, considered the most stable of functions, cannot modify the database. All of your operators are considered immutable.

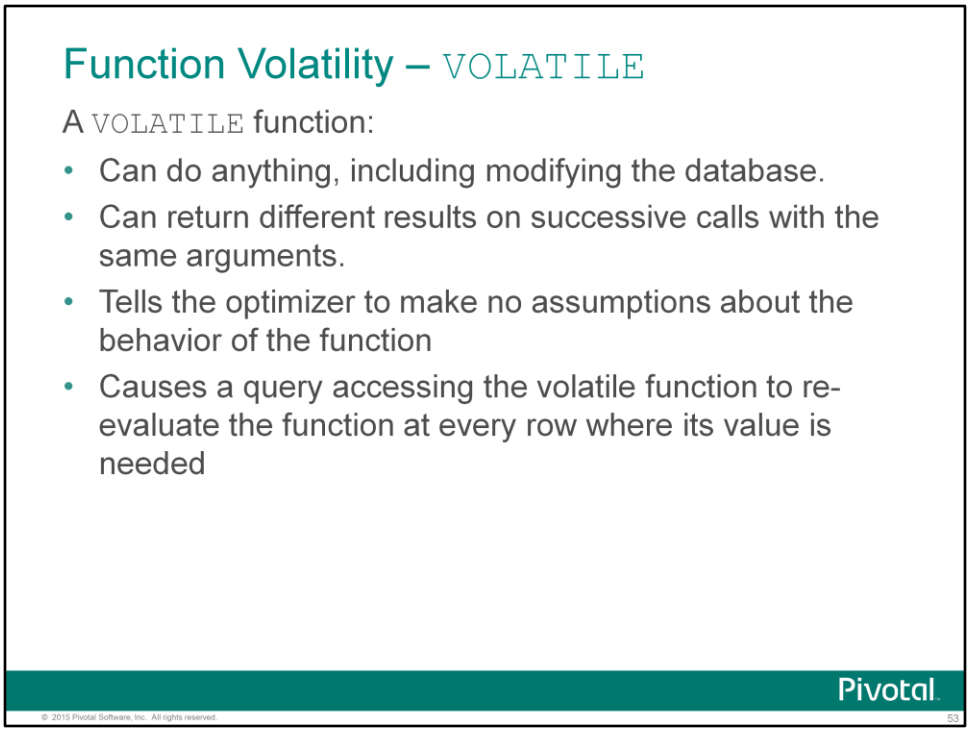
**IMMUTABLE** functions are guaranteed to return the same result given the same set of arguments every time. This allows the optimizer to pre-evaluate the function when a query calls it with the same arguments.

This type of function can be run at the segment level because the system can guarantee its behavior.




As with `IMMUTABLE` functions, `STABLE` functions cannot modify the database. Their use is restricted and so cannot be used at the segment level.

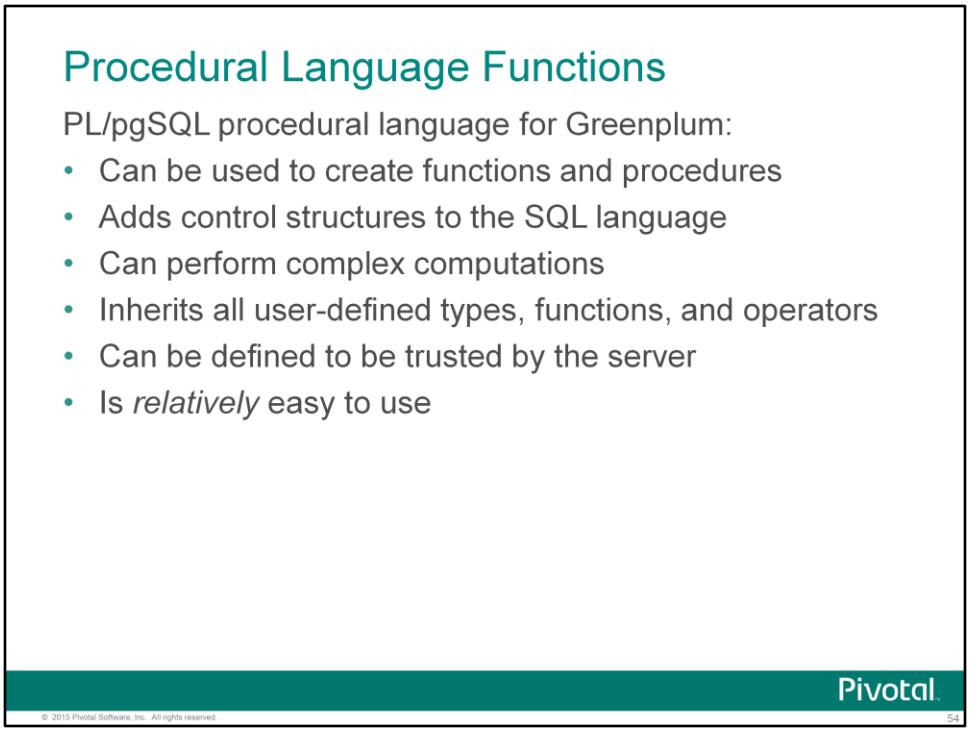
A function defined as `STABLE` will return the same results given the same arguments in a single table scan. However, the results may differ across multiple SQL statements. This does allow the optimizer to optimize multiple calls of the function in a single call.




The use of `VOLATILE` functions is restricted in Greenplum.

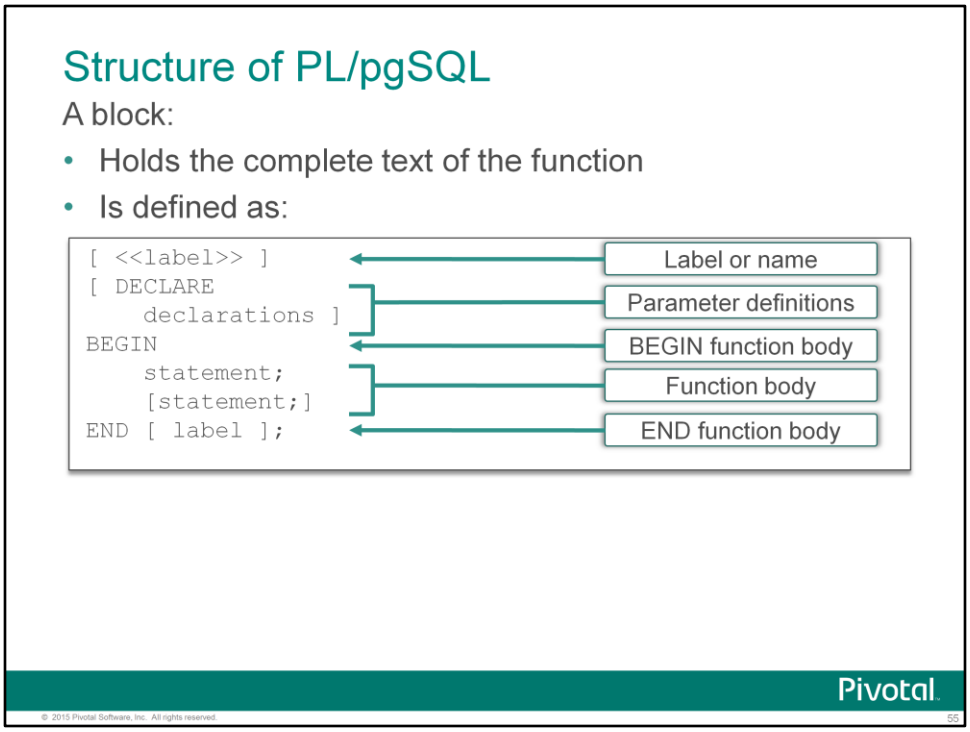
A `VOLATILE` function is capable of doing anything, including modifying the database. The value of the function changes on each successive call even with the same arguments in a single table scan. The optimizer cannot make any assumptions about the behavior of the function.

Now that you have examined building functions in SQL, we will examine building functions in the PL/pgSQL procedural language.



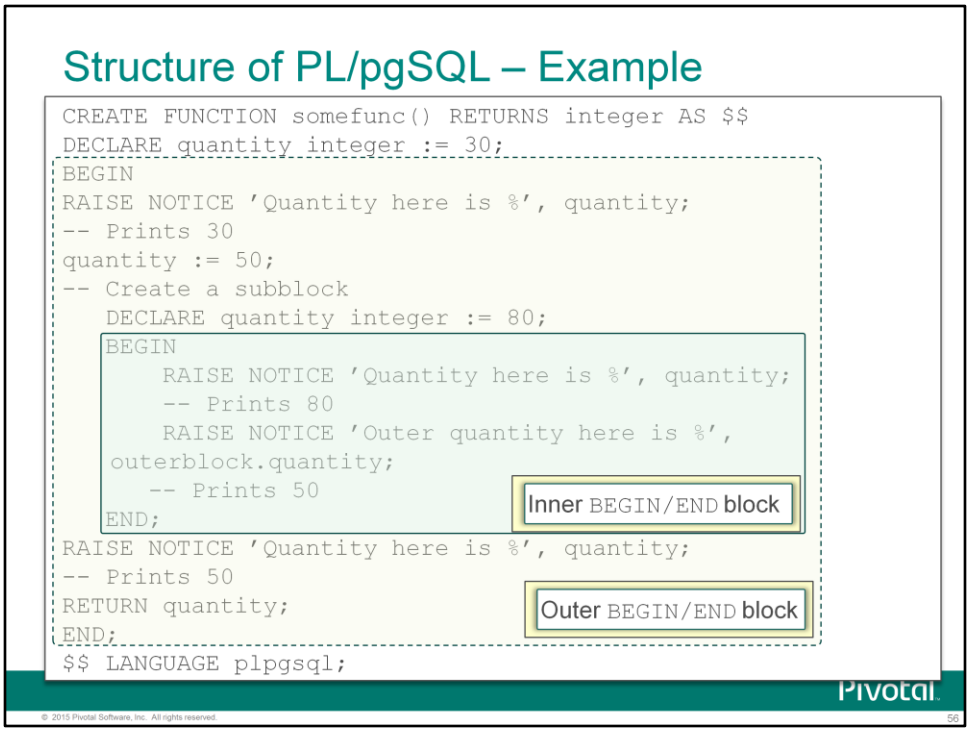

Procedural language support for Greenplum is in PL/pgSQL. This language is registered in Greenplum on installation and is available to all databases. It allows support for flow control, structures, and returns. This is an enhancement over programming with SQL statements allow where it:

- Adds control structures to the SQL language.
- Can perform complex computations.
- Inherits all user-defined types, functions, and operators.
- Is relatively easy to use.




A function defined in PL/pgSQL must be defined within a block. The structure of the block:

- May contain a label.
- Must declare a parameter for each value passed in.
- Starts and ends with `BEGIN` and `END` respectively to define the boundaries of the block.
- Contains a number of statements within the `BEGIN/END` block. A block may also contain inner or subblocks, where you define and user local variables not visible to the outer block.

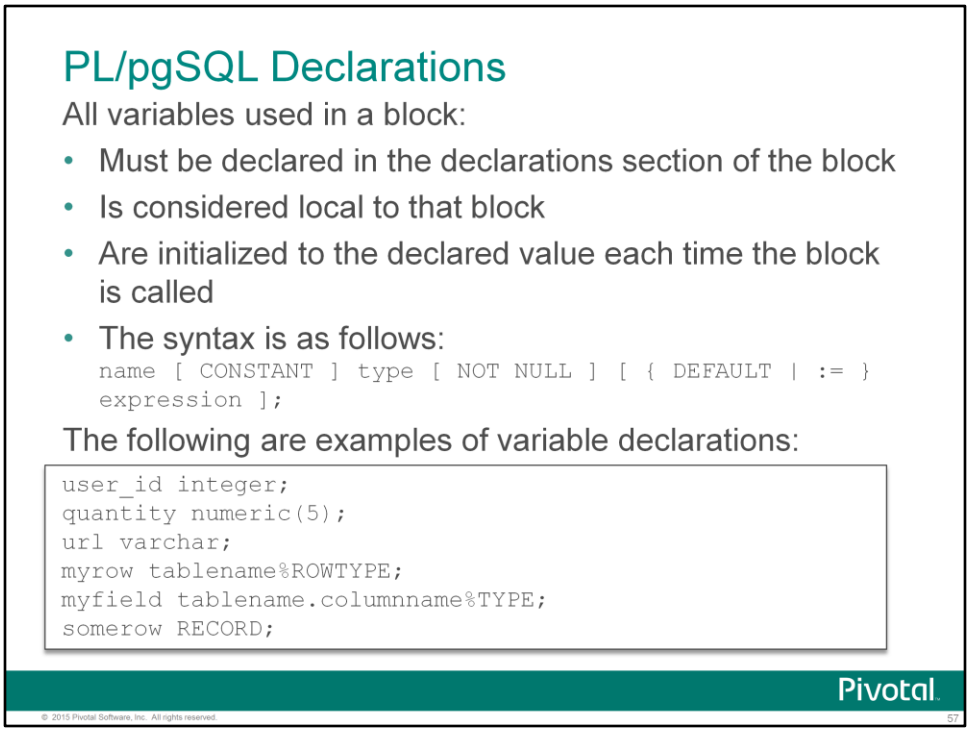



This examples shows the structure of a PL/pgSQL function, where within the innermost `BEGIN/END` block, you:

- Print the value of quantity as it was declared right before the innermost `BEGIN/END` block.
- Print the value of the quantity variable as defined in the outer block.

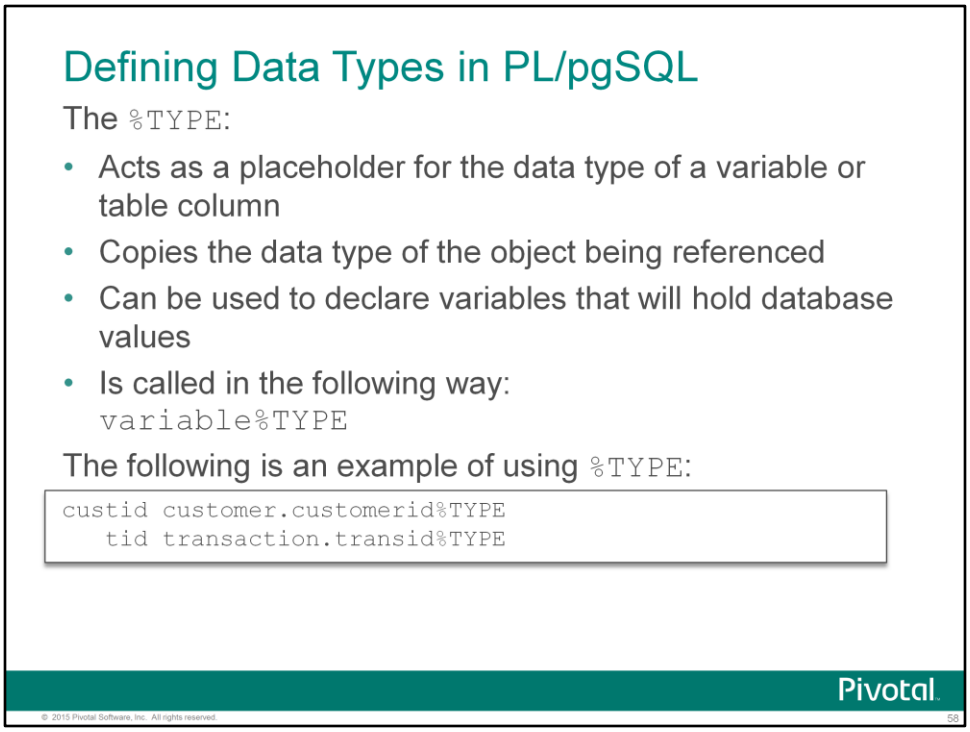
The value of quantity:

- Was initially declared as 30 for the first `BEGIN/END` block.
- Updated directly to 50.
- Declared as 80 for the inner block.
- Is still 50 after the inner block has completed execution.



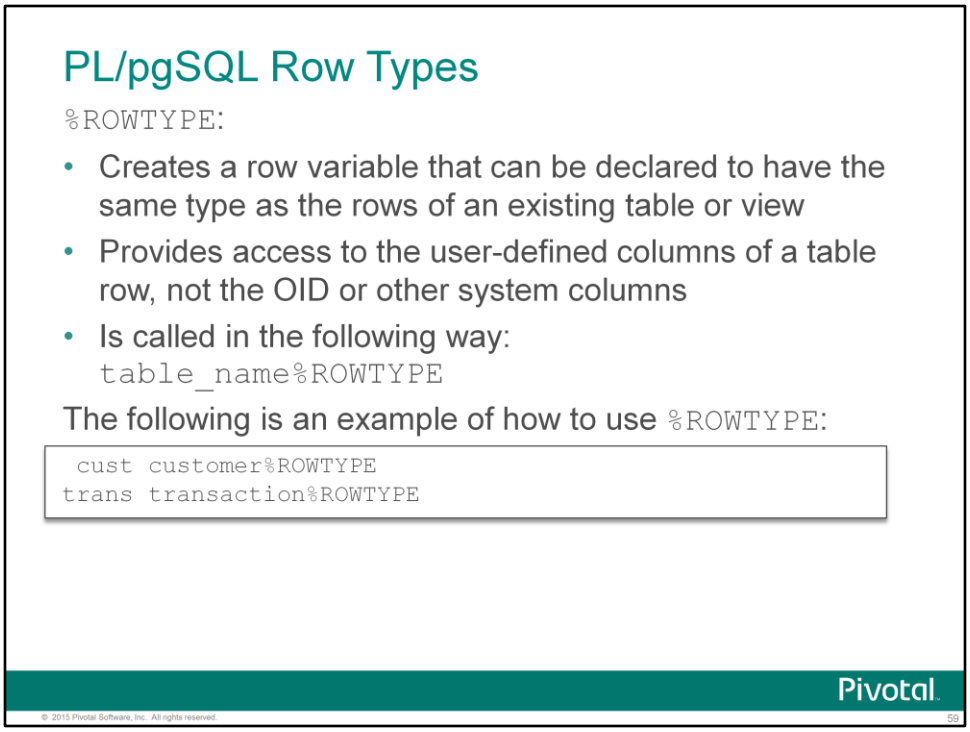

When you declare a block, you must define the name and data type for the block. Everything else is optional.

The variable is considered local to that block, meaning an outer variable is not aware of the definition.

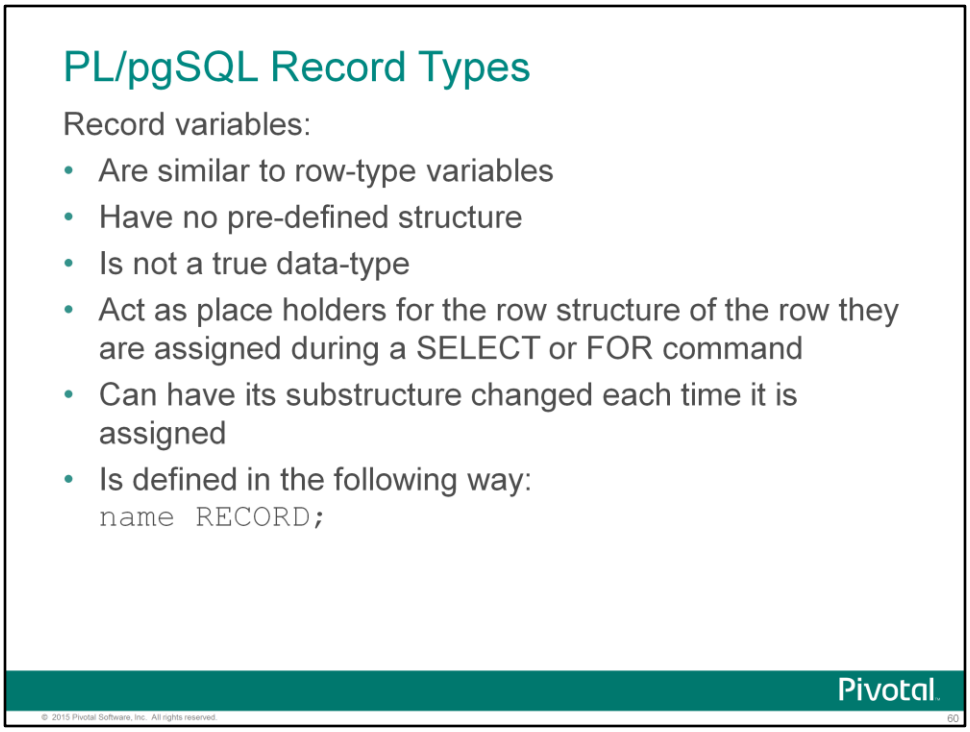



By using `%TYPE` as a place holder for the data type, you do not need to know the data type of the structure you are referencing. Most importantly, if the data type of the referenced item changes in the future, such as if you change the type of `customerid` from `integer` to `real`, you might not need to change your function definition.

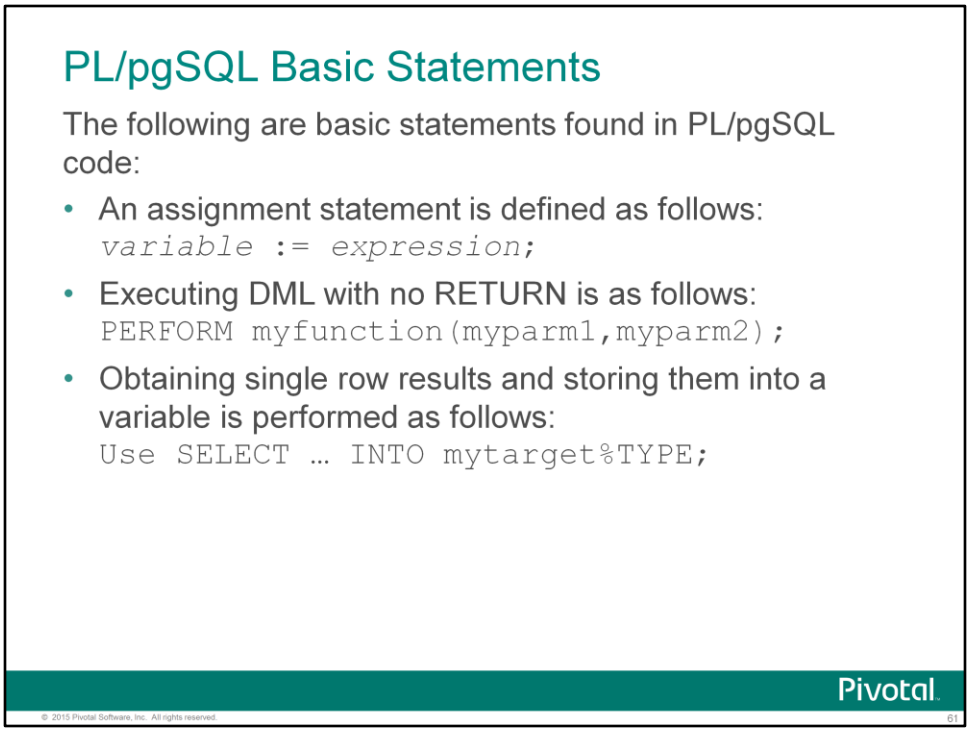
`%TYPE` is particularly valuable in polymorphic functions, as the data types needed for internal variables can change from one call to the next. Appropriate variables can be created by applying `%TYPE` to the function arguments or result placeholders.




A row type is a variable of a composite type. You can define a row type for rows of a specific table you are working with. This lets you hold an entire row for a SELECT statement, as long as the columns of the query match the declared type of the variable.




The record structure changes based on what is assigned to it. This is ideal if you do not know the row structure you are working with. In this sense, it acts as a place holder for the variable.

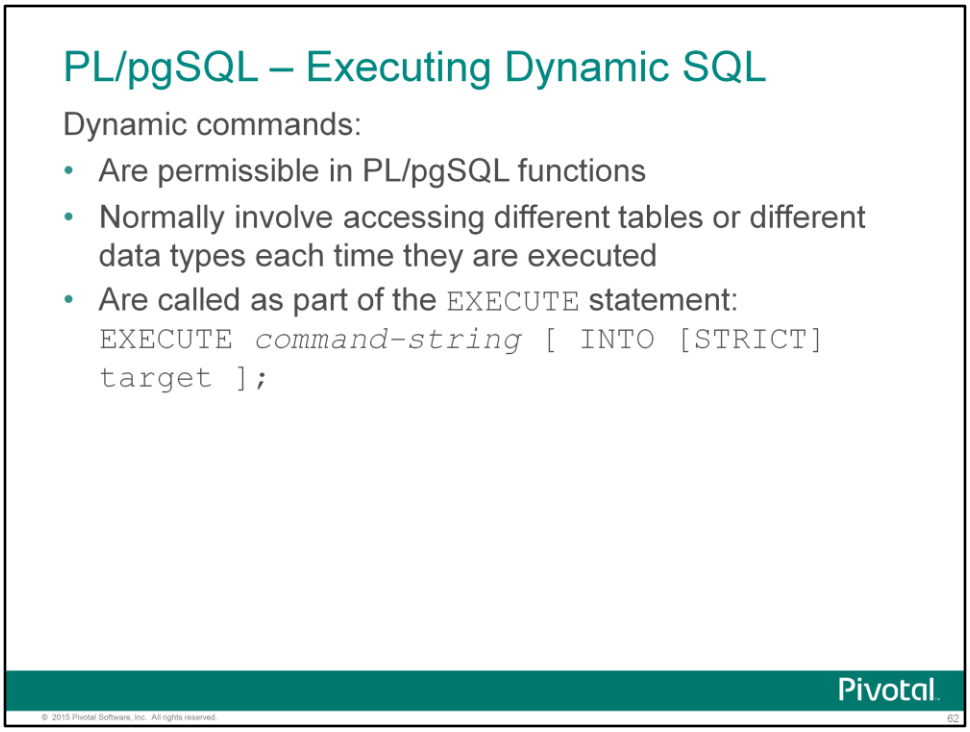



The result of a SQL command yielding a single row, possibly of multiple columns, can be assigned to a record variable, row-type variable, or list of scalar variables. This is done by writing the base SQL command and adding an `INTO` clause.

If a row or a variable list is used as target, the result columns of the query must exactly match the structure of the target with the appropriate number and data types of the column, or a run-time error occurs.

When a record variable is the target, it automatically configures itself to the row type of the query result columns.

The `INTO` clause can appear almost anywhere in the SQL command. Customarily, it is written either just before or just after the list of select expressions in a `SELECT` command, or at the end of the command for other command types.

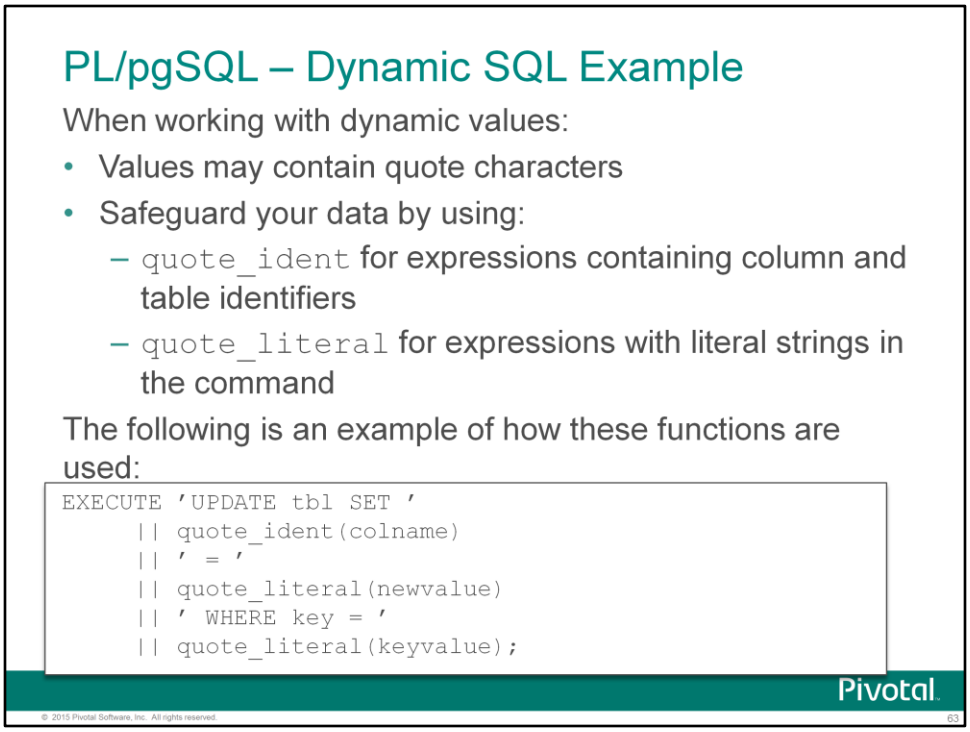



Substitution of PL/pgSQL variables is not performed on the computed command string. Any required variable values must be inserted in the command string as it is constructed. Instead, the command is prepared each time the statement is run. The command string can therefore be dynamically created within the function to perform actions on different tables and columns.

The `INTO` clause has the following conditions:

- It specifies where the results of a SQL command returning rows should be assigned.
- If a row or variable list is provided, it must exactly match the structure of the query result set.
- When a record variable is used, it will configure itself to match the result structure automatically.
- If multiple rows are returned, only the first will be assigned to the `INTO` variable.
- If no rows are returned, `NULL` is assigned to the `INTO` variable(s).
- If no `INTO` clause is specified, the query results are discarded.

When working with dynamic commands you will often have to handle escaping of single quotes. The recommended method for quoting fixed text in your function body is dollar quoting.

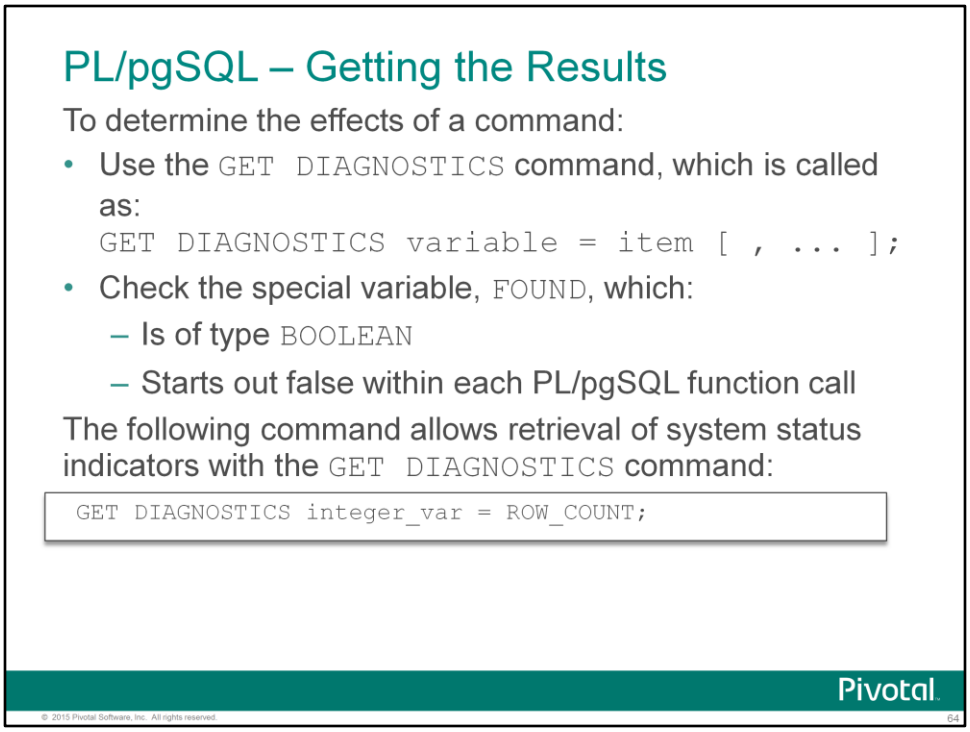



The example shown on the slide demonstrates the use of the `quote_ident` and `quote_literal` functions.

Dynamic values that are inserted into the constructed query require special handling since they may also contain quote characters. When working with dynamic values:

- Expressions containing column and table identifiers should be passed to `quote_ident`.
- Expressions containing values that should be literal strings in the constructed command should be passed to `quote_literal`.

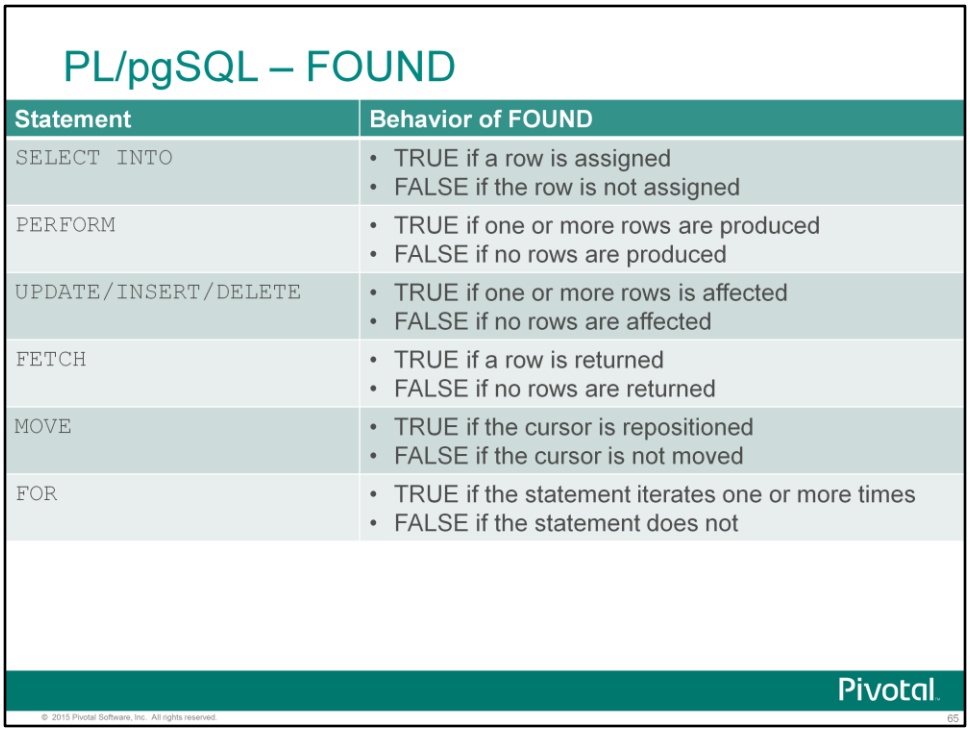
Both take the appropriate steps to return the input text enclosed in double or single quotes respectively, with any embedded special characters properly escaped.




There are several variables you can access from the system when performing a command. The `GET DIAGNOSTICS` command is used to obtain system status indicators after executing a command.

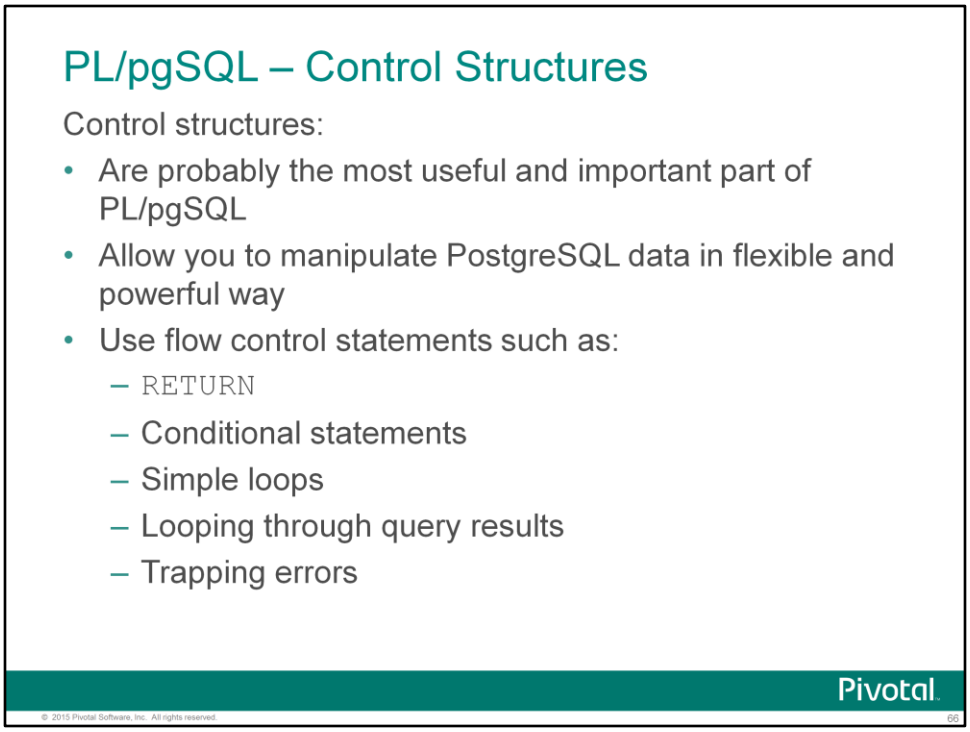
The `FOUND` variable, which is of type `BOOLEAN`, starts out as false for each block, but adjusts depending on the results of the calls made.

The full listing of variables you can retrieve are available on the PostgreSQL website.



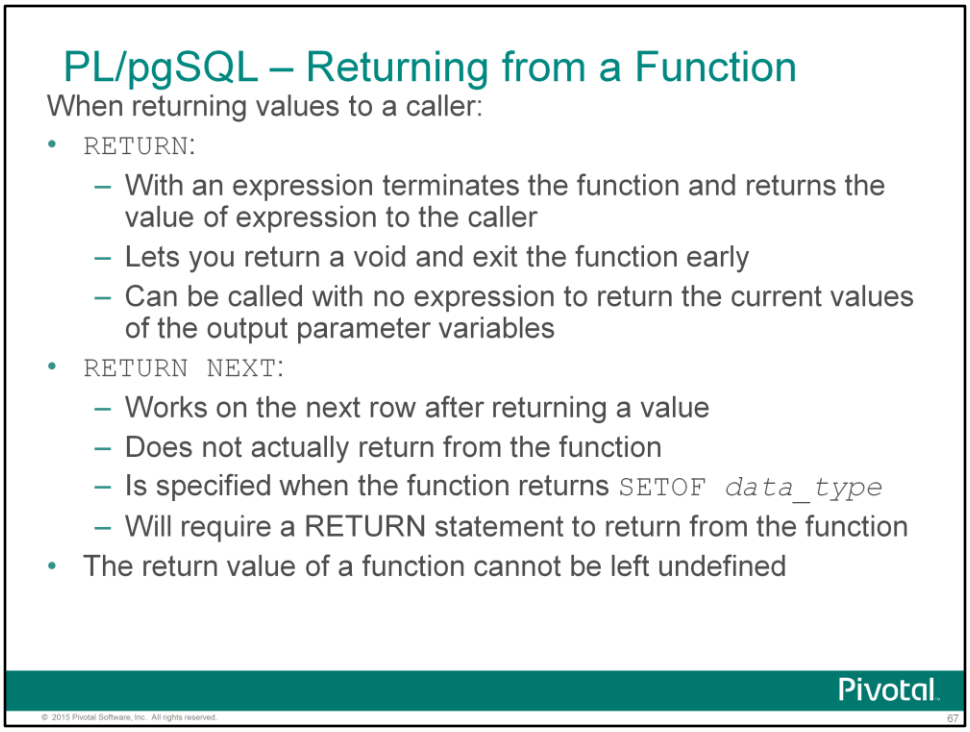

`FOUND` is a local variable within each PL/pgSQL function. Any changes to the variable affect only the current function. The variable is set by a different statements, including:

- `SELECT INTO`
- `PERFORM`
- `UPDATE`
- `INSERT`
- `DELETE`
- `FETCH`
- `MOVE`
- `FOR`




Control structures allows you to manipulate your data with flow control statements, such as:

- `RETURN` statements
- Conditional statements that include `IF/THEN/ELSE` blocks
- Simple loops, such as `LOOP`, `CONTINUE`, and `WHILE`
- Looping through query results `FOR…IN…LOOP` statements
- Trapping errors




`RETURN` and `RETURN NEXT` are two flow control commands that allow you to return data from a function.

`RETURN`:

- Returns a value and terminates the function.
- Lets you return a void and exit the function early if you declared the function to return a void. Do not write a statement after the RETURN statement however as it does not get evaluated.
- Can be called with no expression when the function defines output parameter variables. The function will still return the values of the output parameter variables.

`RETURN NEXT`:

- Works on the next row after returning a value.
- Does not return from the function, instead saving the value of the expression on that iteration.
- Is used when the function returns `SETOF` *data_type*.
- Should be called with a final `RETURN` statement at the end with no arguments.

All functions must have a return type. If you do not have a return type, specify `VOID`.

## PL/pgSQL – Returning from a Function – Example

The following example implements `RETURN NEXT` and then a final `RETURN` after iterating through the rows:

```
CREATE OR REPLACE FUNCTION getAllStores()
RETURNS SETOF store AS
$BODY$
DECLARE r store%rowtype;
BEGIN
FOR r IN SELECT * FROM store WHERE storeid > 0 LOOP
   -- can do some processing here
   RETURN NEXT r;
  -- return current row of SELECT
END LOOP;
RETURN;
END
$BODY$
LANGUAGE plpgsql;
```

`RETURN NEXT` statement

Final `RETURN` statement

Pivotal





The current implementation of `RETURN NEXT` stores the entire result set before returning from the function. This means that if a PL/pgSQL function produces a very large result set, performance might be poor.

Data will be written to disk to avoid memory exhaustion, but the function itself will not return until the entire result set has been generated.

Currently, the point at which data begins being written to disk is controlled by the `work_mem` configuration variable. Administrators who have sufficient memory to store larger result sets in memory should consider increasing this parameter.

## PL/pgSQL – Conditionals

The IF/THEN/ELSE block is defined as follows:

```
IF boolean-expression THEN
   statements
[ ELSIF boolean-expression THEN
   statements ]
[ ELSIF boolean-expression THEN
   statements ]
[ ELSE IF boolean-expression THEN
   statements ]
[ ELSE statements ]
END IF;
```

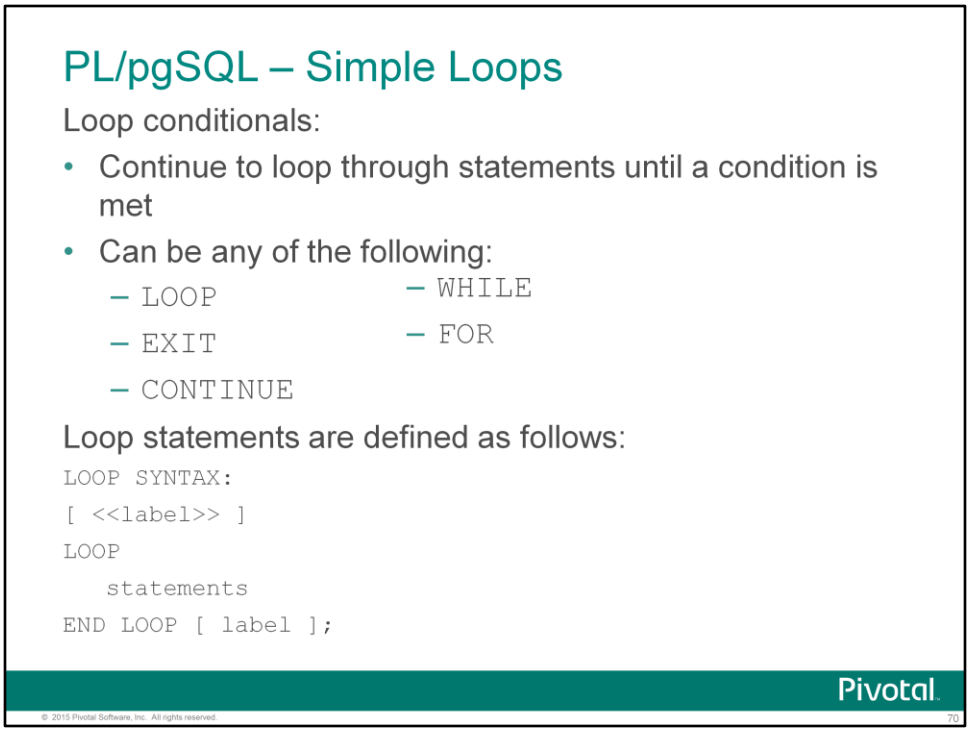
**Note:** Put the most commonly occurring condition first!

Pivotal



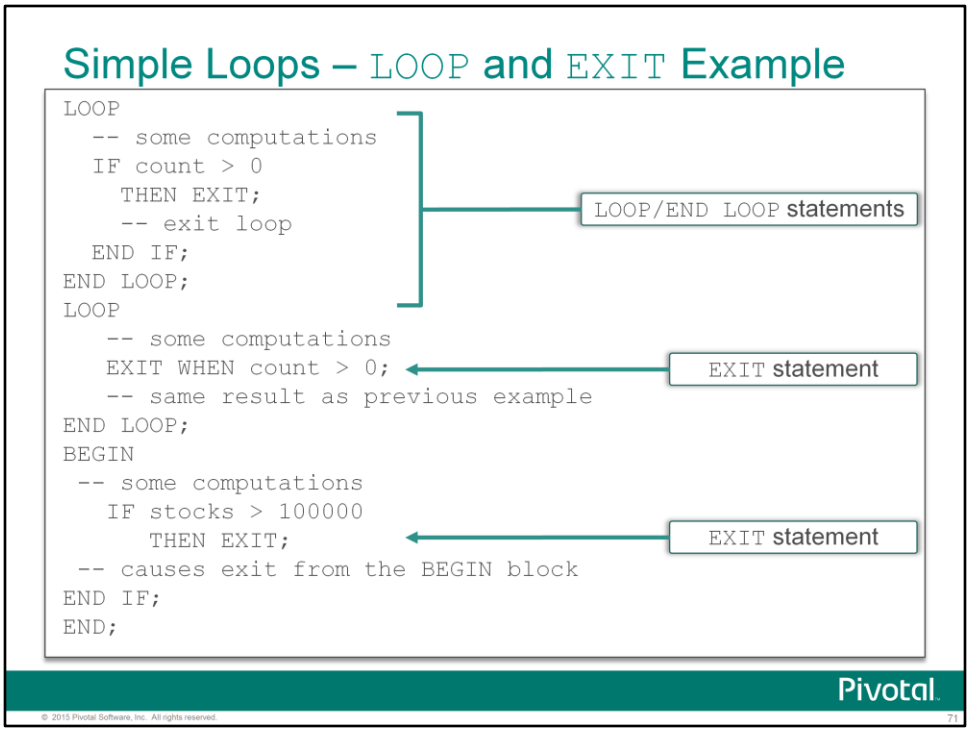


The `IF/THEN/ELSE` conditional is used in flow control to determine which statements are called based on evaluated expressions. Use the most commonly occurring condition first when implementing flow control with the `IF/THEN/ELSE` block.




Loop conditionals will continue to loop through statements until some condition is met. A loop can be any of the following:

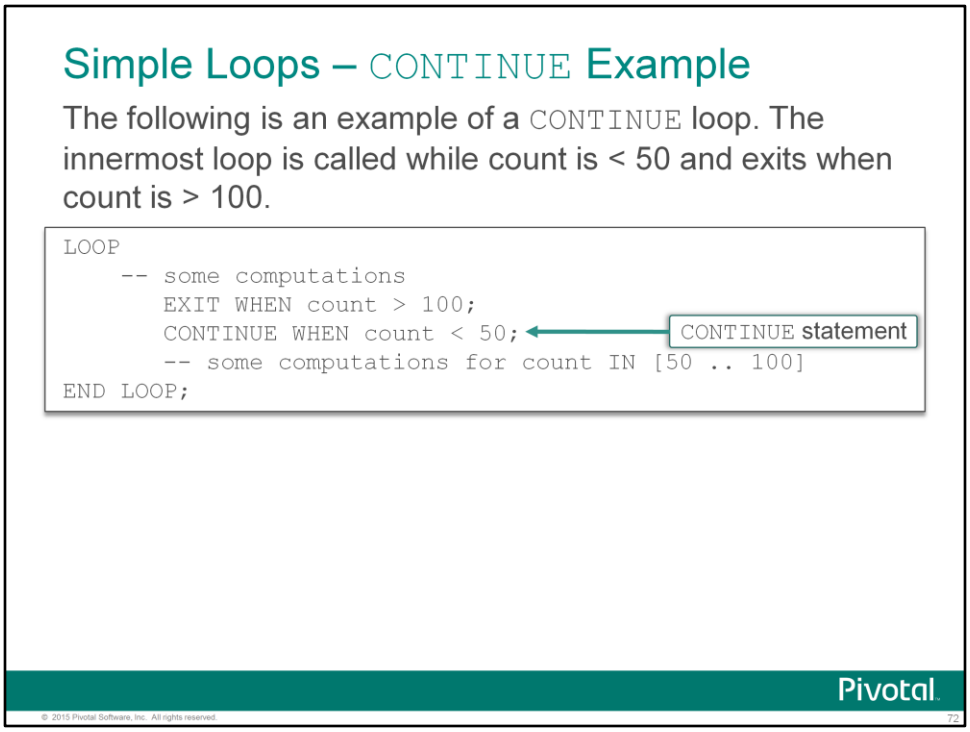
- **`LOOP`** – This defines an unconditional loop that is repeated indefinitely until an EXIT or RETURN statement is encountered.
- **`EXIT`** – This exits the inner most loop or block if no label is specified or an named outer loop if the label was defined for the loop or block.
- **`CONTINUE`** – The next iteration of an inner most loop is started if no label is provided. If a label is provided, it acts on the loop specified.
- **`WHILE`** – This repeats a sequence of statements until a condition has been met and the expression evaluates to true.
- **`FOR`** – Statements within the body of the `FOR` loop are executed over a range of integer values provided in the `FOR` statement. Once the last value in the range has been reached, the iteration stops.




Blocks or loops can be labeled to make it easier to reference them later in the code. The following conditions occur when working with loops:

- If no label is given, the innermost loop is terminated and the statement following `END LOOP` is executed next.
- If a label is given, it must be the label of the current or some outer level of nested SQL Procedural Language or block. Then the named loop or block is terminated and control continues with the statement after the corresponding `END` statement.
- If `WHEN` is specified, the loop exits occurs only if the boolean expression is true. Otherwise, control passes to the statement after `EXIT`.
- `EXIT` can be used with all types of loops and blocks. It is not limited to use with unconditional loops.

When used with a `BEGIN` block, `EXIT` passes control to the next statement after the end of the block.

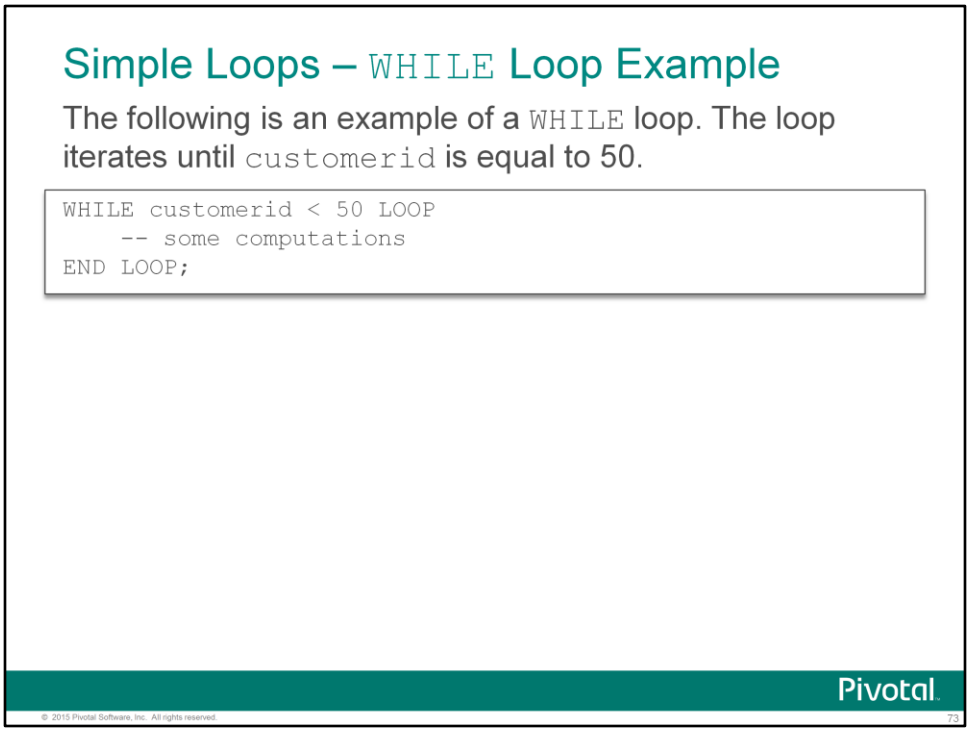



When working with CONTINUE statements:

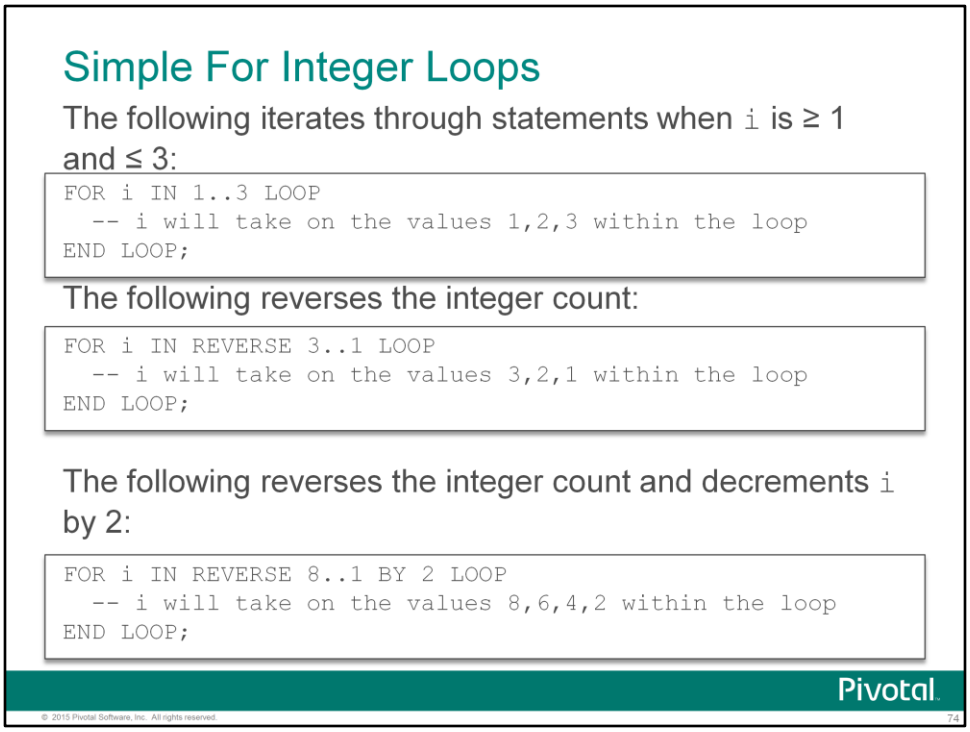
- If no label is given, the next iteration of the innermost loop is begun. All statements remaining in the loop body are skipped and control returns to the loop control expression, if any, to determine whether another loop iteration is needed.
- If the label is present, it specifies the label of the loop whose execution will be continued.

If WHEN is specified, the next iteration of the loop is begun only if the boolean expression is true. Otherwise, control passes to the statement after CONTINUE.

CONTINUE can be used with all types of loops; it is not limited to use with unconditional loops.




The `WHILE` statement repeats a sequence of statements so long as the boolean expression evaluates to true. However, you must control the condition by incrementing or decrementing the condition. The expression is checked just before each entry to the loop body.



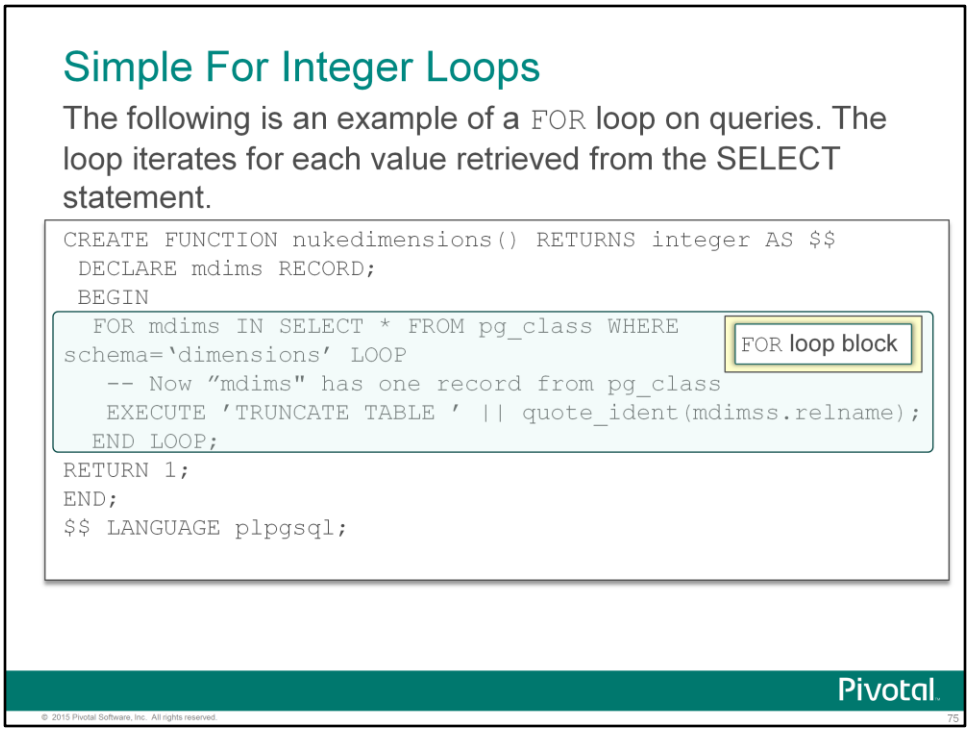

This form of `FOR` creates a loop that iterates over a range of integer values.

The variable name is automatically defined as type integer and exists only inside the loop. Any existing definition of the variable name is ignored within the loop. The two expressions giving the lower and upper bound of the range are evaluated once when entering the loop. If the `BY` clause is not specified, the iteration step is 1, otherwise, the value specified in the BY clause is used. The BY clause is evaluated once on loop entry.

If `REVERSE` is specified, then the step value is subtracted, rather than added, after each iteration.

If the lower bound is greater than the upper bound, or less than if `REVERSE` is used, the loop body does not execute. No error is raised.

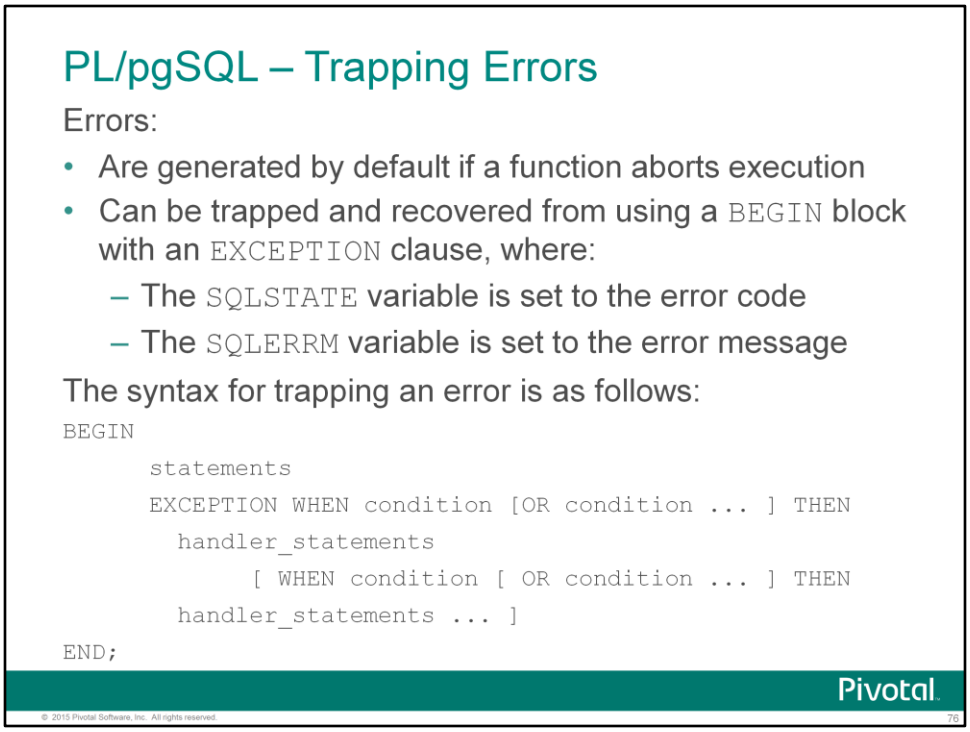
If a label is attached to the `FOR` loop, then the integer loop variable can be referenced with a qualified name, using that label.




Using a FOR loop for queries, you can iterate through the results of a query and manipulate that data accordingly.

In a `FOR` loop that works on queries, if the loop is terminated by an `EXIT` statement, the last assigned row value is still accessible after the loop. The query used in this type of `FOR` statement can be any SQL command that returns rows to the caller. A `SELECT` statement is the most common case, but you can also use `INSERT`, `UPDATE`, or `DELETE` with a `RETURNING` clause. Some utility commands such as `EXPLAIN` will work too.

PL/pgSQL variables are substituted into the query text, and the query plan is cached for possible re-use.




You can trap errors and recover from them by creating BEGIN blocks with an `EXCEPTION` clause.

A block containing an `EXCEPTION` clause is significantly more expensive to enter and exit than a block without one. Therefore, do not use `EXCEPTION` unless absolutely necessary.

Within an exception handler:

- The `SQLSTATE` variable contains the error code that corresponds to the exception that was raised.
- The `SQLERRM` variable contains the error message associated with the exception.

These variables are undefined outside of `EXCEPTION` handlers.

## PL/pgSQL – Common Exception Conditions

The following are a list of common error codes that can be generated:

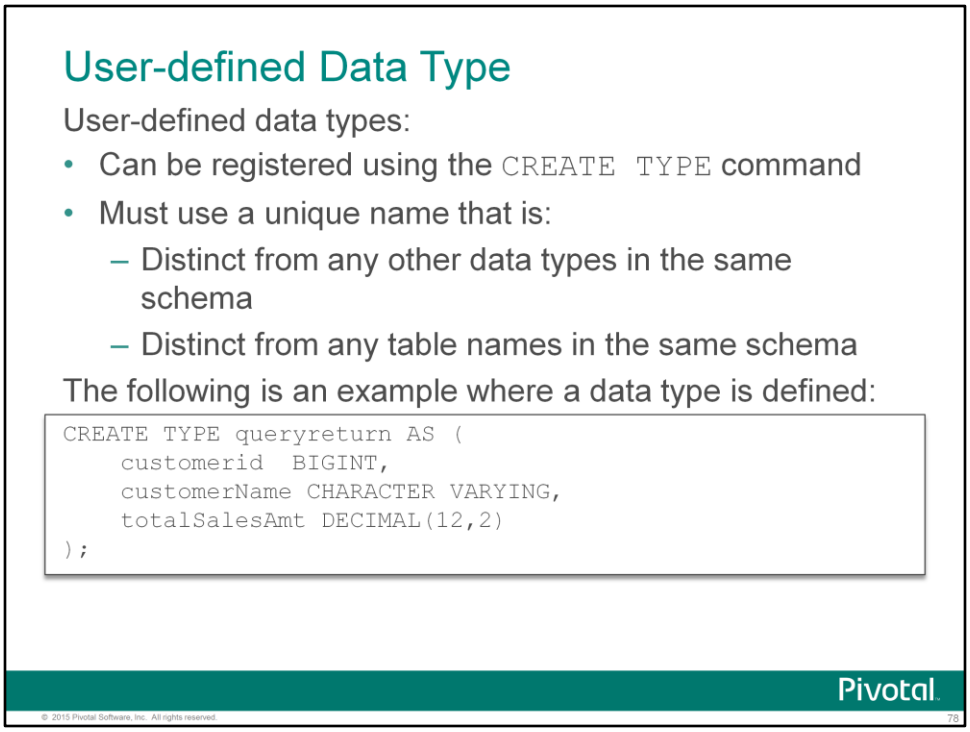
| Error Code | Description |
|---|---|
| CONNECTION_FAILURE | SQL cannot connect |
| DIVISION_BY_ZERO | Invalid division operation 0 in divisor |
| INTERVAL_FIELD_OVERFLOW | Insufficient precision for timestamp |
| NULL_VALUE_VIOLATION | Tried to insert NULL into NOT NULL column |
| RAISE_EXCEPTION | Error raising an exception |
| PLPGSQL_ERROR | PL/pgSQL error encountered at run time |
| NO_DATA_FOUND | Query returned zero rows |
| TOO_MANY_ROWS | Anticipated single row return, received more than 1 row |

Pivotal



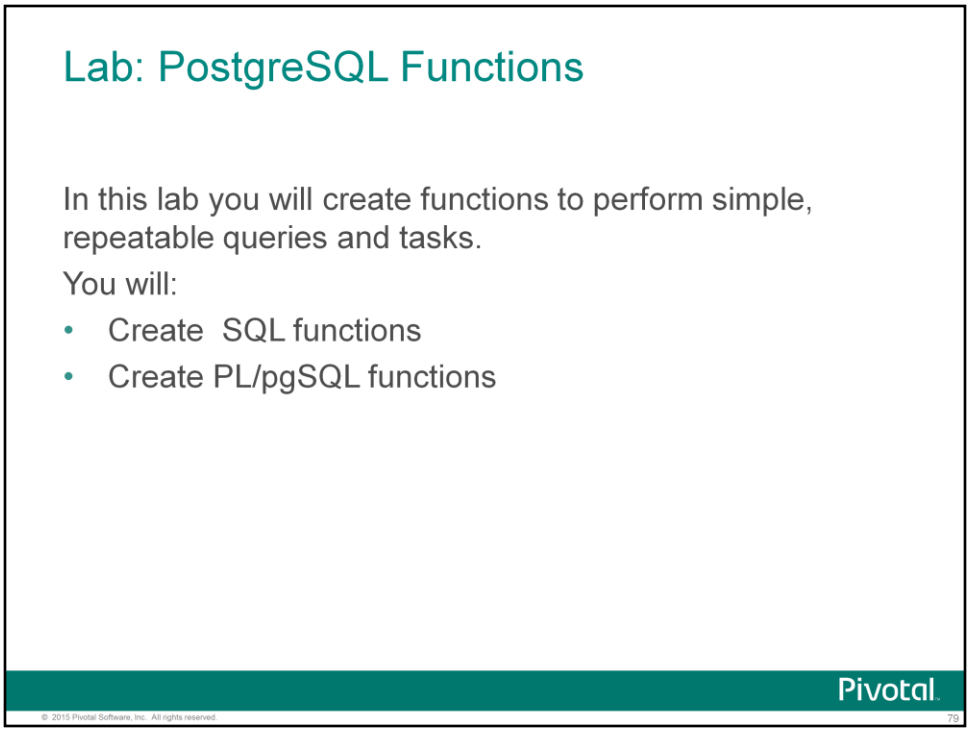


The slide shows a number of common error codes. A complete listing of error codes that can be captured are available in the PostgreSQL guide.

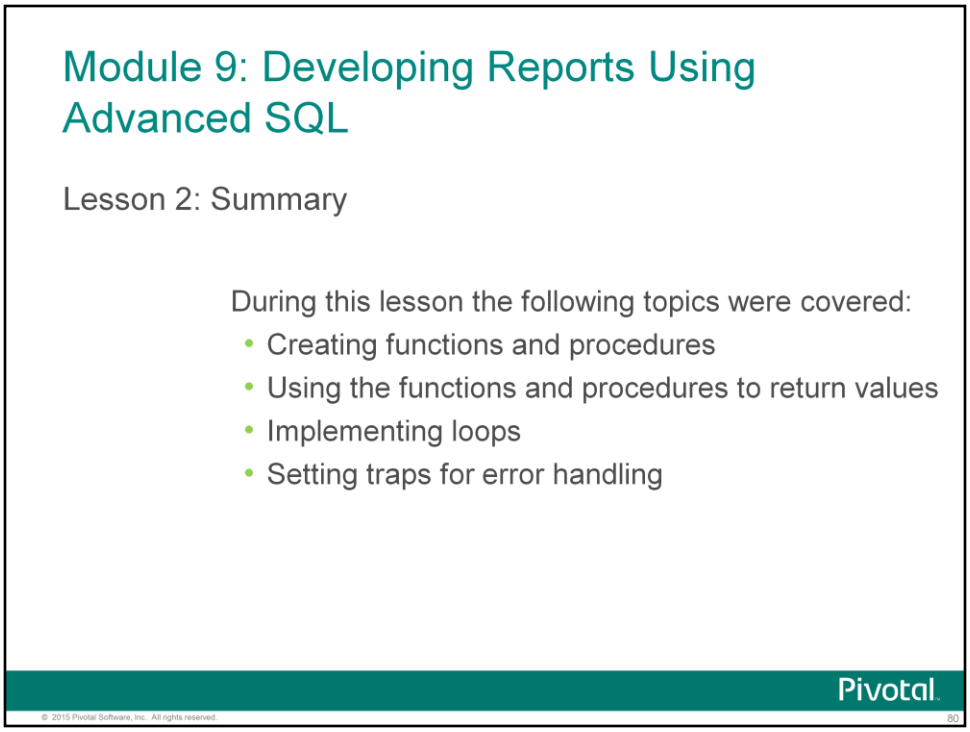



Often a function will need to return a data type other than one based on a table.

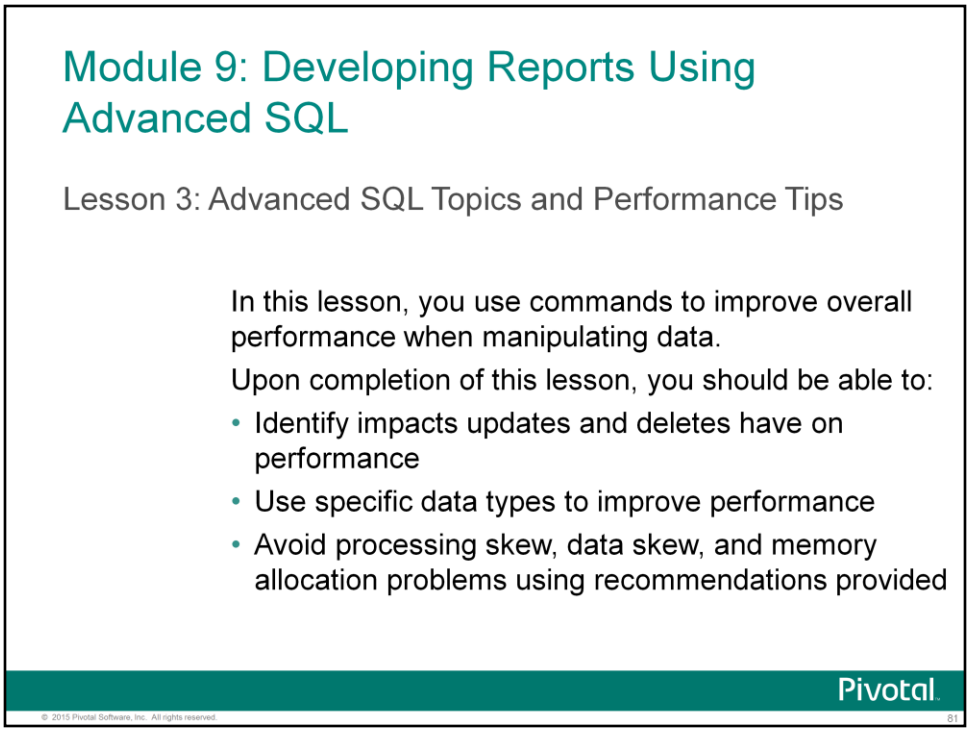
`CREATE TYPE` registers a new data type for use in the current database. The user who defines a type becomes its owner. If a schema name is given then the type is created in the specified schema. Otherwise it is created in the current schema. The type name must be distinct from the name of any existing type or domain in the same schema. The type name must also be distinct from the name of any existing table in the same schema.




In this lab you will create functions to perform simple, repeatable queries and tasks.



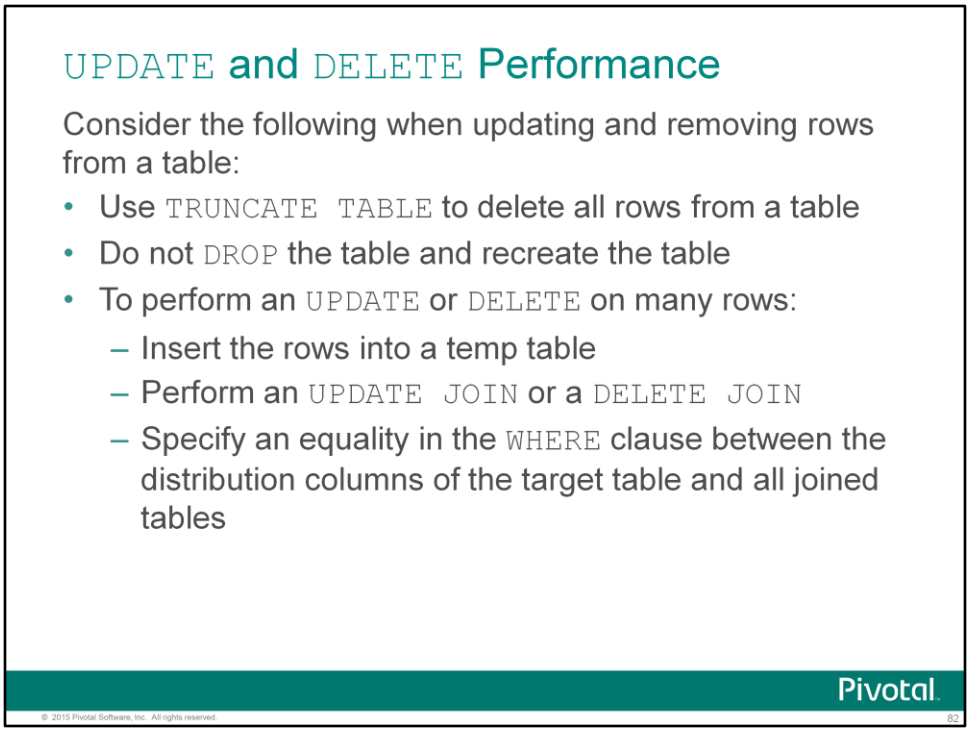

This lesson covered how to create and manage functions and procedures using a variety of programming languages available within the Greenplum environment. The various types of functions, input and output parameter definition, returning values, implementing loops, and setting traps for error handling were all discussed in the lesson.




Processing and data skew can have a strong effect on performance, especially when working with large data. An imbalance in how segments are loaded with data is directly impacted by how well you understand your data and how the data will be used. How you design for your data can directly impact performance, positively or negatively. Badly written SQL code or simply choosing one type of SQL command over another is another factor in poor performance. The end result is that you must learn to develop your code so that it does not, or reduces the chance, that it negatively impacts the performance for retrieving data.

In this lesson, you will:

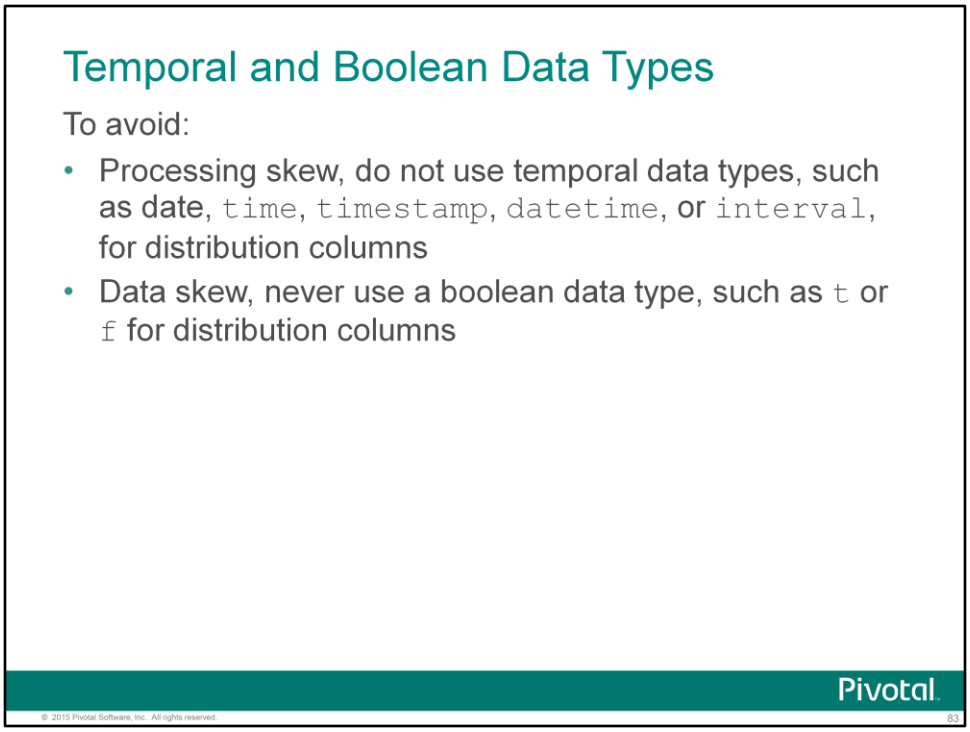
- Identify impacts updates and deletes have on performance.
- Identify the link between data types and performance.
- Use specific data types to improve performance.
- Avoid processing skew, data skew, and memory allocation problems using recommendations provided.




There are performance considerations when performing inserts, updates and deletes in a data warehouse environments:

- When all rows from a table must be deleted, always use `TRUNCATE TABLE` rather than dropping and re-creating the table.
- To perform an update or delete to many rows in a table, insert the rows into a `TEMP` table then perform an `UPDATE JOIN` or `DELETE JOIN` with the `TEMP` table and base table. An `UPDATE JOIN` or `DELETE JOIN` will run an order of magnitude faster than single updates or deletes.

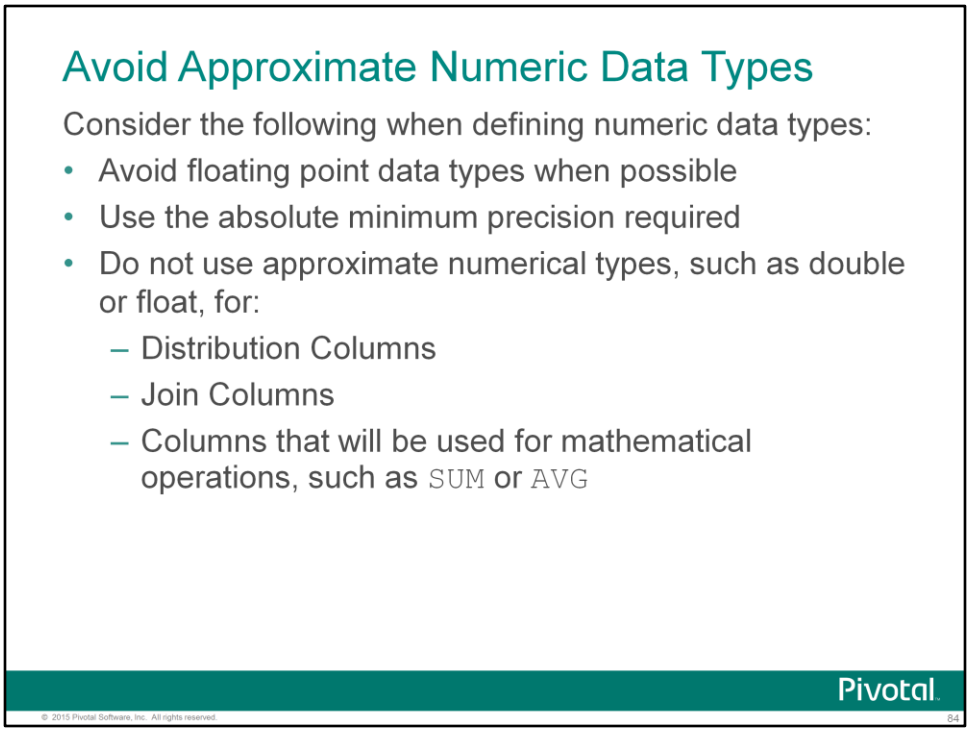
Keep in mind to perform an `UPDATE JOIN` or `UPDATE DELETE` the `WHERE` condition for the join must specify equality between corresponding distribution columns of the target table and all joined tables.




For some temporal formats, ordering of month and day in date input can be ambiguous. When possible, avoid using temporal data types for distribution columns to avoid processing skew.

If temporal data types are required, choose an optimal type. For example, use a `DATE (4 bytes)` if that is all that is needed rather than a `TIMESTAMP (8 bytes)`. If you need both a `DATE (4 bytes)` and `TIME (8 bytes)`, consider using a combined `TIMESTAMP (8 bytes)`.

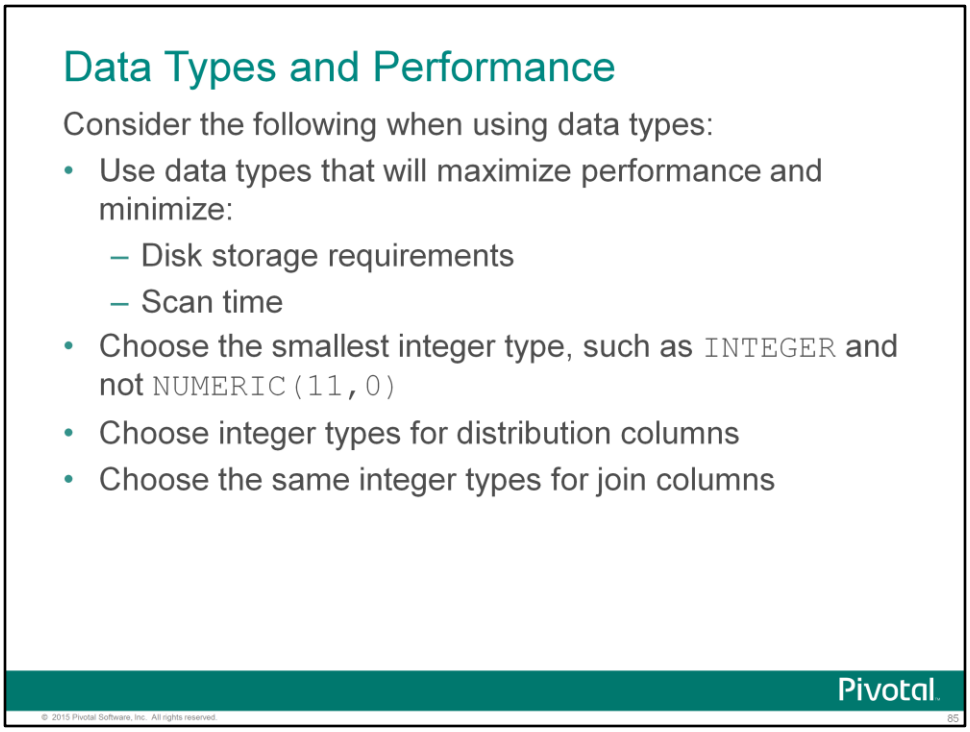
To avoid data skew, never use a boolean data type for distribution columns. Use a distribution key with unique values and high cardinality to distribute the data evenly across all segment instances.




Avoid the use of approximate numeric types, such as double or float, whenever possible to optimize performance in the Greenplum database.

Floating point data types inherently have performance implications in data warehouse environments. For example, a hash join can not be performed on a floating point data type rather a slower sort merge join is used.

Most importantly, do not use approximate numeric data types for distribution columns, join columns or columns that will be used for mathematical operations, such as `SUM` or `AVG`.



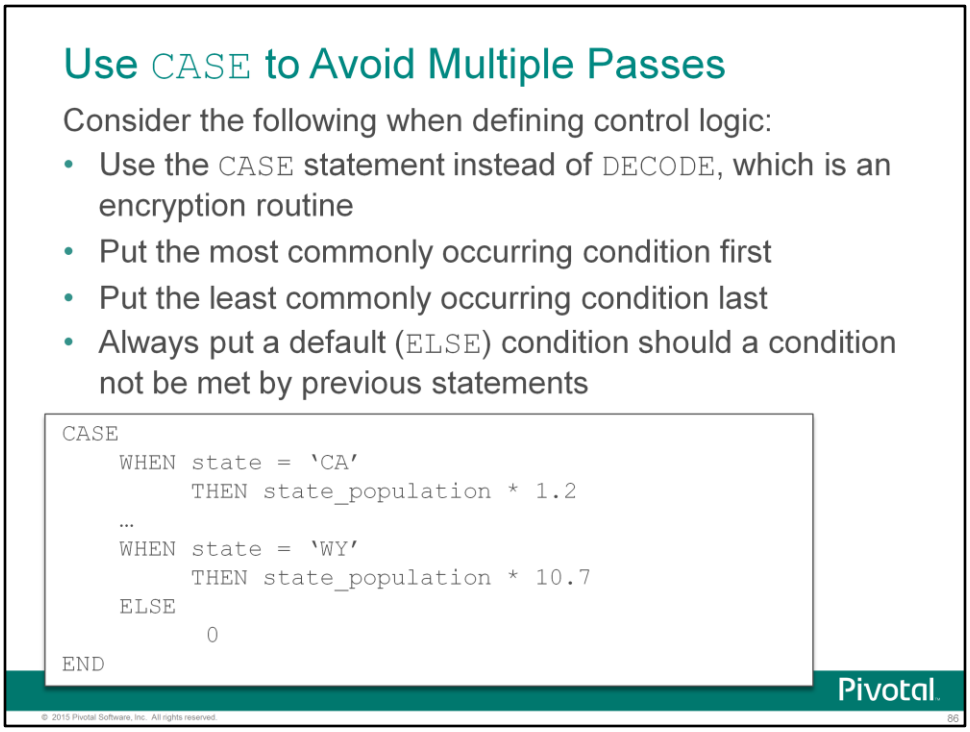

It is important to use care when choosing data types.

Ideally, the data type used will:

- Minimize disk storage
- Minimize scan time thereby maximizing performance.

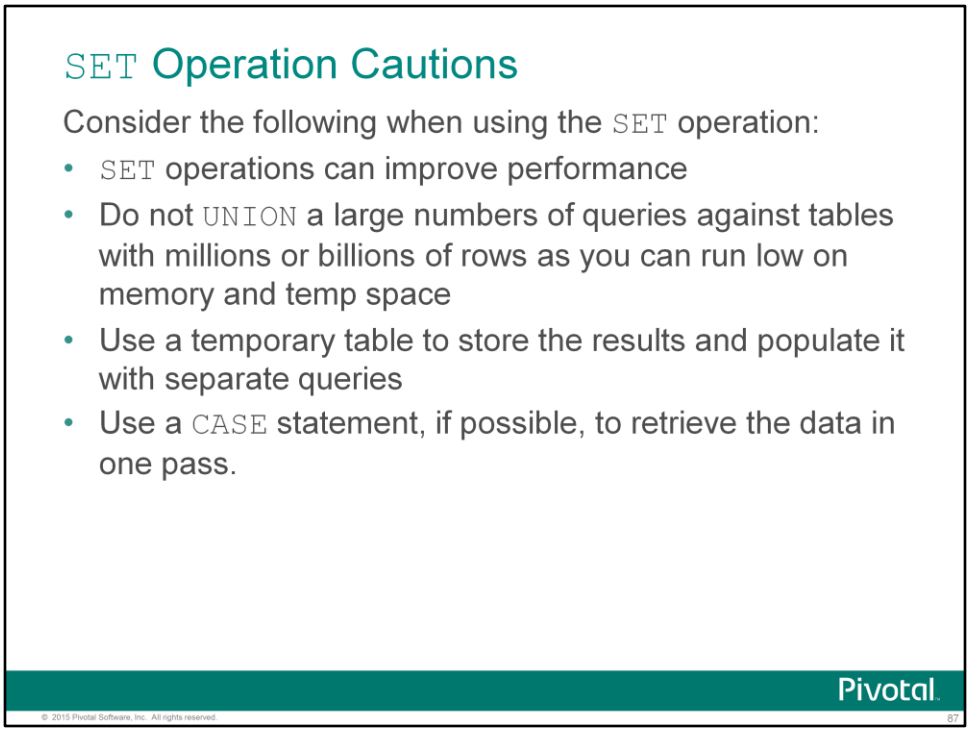
For numeric column data types, use the smallest data type to accommodate the data size. For example, use `SMALLINT` or `INT` instead of `BIGINT` to yield better performance and reduce storage space.

Choose integer types for distribution columns and choose the same integer types for commonly joined columns to maximize performance when joining tables.



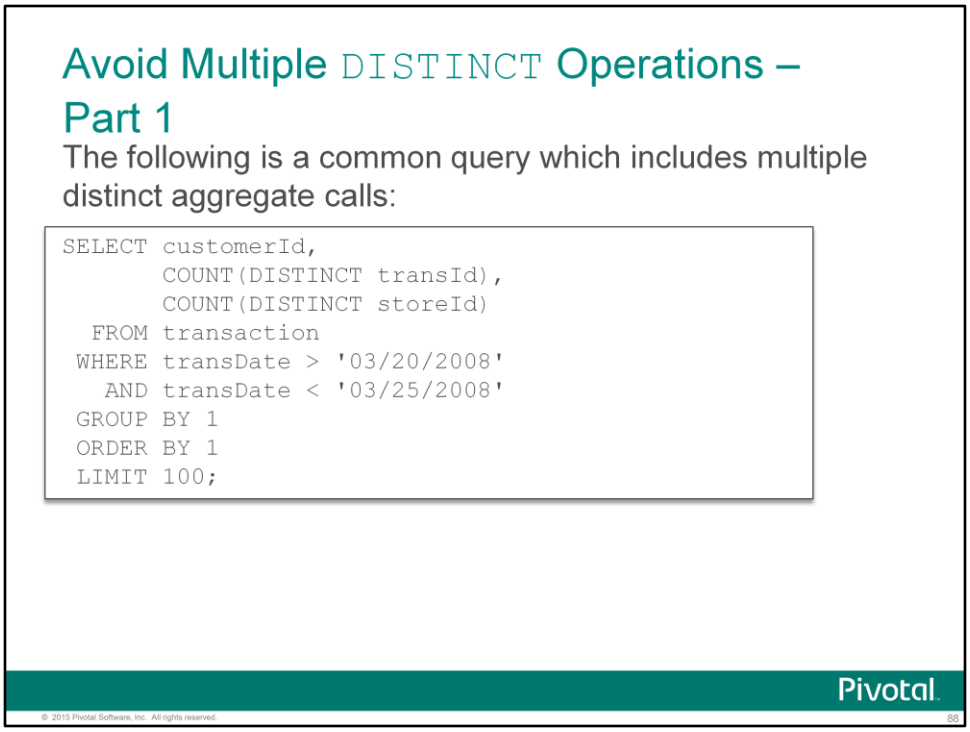

The CASE statement is similar to the IF/THEN/ELSIF/ELSE block. DECODE is an encryption routine in Greenplum, whereas in Oracle, DECODE is used as part of a case statement.

When defining CASE or IF/THEN/ELSE blocks, put the most commonly occurring condition first and the least commonly occurring condition last. This reduces the amount of checks that have to be performed before a condition is met.



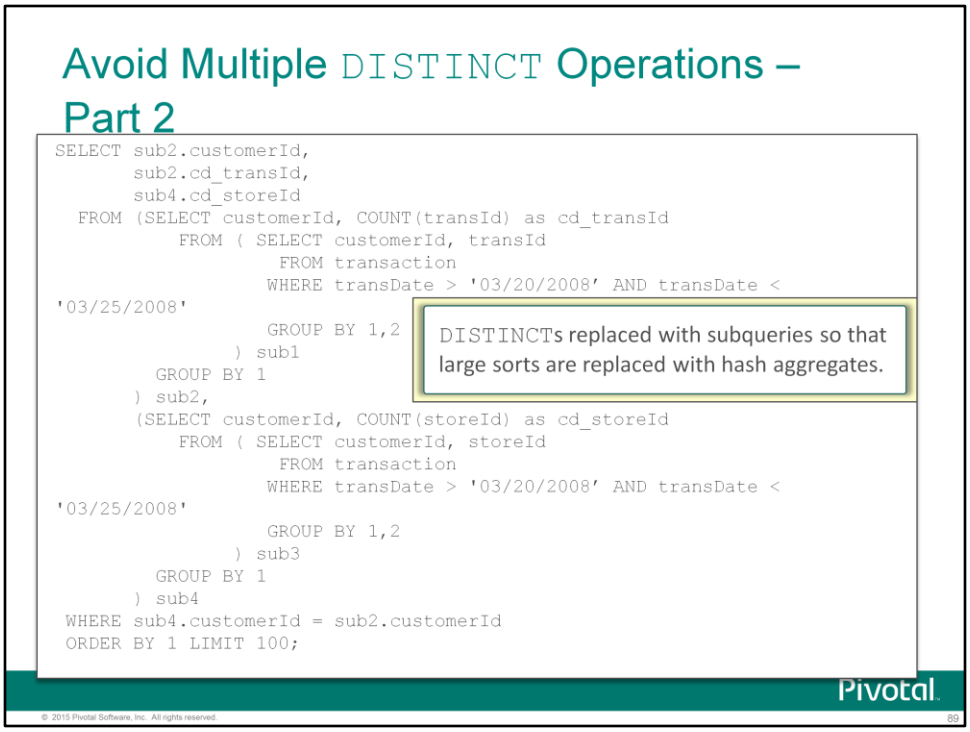

SET operations can improve performance overall. However, when using the SET operation, consider the following:

- Do not UNION a large numbers of queries against tables with millions or billions of rows. The optimizer will attempt to launch each query in parallel, resulting in running low or running out of memory and temp space.
- Use a temporary table to store the results and populate it with separate queries.
- Use a CASE statement, if possible, to retrieve the data in one pass.




The slide shows a fairly common query for a report. BI tools like Microstrategy and Business Objects often tend to use this type of syntax.

While you may not be able to affect code generated in tools, you can develop cleaner and more effective ad-hoc queries. Running an explain of this plan shows `GROUP AGGREGATE` functions which can have performance impacts.




Though it may not look as efficient, not having to sort on a huge table can have significant performance improvements.

Why is the second version faster? To execute the first query, the optimizer executes scans and SORTs of the data for each COUNT(DISTINCT) combination. This leads to a very resource intensive operations.

An explain of this example shows HASH AGGREGATES. Remember, it is better to replace large SORTs with hash aggregates.

## IsDate Function

The following function behaves like the IsDate function found in other DBMS systems:

```
CREATE OR REPLACE FUNCTION public.isdate(text)
   RETURNS boolean AS $BODY$
begin
   perform $1::date;
   return true;
exception when others then
   return false;
end
$BODY$
   LANGUAGE 'plpgsql' VOLATILE;
```

Pivotal





This function is not in the system, but is an excellent one to have on hand. It is a good utility function that allows you to verify whether or not a specified date is valid.

## Using Arrays

With Greenplum, it is possible to have an `ARRAY` data type with the following conditions:

- Arrays may not be the distribution key for a table
- You may not create indexes on `ARRAY` data type columns

The following is an example of an array that stores OIDs of dimension objects:

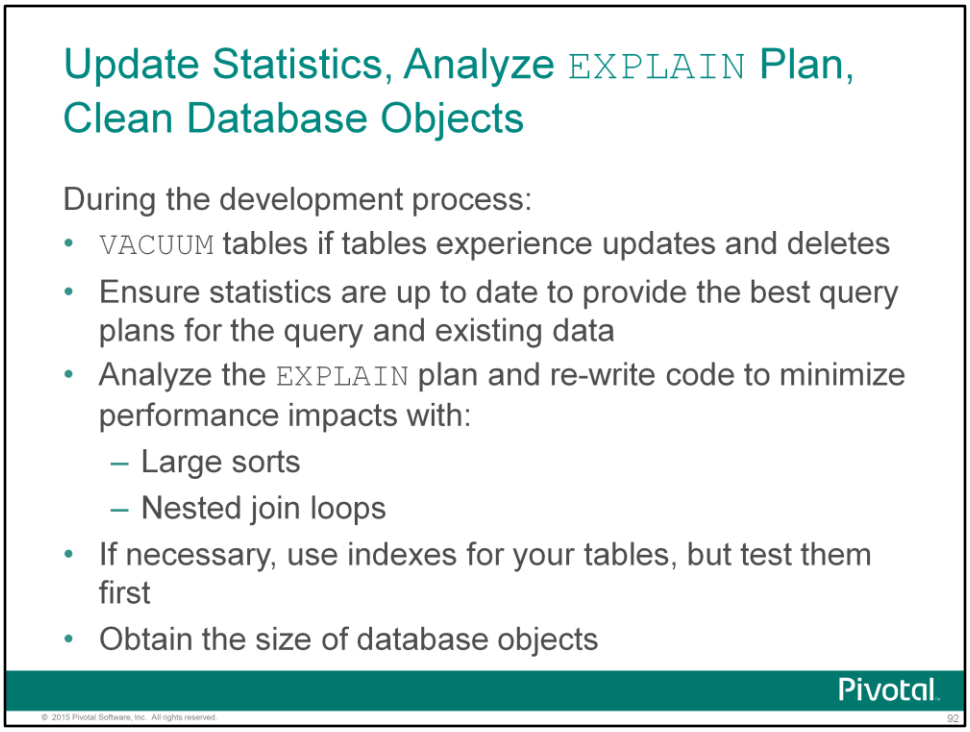
```
CREATE TEMPORARY TABLE dimensionsOID AS(
SELECT ARRAY(SELECT c.OID
               FROM pg_class c,
                    pg_namespace n
              WHERE n.oid = c.relnamespace
                AND n.nspname = 'dimensions')
             AS oidArray)
DISTRIBUTED RANDOMLY;
```

Pivotal





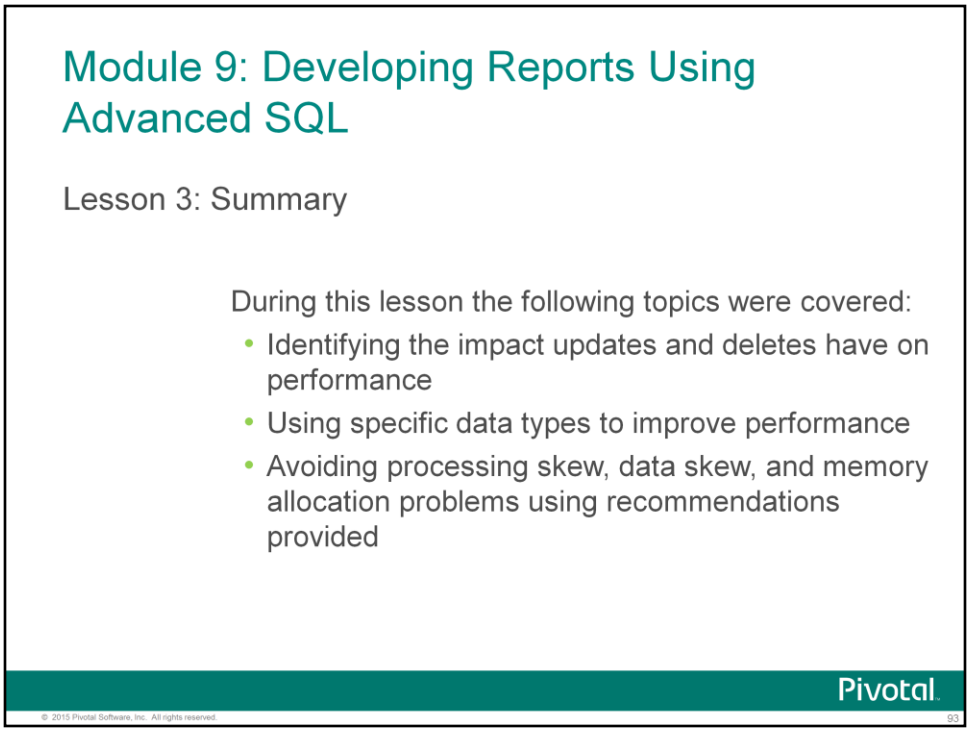
You can create an array, but you cannot index columns that are array data types.



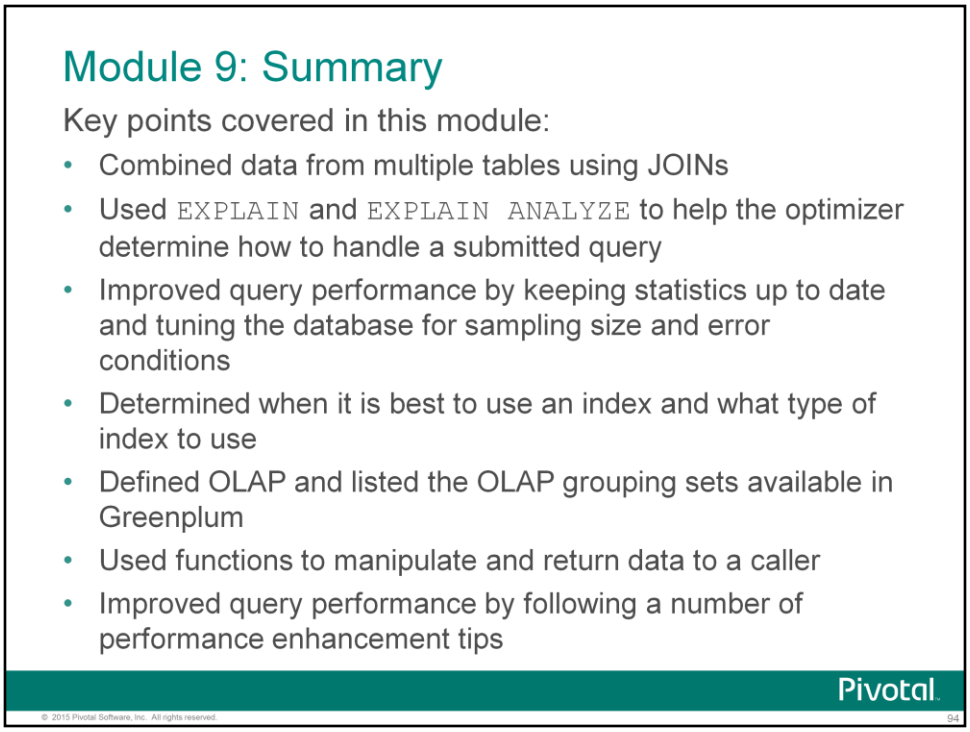

There are numerous things that you can do to improve the performance on your system. Remember to establish a baseline and establish reasonable goals against service level agreements.

When developing your code:

- Use `VACUUM` on tables to reduce bloat that may occur if the tables are constantly updated or deleted.
- Ensure statistics are up to date on the tables to get the best query plans that the optimizer can provide.
- Analyze the EXPLAIN plan to assess where the greatest time is being spent and reduce those times if possible through re-writes, changing data types, or balancing data on segments. In general for code development, eliminate large sorts and nested join loops. These can be resource hogs.
- If you are using indexes, use them appropriately. Ensure that they are being used by the optimizer and that they do have a positive impact on performance.
- In general, check the size of database objects, such as indexes and tables.




This lesson reviewed some of the performance topics discussed earlier in the course, but emphasized its impact  with regards to more advanced functions and behaviors. Specific cases on the impact of updates and deletes on performance, selecting appropriate data types for performance enhancements, and avoiding processing, data skew, and memory allocation problems were all discussed in the lesson.




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

- Combine data from multiple tables using `JOIN`s.
- Use `EXPLAIN` and `EXPLAIN ANALYZE` to help the optimizer determine how to handle a submitted query.
- Improve query performance by keeping statistics up to date and tuning the database for sampling size and error conditions.
- Determine when it is best to use an index and what type of index to use.
- Define OLAP and list the OLAP grouping sets available in Greenplum.
- Use functions to manipulate and return data to a caller.
- Improve query performance by following a number of performance enhancement tips.

## Course Summary

Key points covered in this course:

- Gained a basic understanding of data warehousing concepts and be able to describe PostgreSQL to develop a foundation for understanding the Greenplum solution.
- Learned about the Greenplum architecture and hardware solutions to support the architecture so that they understand how to properly implement a solution based on their company's business needs.
- Built and implemented a Greenplum software solution to manage the Greenplum environment and database through a variety of Greenplum utility tools and PSQL.
- Learned PostgreSQL and Greenplum specific SQL features and functions to manage data and optimize SQL query performance in a Greenplum database.

Pivotal




Listed are the key points covered in this course.

# Thank You

Pivotal


Pivotal
A NEW PLATFORM FOR A NEW ERA

This slide is intentionally left blank.

Pivotal