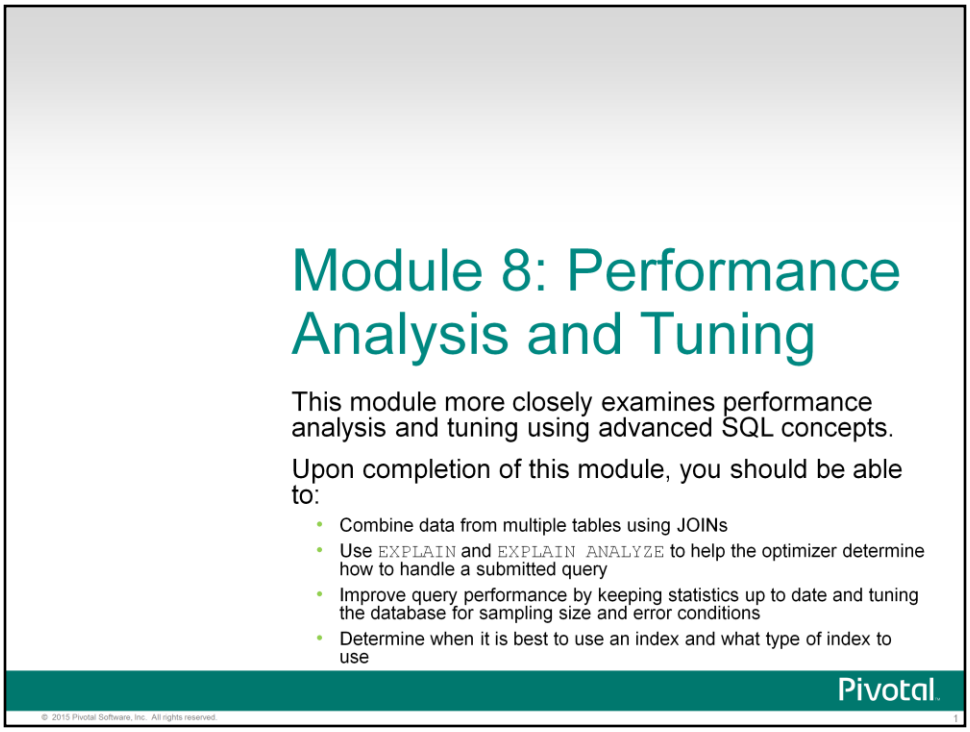
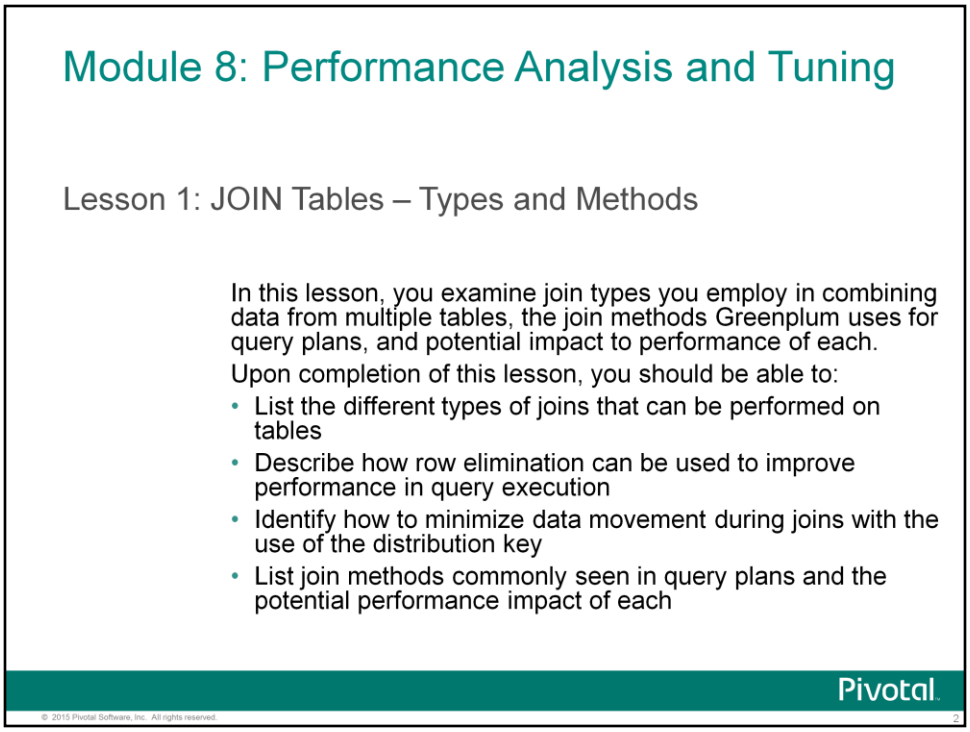


This module approaches performance analysis and tuning through advanced SQL topics. You learn about joining tables and the impact that different types of joins can have on overall performance. Statistics collection and analysis continue to play a key role in reducing the time the query planner and the query perform. Indexes are not always beneficial to your data, so we examine specifically how they can be used. OLAP groups and windowing functions offers benefits not only for coding, but also to improve performance.

In this module, you will:

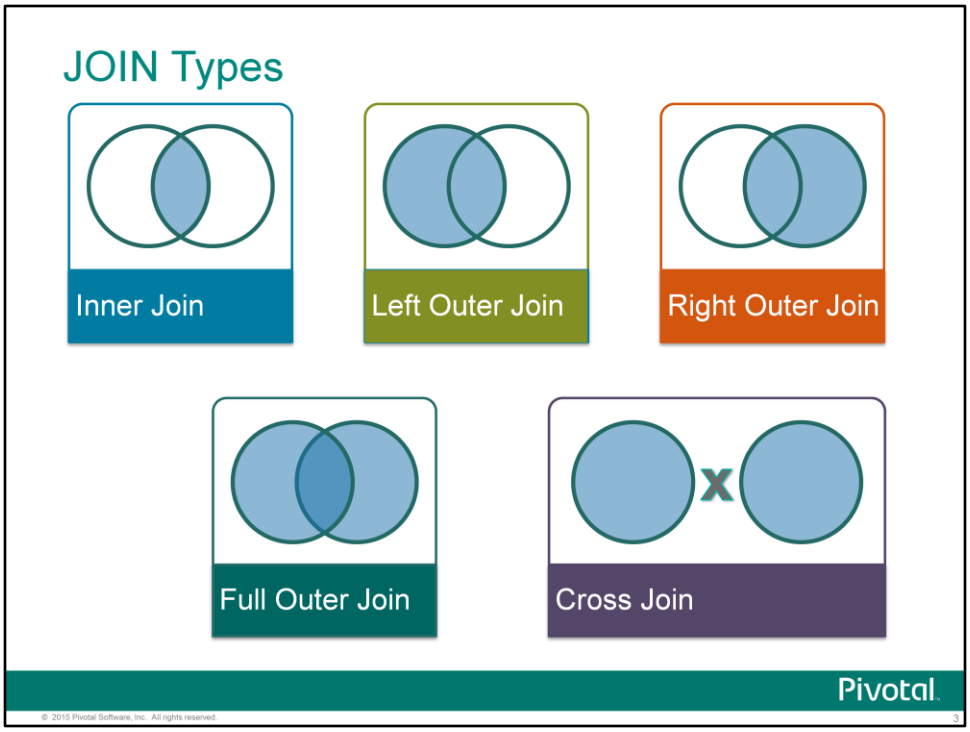
- Combine data from multiple tables using JOINs.
- Use `EXPLAIN` and `EXPLAIN ANALYZE` to help the optimizer determine how to handle a submitted query.
- Improve query performance by keeping statistics up to date and tuning the database for sampling size and error conditions.
- Determine when it is best to use an index and what type of index to use.



When data is separated across multiple tables, joins are a common way of combining the data from two or more database tables to create a single set of data. Maintaining data in different table, such as when using the 3NF data model, can make it easier to maintain, but it can have performance impacts when querying against that data to return meaningful subsets of data.

In this lesson, you will:

- List the different types of joins that can be performed on tables.
- Describe how row elimination can be used to improve performance in query execution.
- Identify how to minimize data movement during joins with the use of the distribution key.
- List join methods commonly seen in query plans and the potential performance impact of each.



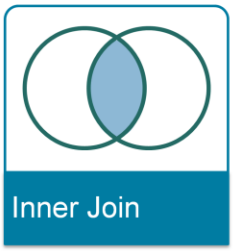
The five major types of joins that will be discussed in this lesson include:

- **Inner join** – The inner join is possibly the most common type of join. The resulting data set is obtained by combining two tables on a common column. Each row of the left table is compared against each row of the right table. All matching rows are returned as part of the result set. An equijoin is an inner join that uses only equality comparisons in the join predicate.
- **Left outer join** – Left outer join returns all of the rows from the left table even if there is no matching row in the right table. It also returns matching rows from the right table. Rows in the right table that do not match are not included as part of the result set.
- **Right outer join** – Right outer join returns all of the rows from the right table even if there is no matching row in the left table. It also returns the matching rows from the left table.
- **Full outer join** – Full outer join returns all rows from both tables where there is a match and returns NULL for rows that do not have a match.
- **Cross join** – Cross join returns the Cartesian product of rows from tables in the join. The resulting data set consists of a combination of each row in the left table with each row in the right table. Two tables, each with five rows, will produce a resulting data set that contains twenty-five rows.

Inner Join

Inner joins:

- Can be simple to write
- Require a join condition be specified
- Are also known as a SIMPLE JOIN or EQUI-JOIN



The following is an example of an inner join:

Example: Join two tables using an equijoin

```
SELECT
      EMP.EMPLOYEE_ID
    , EMP.EMPLOYEE_NAME
    , DPT.DEPT_NAME
FROM
      EMPLOYEES   EMP
    , DEPARTMENTS DPT
WHERE
      EMP.DPT_ID = DPT.DPT_ID;
```

Pivotal

4

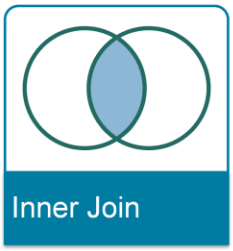
The syntax for an inner join is simple to develop. It does require that a join condition be specified – one cannot be implied. It is also known as a simple join or an equi-join.

In the inner join example shown, the `DPT_ID` columns of the `EMP` and `DPT` table are compared to each other. This query only retrieves rows based on the following join condition: A corresponding row must exist in each table, thus the term INNER JOIN.

ANSI Syntax for Joins

Example: Join two tables using an inner join

```
SELECT
      EMP.EMPLOYEE_ID
    , EMP.EMPLOYEE_NAME
    , DPT.DEPT_NAME
FROM
    EMPLOYEES        EMP
    INNER JOIN DEPARTMENTS  DPT
    ON ( EMP.DPT_ID = DPT.DPT_ID );
```



Example: Alternative usage of INNER without the JOIN

```
...
FROM
    EMPLOYEES     EMP
    JOIN DEPARTMENTS DPT
    ON EMP.DPT_ID = DPT.DPT_ID;
```

Pivotal

5

Both examples on the slide are alternative methods for writing an inner join. The first syntax uses the term `INNER JOIN` between the two tables followed by the predicate `ON (`*`table1.column = table2.column`*`)`.

The second example uses `JOIN` between the two tables with the same `ON` predicate.

Note that you must use ANSI syntax for any join other than `INNER`.

Left Outer Join

A left outer join:

- Returns all rows from the left table and rows
  from the right table that match the left one
- Is used when the right table can supply information missing from the left table
- Can be referenced as `LEFT OUTER JOIN` or `LEFT JOIN`

Example: LEFT OUTER JOIN example

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

Pivotal

6

LEFT OUTER JOIN:

- Returns all rows from the left table and only those rows from the right table that match the left one. This is handy when there may be missing values in the left table, but you want to get a description or other information from the right table.
- Is defined with the syntax `LEFT OUTER JOIN` or abbreviated to `LEFT JOIN`.
- Extends the rows from the left table to the full width of the joined table by inserting null values for the right-hand columns.

Right Outer Join

A right outer join:

- Is the inverse of the `LEFT OUTER JOIN`
- Is functionally the same as the `LEFT OUTER JOIN`, but care must be taken when specifying the left and right tables
- Can be referenced as `RIGHT OUTER JOIN` or `RIGHT JOIN`

Example: RIGHT OUTER JOIN example

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

Pivotal

A right outer join:

- Returns all the joined rows, plus one row for each unmatched right-hand row.
- Is functionally the same as the `LEFT OUTER JOIN`. However, in either case, you must take care to choose the left table and the right table.
- Can be referenced either as `RIGHT OUTER JOIN` or `RIGHT JOIN`.

Full Outer Join

Full outer join:

- Is used to retrieve every row irrespective of match
- Can be used to detect *true changes*

Example: FULL OUTER JOIN example

```
SELECT COALESCE( ORIG.transid, STG.transid) AS transid,
    CASE
    WHEN ORIG.transid IS NULL AND STG.transid IS NOT NULL
      THEN 'INSERT'
    WHEN ORIG.transid IS NOT NULL  AND STG.transid IS NOT NULL
THEN 'UPDATE'
    WHEN ORIG.transid IS NOT NULL  AND STG.transid IS NULL
      THEN  'DELETE'
    ELSE 'UNKNOWN'
  END AS Process_type
FROM  facts.transaction          ORIG
  FULL OUTER JOIN staging.transaction_w  STG
  ON STG.transid = ORIG.transid;
```

Pivotal

8

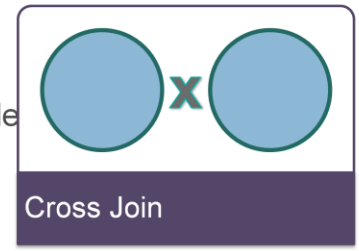
This interesting join is used when you want to retrieve every row, whether there is a match or not based on the join criteria. This type of join can be used for detecting *true changes* for a table.

In the example shown, the `COALESCE` function ensures that a transaction ID is retrieved to work with in subsequent processing. The `FULL OUTER JOIN` syntax joins on the `transid` column that the `COALESCE` function is performing a check on.

Cross Join (Cartesian Product)

A cross join:

- Oftentimes is used unintentionally
- Combines every row in the left table with every row in the right table
- Is specified with the CROSS JOIN syntax or by comma separated tables with no predicates



Example: CROSS JOIN example

```
SELECT
  t.transid,
  t.transdate,
  COALESCE( c.custname, 'Unknown Customer') AS custname
FROM facts.transaction t, dimensions.customer c;
```

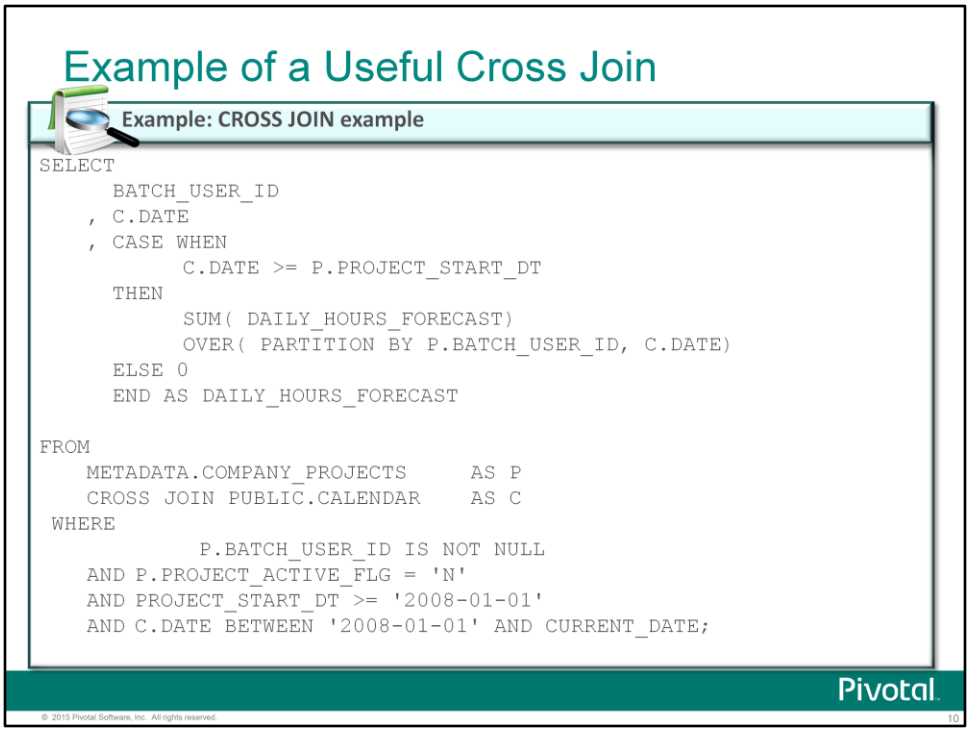
Pivotal

A cross join may not be the type of join you should use when working on large data. This can result in huge transactions.

For example, if you have a billion row table and `CROSS JOIN` it to a 100 row table, your resulting set will have 100 billion rows.

It can take much longer to perform a cross join when working with large data. For example, performing a 23 trillion row cross join in one table could take hours to complete the transaction. However, selecting against the table did eventually yield the desired result.

The syntax shown defaults to a cross join, so care should be taken when using this type of syntax. Cross joins can be specified with the `CROSS JOIN` syntax.



The example shown on the slide is an example of real use of the `CROSS JOIN`. The code aggregates all of the forecasts for every project, every day, from the Jan 1st, 2008 until today.

Row Elimination

During join executions:

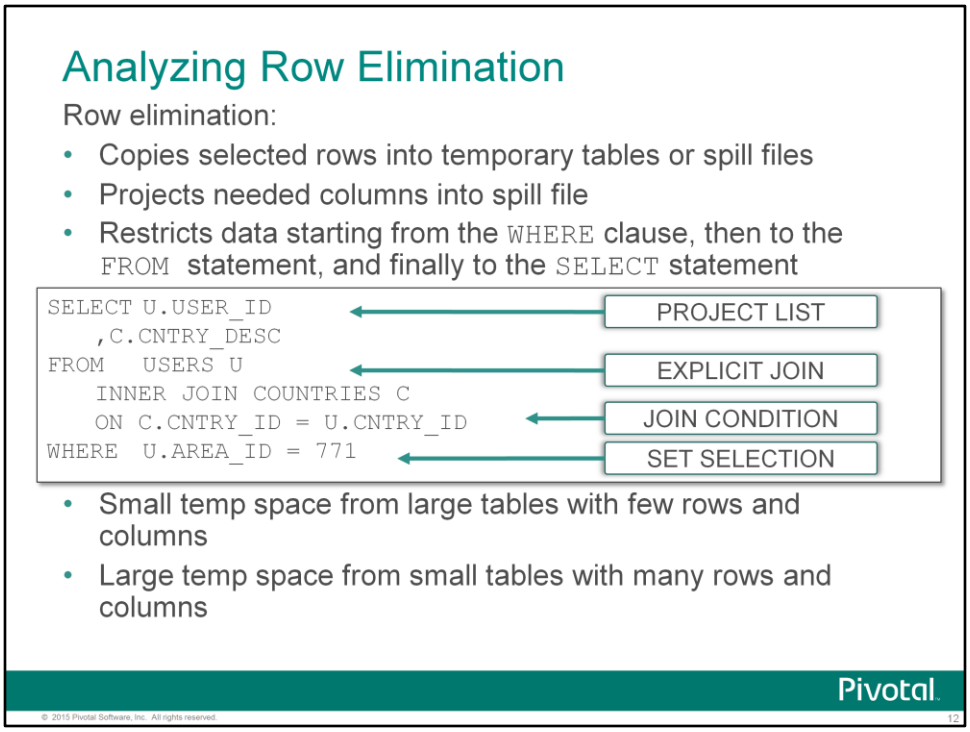
- Greenplum often has to use disk space to temporarily store data.
- Query optimizer minimizes the amount of memory and disk space required by:
  - Projecting (copying) only those columns that the query requires
  - Doing single-table set selections first (qualifying rows)
  - Putting only the smaller table into the spool whenever possible

Note: Non-equality join operators produce a partial Cartesian product. Join operators should always be equality conditions.

Pivotal

11

When Greenplum executes a plan, it attempts to eliminate as many rows as possible. The optimizer tries to be as restrictive as possible, before the joins, to reduce the number of rows returned for the join. The intention is that the join is performed only rows and columns needed for the join.



The example on the slide shows four columns the query is working on:

- The first restriction is developed using the `WHERE` clause. It restricts the number of rows returned.
- Next, the columns specified in the `FROM` statement determine which columns are required by the query.
- Finally, the columns selected are specified in the `SELECT` statement.

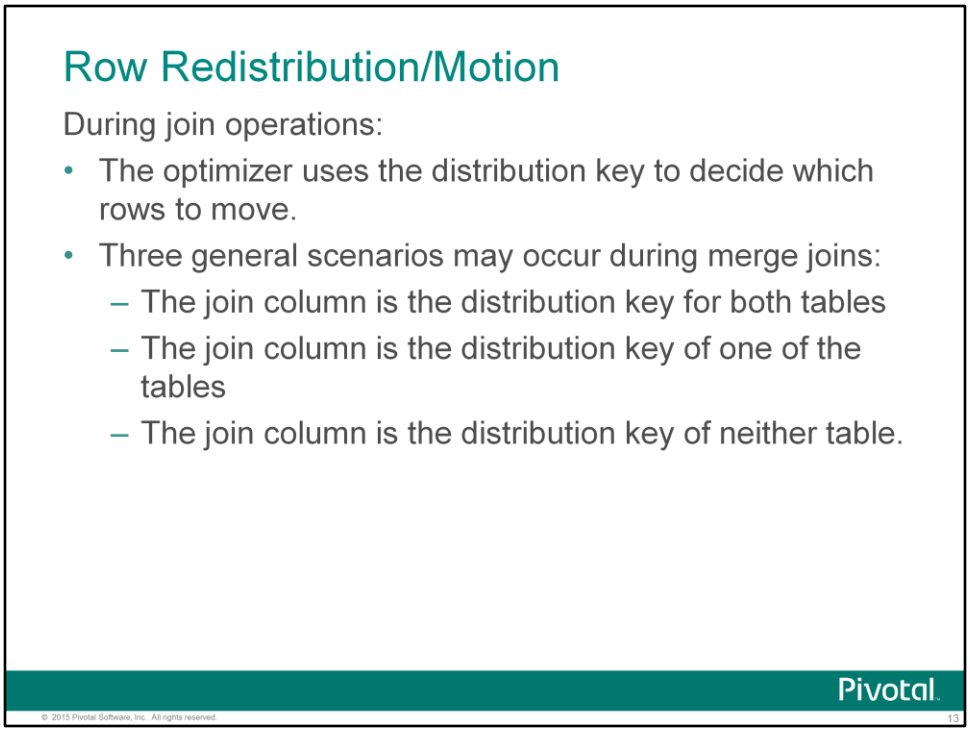
Note that you must make sure your data is as clean as possible before the optimizer is invoked as part of the query plan. If the data is not cleaned, vacuum and analyze the tables. This removes any old or stale data, as would occur during high read/write transactions, such as when deleting or updating rows in a table.

In the example shown:

- Row selection is performed as soon as possible. Only rows from the `USERS` table with a `AREA_ID` of `771` are qualified into a temporary table or spill file.
- The Optimizer will typically *project* only the columns needed into a spill file. In this example, the columns, `USER_ID`, `CNTRY_ID`, `CNTRY_DESC` are projected.

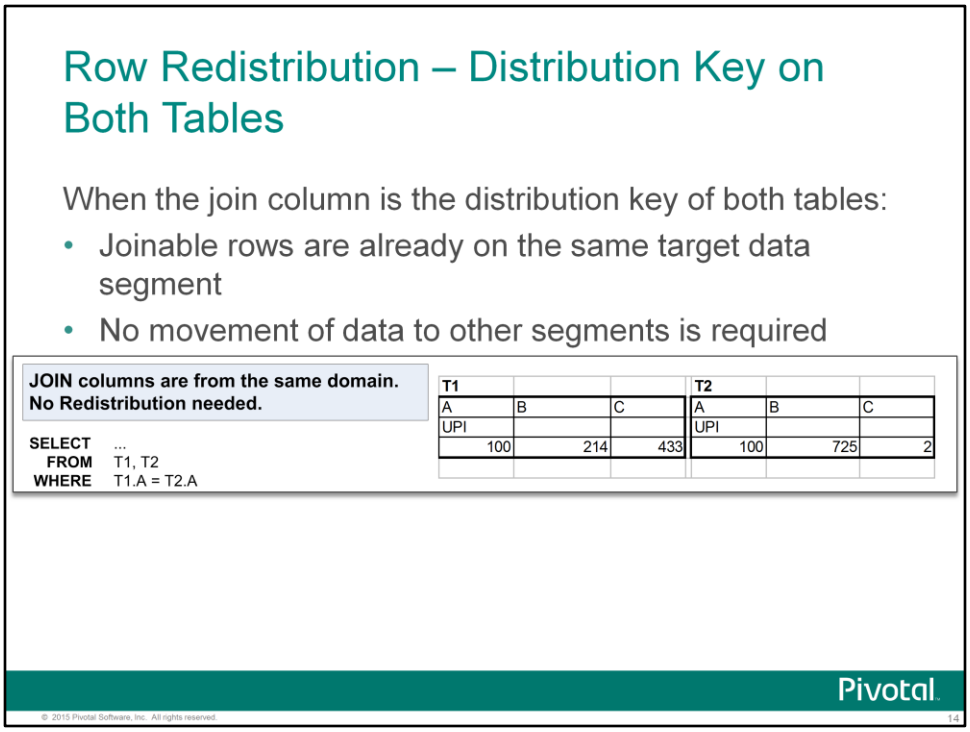
The temporary space created for row elimination has the following uses:

- Small temporary space from large tables with few rows or columns projected.
- Large temporary space from small tables with many rows or columns projected.



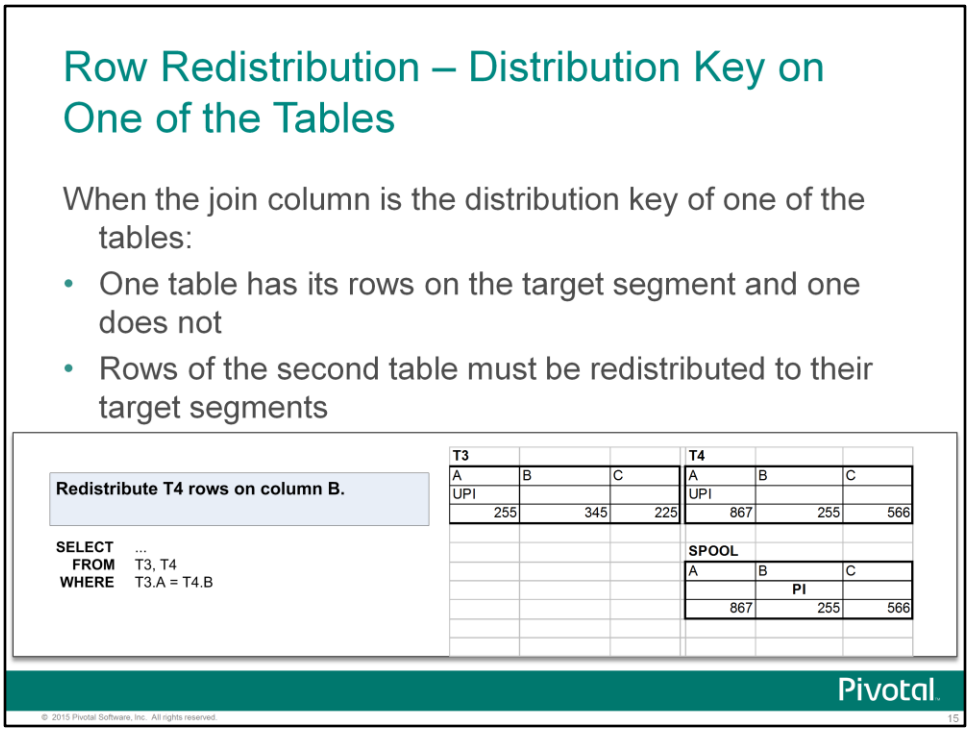
Data is moved around in spool, temporarily, during redistribution. The optimizer considers how to move data based on the distribution key. There are three general considerations when merging rows:

- The join column is the distribution key for both tables.
- The join column is the distribution key of one of the tables.
- The join column is not the distribution key for either table.



When the join column is the distribution key on both tables:

- Joinable rows are already on the same target data segment, since equal distribution key values always hash to the same data segment.
- No movement of data to other segments is required.
- The rows are already sorted in hash sequence because of the way in which they are stored by the file system.
- With no need to sort or move data, the join can take place immediately.



When the join column is the distribution key for one of the tables:

- One table has its rows on the target segment and one does not.
- The rows of the second table must be redistributed to their target segments by the hash code of the join column.
- If the table is small, the optimizer may decide to simply duplicate the entire table on all segments instead of hash redistributing the data.
- The rows of one table will be copied to their target data segments.

Row Redistribution – Distribution Key on Neither of the Tables

When the join column is not the distribution key for either of the tables:

- Both tables may require preparation for joining
- Various combinations of hash distributions or table duplications may be used
- This approach involves the maximum amount of data movement

Redistribute T5 rows on B.
Redistribute T6 rows on C.

```
SELECT   ...
  FROM   T5, T6
 WHERE   T5.B = T6.C
```

T5

A	B	C
UPI		
456	777	876

T6

A	B	C
UPI		
993	228	777

SPOOL

A	B	C
	PI	
456	777	876

SPOOL

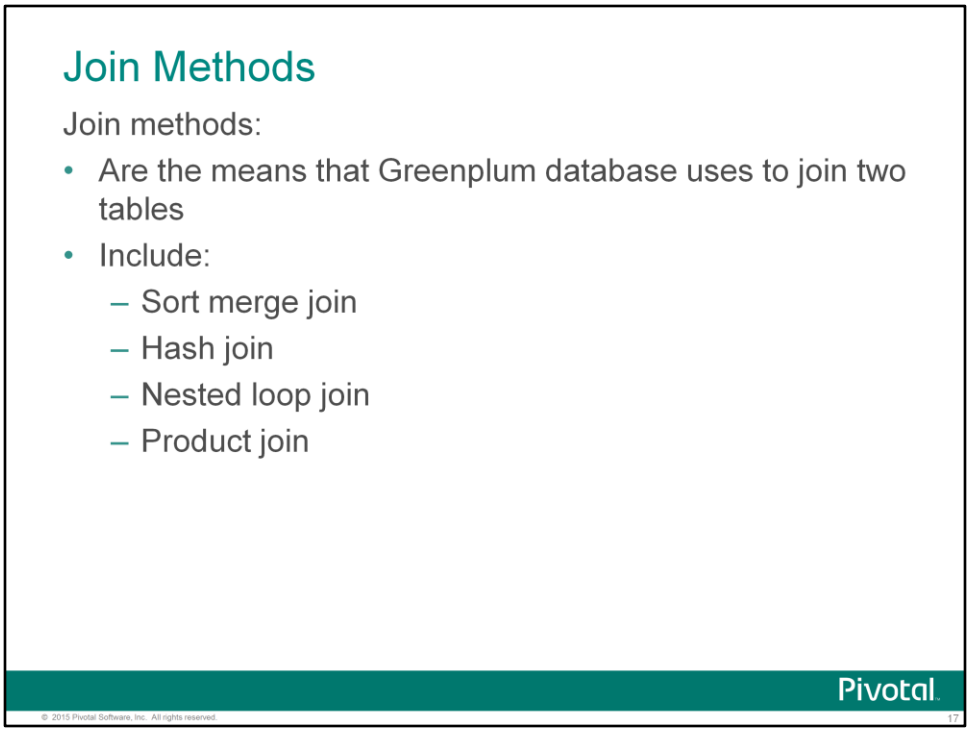
A	B	C
		PI
993	228	777

Pivotal

16

When the join column is not the distribution key for either tables in the join:

- Both tables may require preparation for joining.
- This may involve various combinations of hash distributions or table duplications.
- This approach involves the maximum amount of data movement.

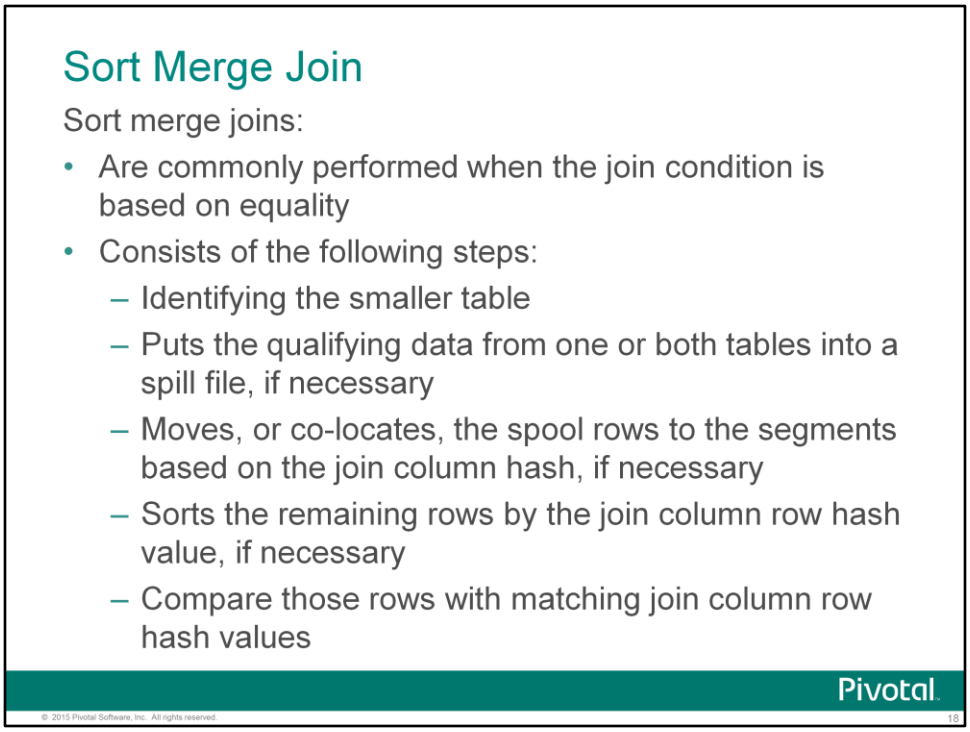


These are the join methods seen in a query plan:

- Sort Merge Join
- Hash Join
- Nested Loop Join
- Product Join

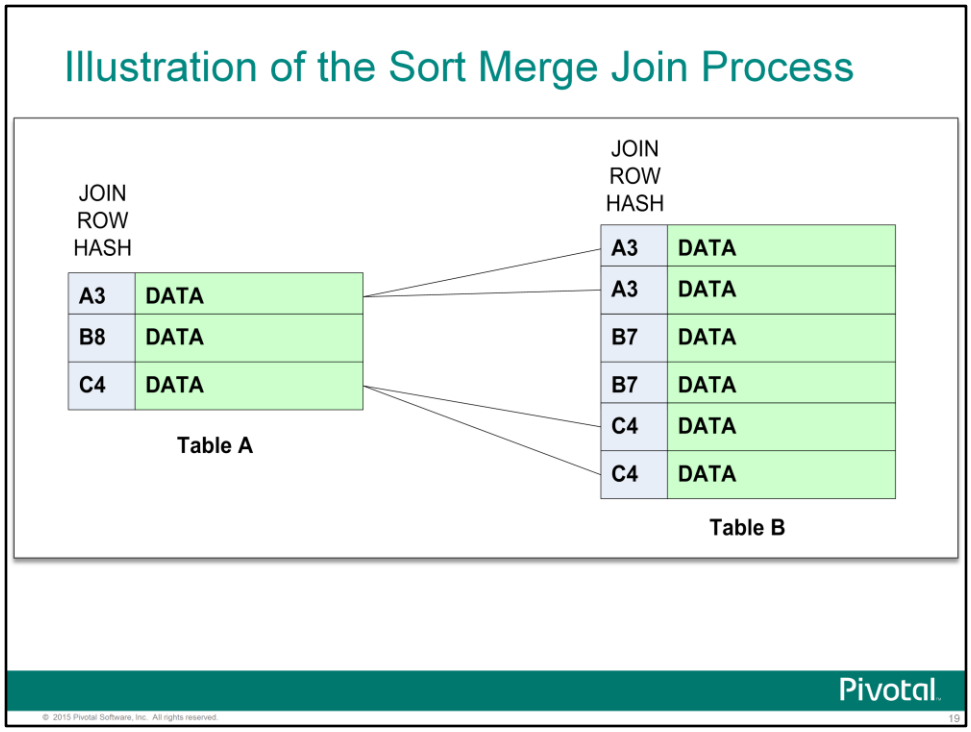
Code designers and developers should question the use of nested loop and product joins in their queries.

Nested loop joins have the potential to run for a very long period of time.

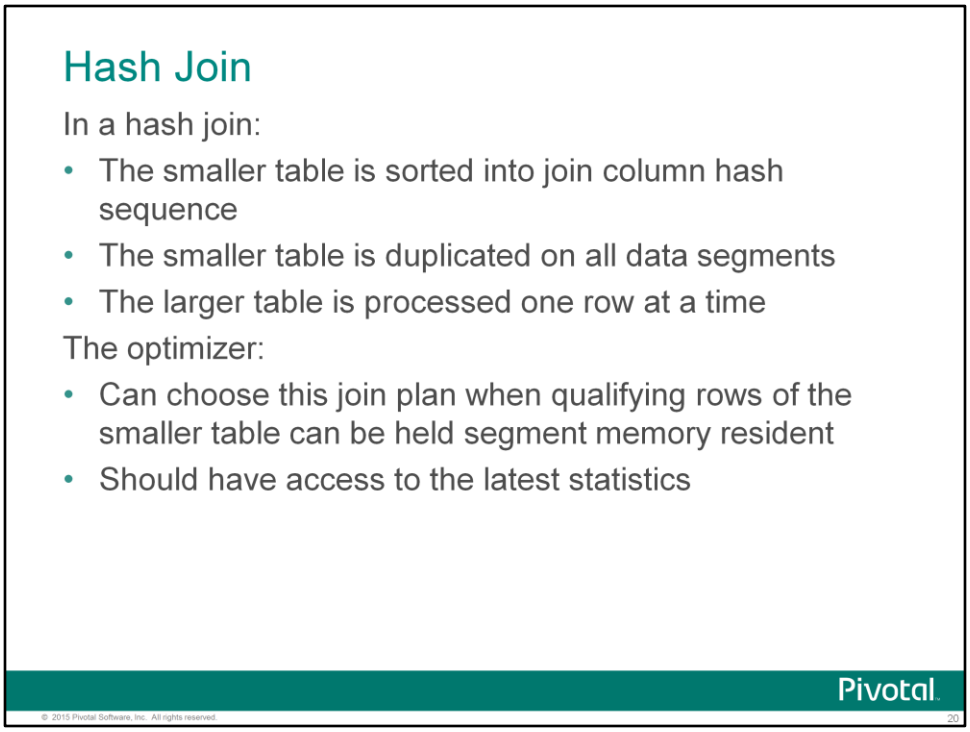


Sort merge joins are commonly performed when the join condition is based on equality. It consists of several steps, the main ones involving identifying the smaller table and comparing its rows against the other table's rows using the join column hash values.

Placing data in the temporary space, moving rows to other segments, and sorting have performance impacts that can slow down a query significantly. These steps are only performed if necessary.



In this illustration, the smaller table, table A, is identified by the optimizer during the join. Values are compared from the smaller table to the larger table. The values in the join row that are matched include A3 and C4. This occurs several times in the larger table, table B.

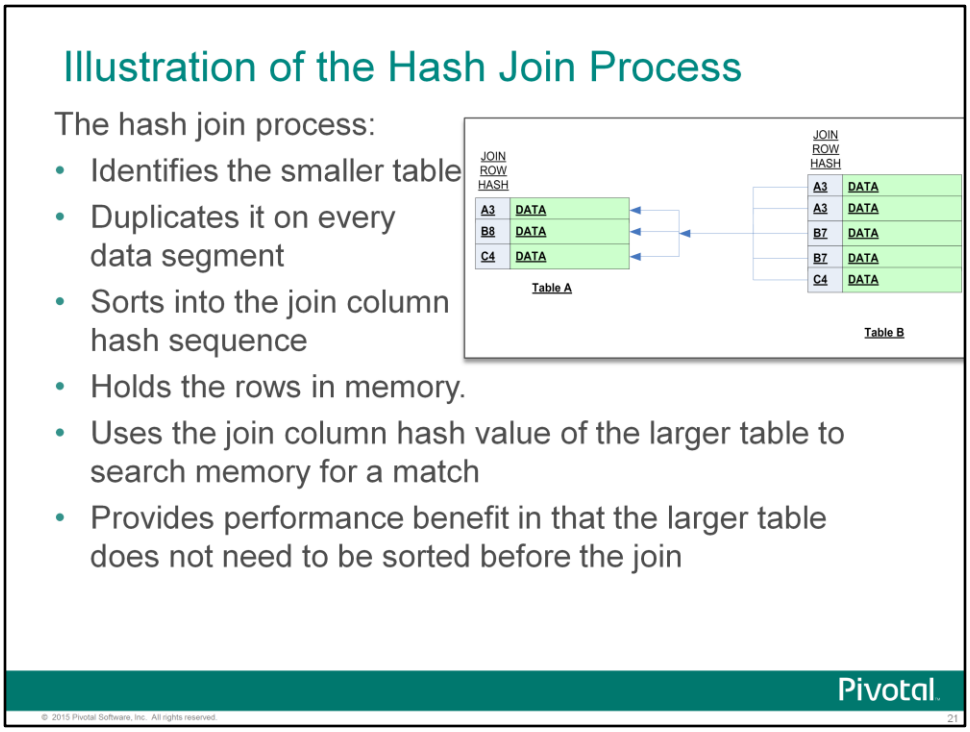


Merge join requires that both sides of the join have their qualifying rows in join columns hash sequence. In addition, if the join columns are not the primary index columns, some redistribution or duplication of rows precedes the sort.

In a Hash Join, the smaller table is sorted into join column hash sequence and then duplicated on all data segments. The larger table is then processed one row at a time. For those rows that qualify for joining the row hash value of the join columns is used to do a binary search through the smaller table for a match.

The optimizer can choose this join plan when qualifying rows of the smaller table can be held segment memory resident – the whole row set held in the each segment's memory.

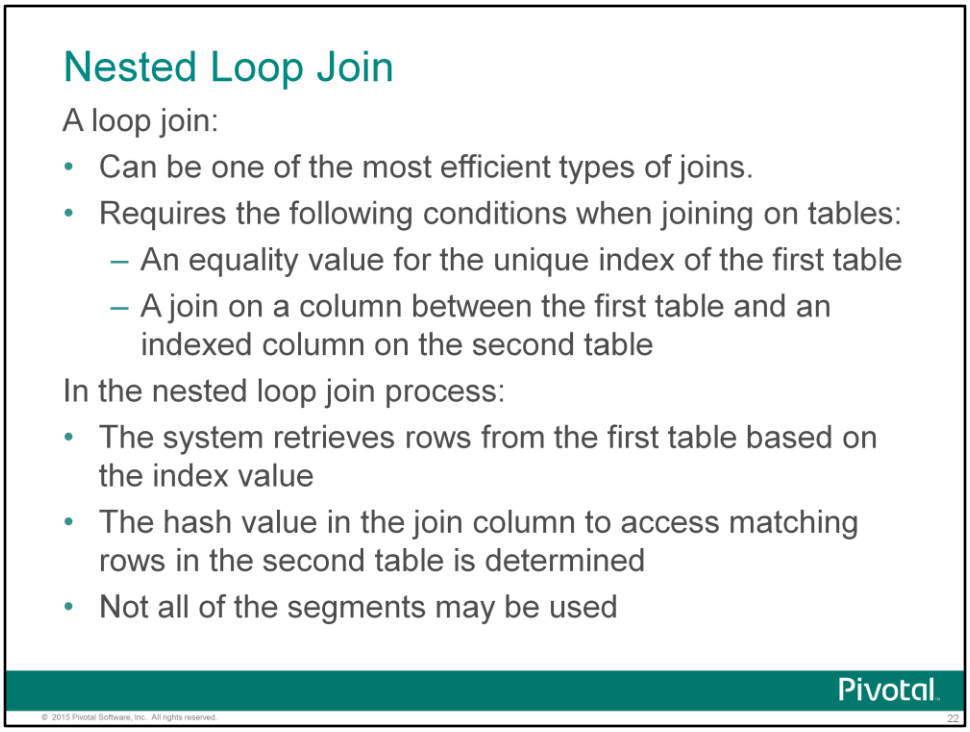
Use `ANALYZE` on both tables to help guide the optimizer.



The hash join process:

- Identifies the smaller table, in this example, A.
- Duplicates the smaller table on every data segment.
- Sorts the rows using the join column hash sequence
- Holds the rows in memory.
- Uses the join column hash value of the larger table to search memory for a match

A performance benefit of this join can result from the fact that there is no need for sorting larger join tables before performing the join.

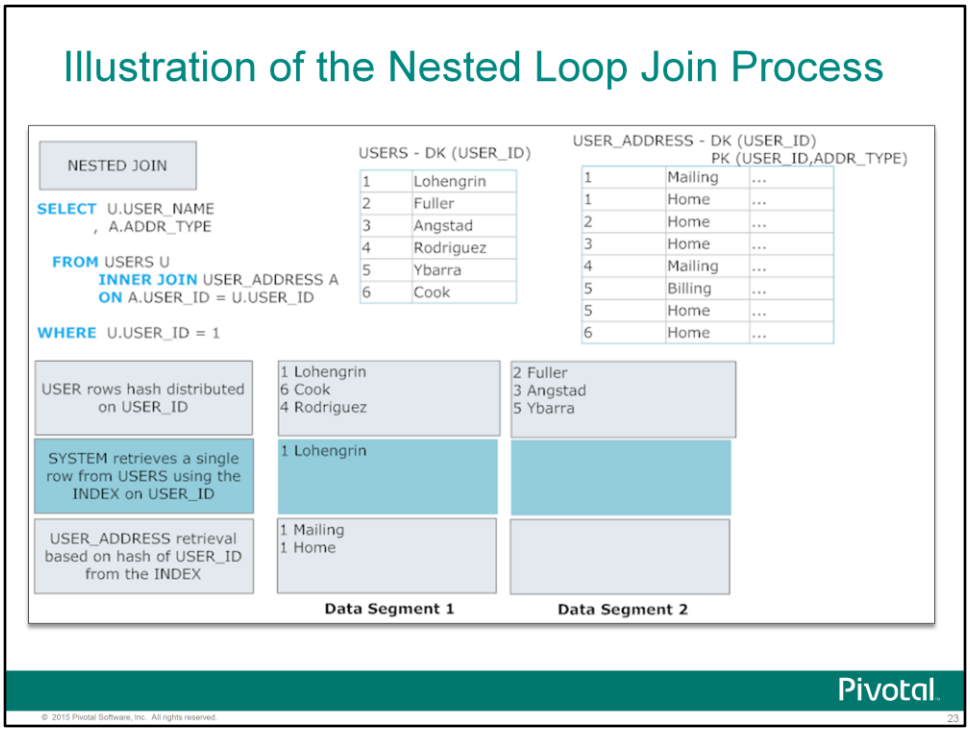


A loop join can be one of the most efficient types of joins. For a nested loop join to be performed between two tables, the following two conditions must occur:

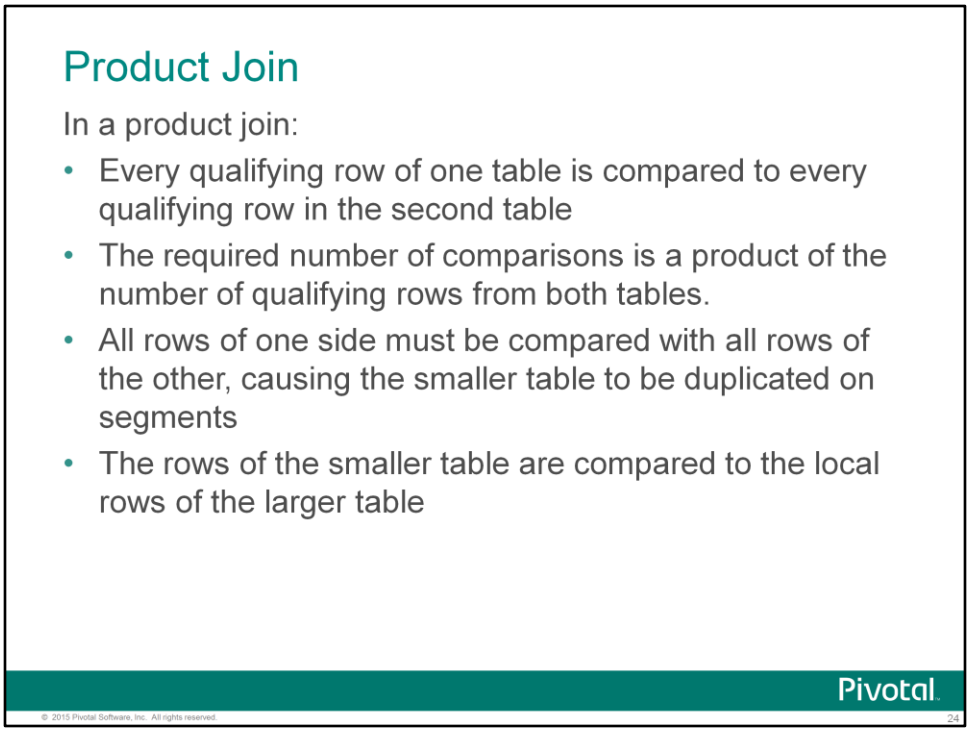
- An equality value for the unique index of the first table. This is retrieved from a single row, or a small number of rows, such as from an IN expression.
- A join on a column between the first table and an indexed column the second table.

During the nested loop join process:

- The system retrieves rows from the first table based on the index value. It then determines the hash value in the join column to access matching rows in the second table.
- Nested loop joins are the only types of Join that do not always use all of the segments during the join process.

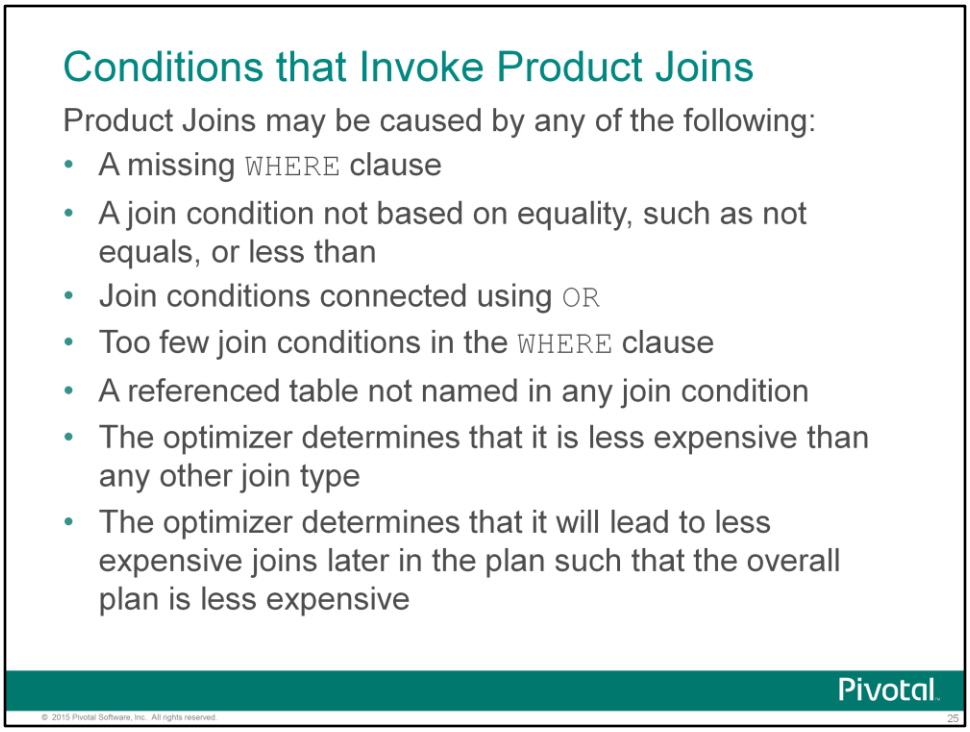


The illustration shows the equality value is chosen for the first and smaller table, `USERS`. The system then retrieves a single row from `USERS` and joins on an indexed column in the second table, `USER_ADDRESS`.



In a product join, every row in one table will be matched against every row in another table. So if you have 100 rows in one table and 1000 rows in the second table, you'll have 100,000 rows in the end.

The smaller table is always broadcast to the segments. With regards to resources, this can be dangerous. If you have 1 million rows being broadcast, you'll need appropriate storage for holding that amount of data on each segment.

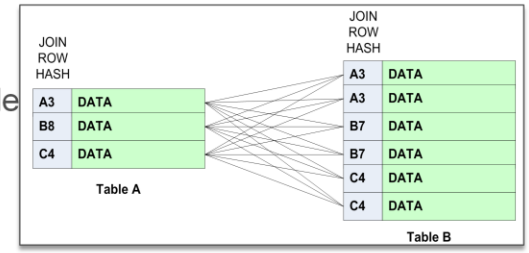


In general, product joins are considered to be bad. It is rare when the optimizer chooses product joins over other joins. The optimizer may choose a join when the product join is more expensive but will lead to less expensive joins later in the plan. This choice means that the overall plan is less expensive.

Illustration of the Product Join Process

The product join process:

- Identifies the smaller table
- Duplicates the smaller table in temporary space or memory on all data segments
- Joins each row of the smaller table to every row in the larger table
- Calculates the number of comparisons as the number of qualified rows in the first table to the number of qualified rows in the second table
- Can have costly comparisons when there are more rows than segment memory can hold at one time



Pivotal

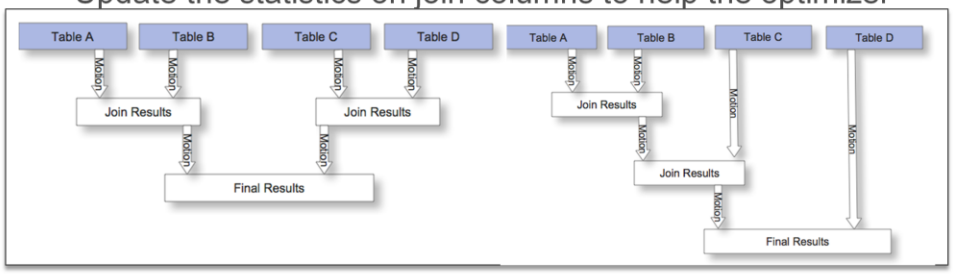
26

In the product join process, the smaller row is duplicated to all data segments and the rows of the smaller table are joined to every row in the larger table. A product join can become quite costly over the course of the process, especially if there are more rows than segment memory can hold at any one time.

N-Tables

When joining tables:

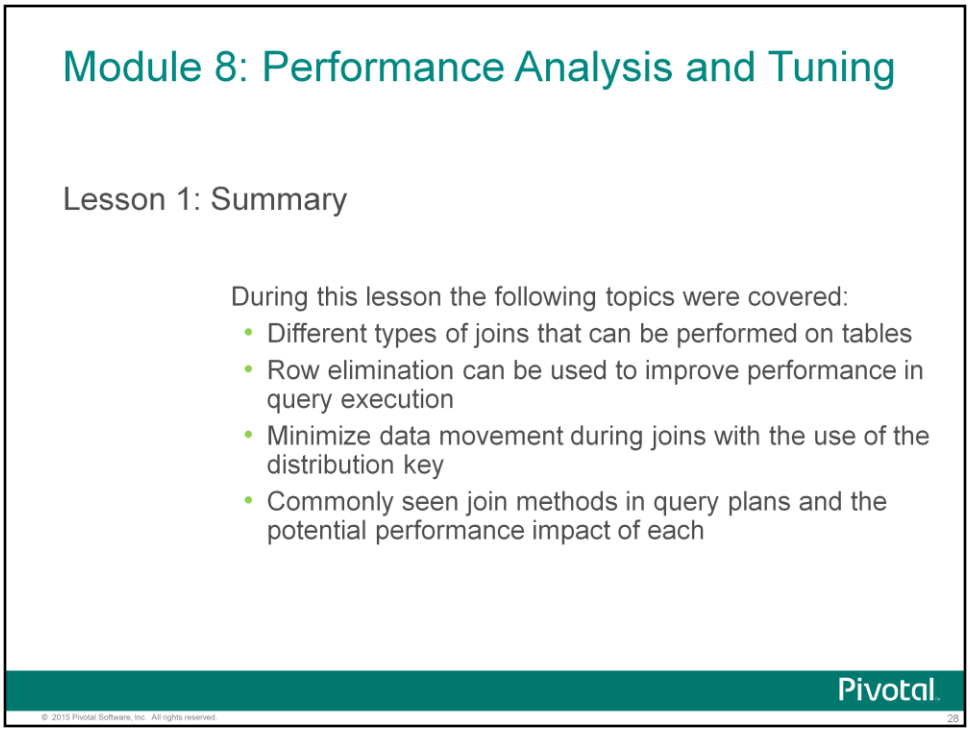
- All *n*-table joins are reduced to a series of two-table joins
- The optimizer attempts to determine the best join order
- The query engine can only work on two tables at a time
- The results of the previous join operation are applied to another table or another join results
- Update the statistics on join columns to help the optimizer



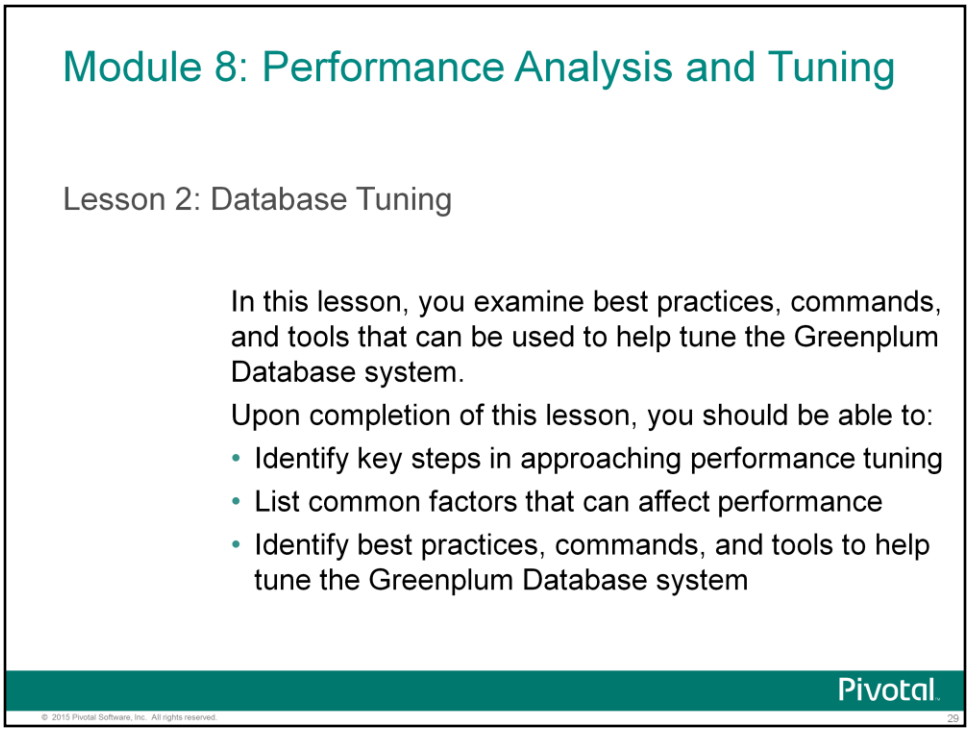
Pivotal

27

Every join occurs two-tables at a time. It can join using the first method depicted on the slide, or using the second method on the slide. The optimizer uses the statistics retrieved from `ANALYZE` to determine which to use.



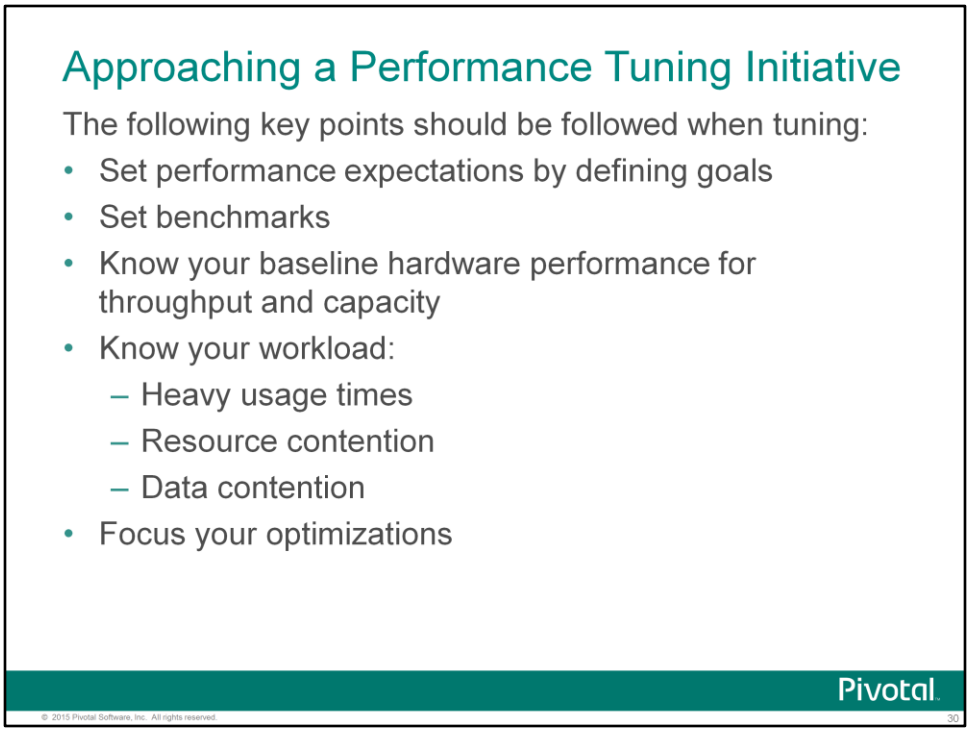
This lesson covered the various types of joins that can be performed on tables, how to minimize data movement during joins, the performance impact of various joins depending on data and table definition, and commonly seen join methods used in query plans once the optimizer has defined the best plan for your query.



From time to time, you may experience unsatisfactory performance in your environment. You should be aware of the factors that affect performance as well as how to address them.

In this lesson, you:

- Identify steps in approaching performance tuning.
- Identify common factors that can affect performance in the Greenplum Database system.
- Identify best practices, commands, and tools to help you tune your environment.



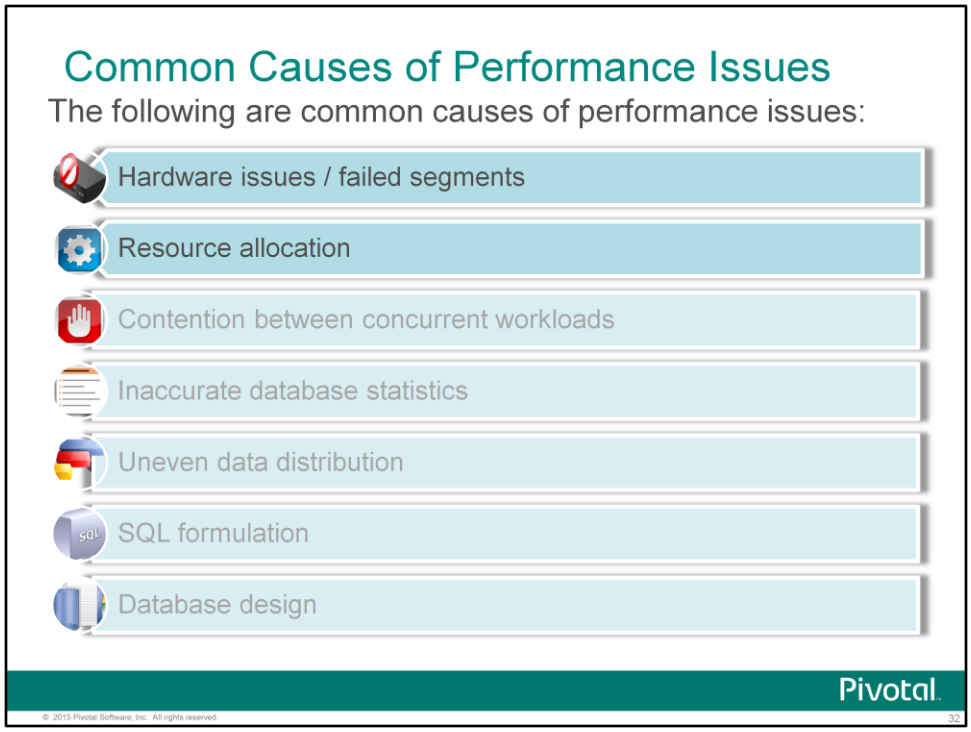
Performance is the rate at which the database management system (DBMS) supplies information to those requesting it.

When approaching performance and tuning:

- **Set expectations** – Without setting an acceptable threshold for database performance, you will be in a position where you define unattainable goals. Determine what is considered *good performance* in your particular environment. This could include setting an expected query response time or defining throughput goals.
- **Benchmark** – To maintain good performance or improve performance issues, you must know the capabilities of your DBMS on a defined workload. A benchmark is a predefined workload that produces a known result set, which can then be used for comparison purposes. Periodically running the same benchmark tests can help identify system-related performance degradation over time. Benchmarks can also be used as a comparison to other workloads in an effort to identify queries or applications in need of optimization.

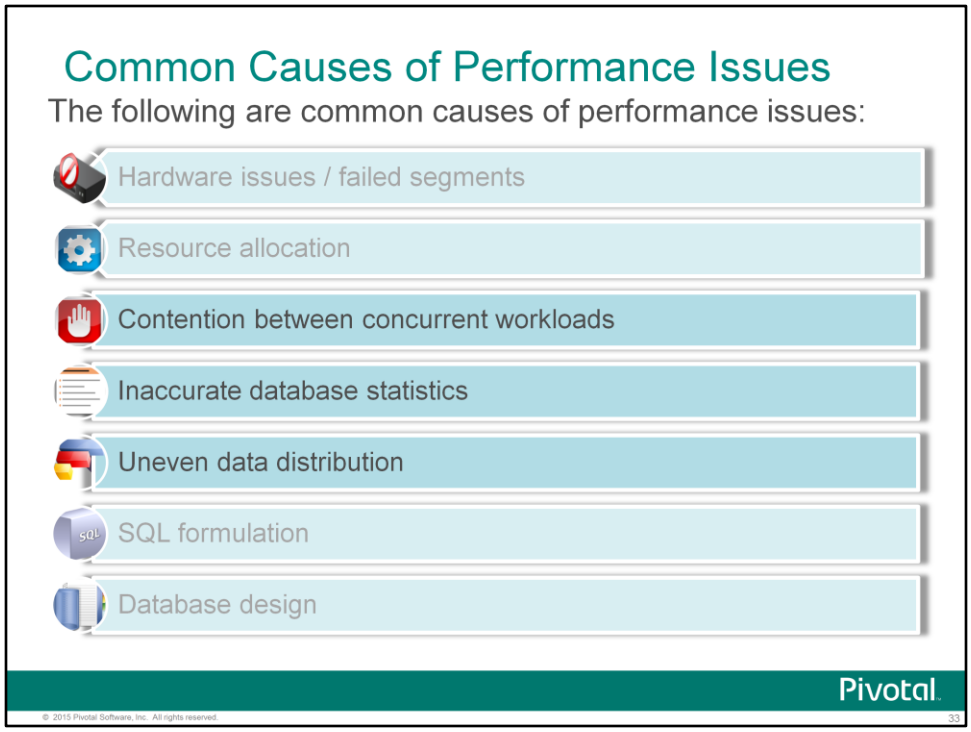
Approaching a Performance Tuning Initiative (Continued)

- Hardware – Database performance relies heavily on disk I/O and memory usage. Knowing the baseline performance of the hardware on which your DBMS is deployed is essential is setting performance expectations. Performance of hardware components such as CPUs, hard disks, disk controllers, RAM, and network interfaces, as well as the interaction of these resources, can have a profound effect on how fast your database performs. The majority of database performance problems are caused not by the database itself, but by the underlying systems on which the database is running. I/O bottlenecks, memory problems, and network problems can significantly degrade database performance. Therefore, it is important to know the baseline capabilities of your hardware and operating system (OS). This will help you to identify and troubleshoot hardware-related problems before undertaking database-level or query-level tuning initiatives.
  A system's throughput defines its overall capability to process data. DBMS throughput can be measured in queries per second, transactions per second, or average response times. DBMS throughput is closely related to the processing capacity of the underlying systems (disk I/O, CPU speed, memory bandwidth, and so on), so it is important to know the throughput capacity of your hardware when setting DBMS throughput goals.
- Workload – Your workload equals the total demand from the DBMS. The total workload is a combination of ad-hoc user queries, applications, batch jobs, transactions, and system commands directed through the DBMS at any given time. Workload, or demand, can change over time. For example, it may increase when month-end reports need to be run, or decrease on weekends when most users are out of the office. Workload is a major influence on database performance. Knowing your workload and peak demand times will help you plan for the most efficient use of your system resources, and enable the largest possible workload to be processed.
  Contention is the condition in which two or more components of the workload are attempting to use the system in a conflicting way. For example, trying to update the same piece of data at the same time, or running multiple large workloads at once that compete with each other for system resources causes content. As contention increases, throughput decreases.
- Optimization – Once you have determined what your system is capable of and what kind of performance you expect, you can focus your tuning initiatives on those areas and queries that fall short of your performance goals. Things like SQL formulation, database parameters, table design, data distribution, and so on should be altered only when performance is not acceptable.

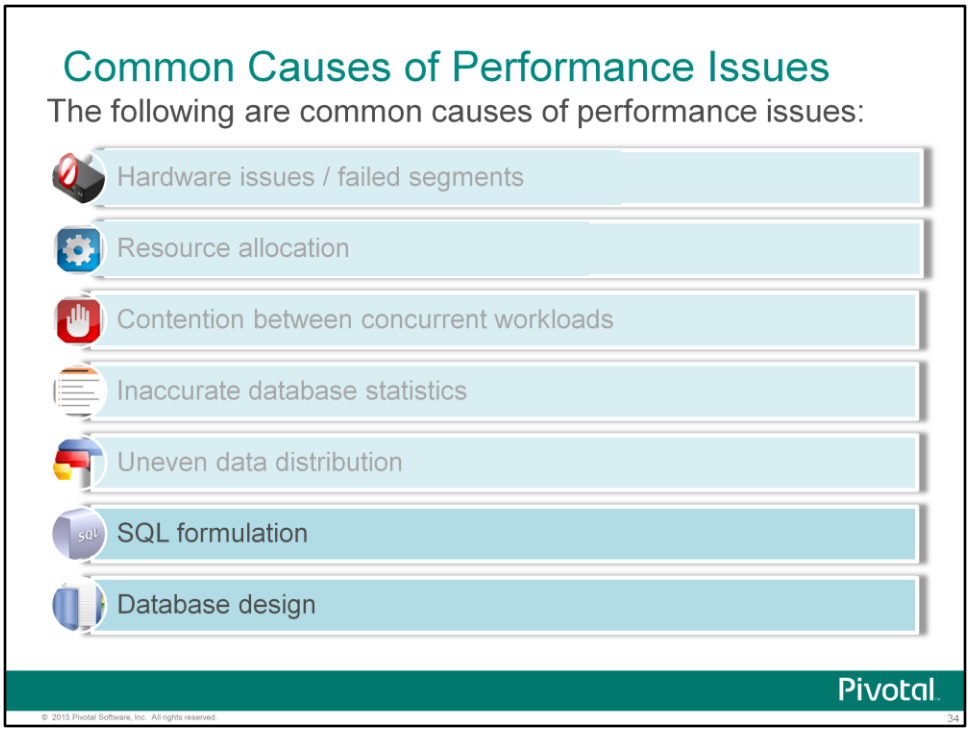


The following are common performance issues:

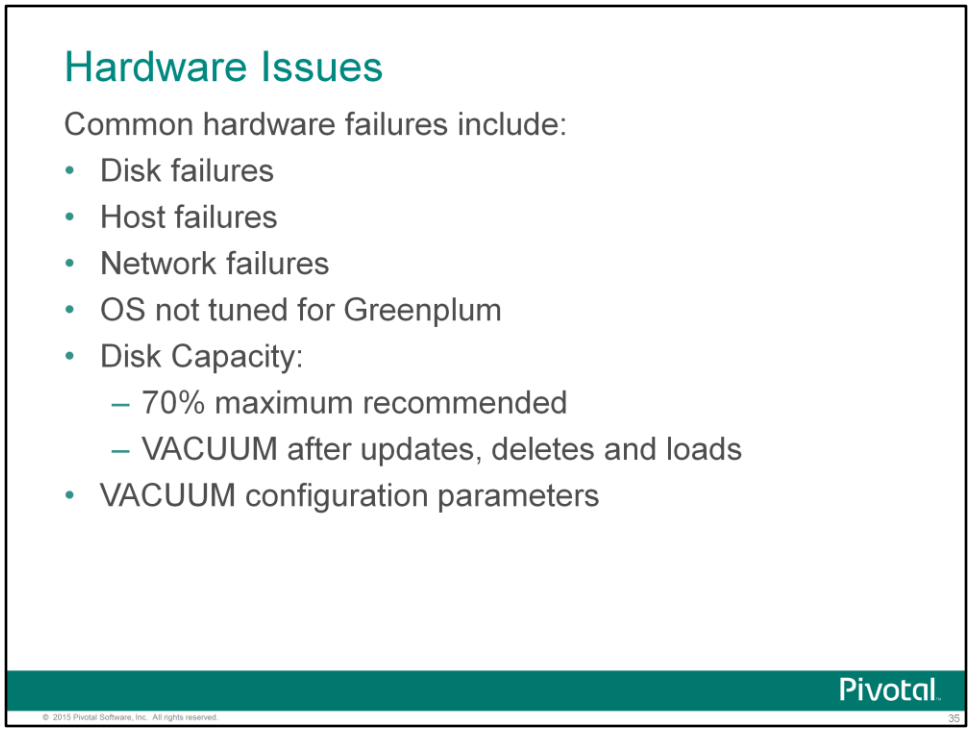
- **Hardware Problems** – As with any database system, the performance of Greenplum Database is dependant upon the hardware and IT infrastructure on which it is running. Performance will be as fast as the slowest host in the Greenplum array. Problems with CPU utilization, memory management, I/O processing, or network load will affect performance. Many hardware failures will cause the segment instances running on that hardware to fail. If mirrors are enabled, this can mean other segment hosts are doing double-duty.
- **Resource Allocation** - A database system has a limited capacity of CPU, memory, and disk I/O resources. When multiple workloads compete for access to these resources, database performance suffers. Resource allocation, also referred to as workload management, can be used to maximize system throughput while still meeting varied business requirements. Greenplum Database workload management limits the number of active queries in the system at any given time in order to avoid exhausting system resources. This is accomplished by creating role-based resource queues. A resource queue has attributes that limit the size and/or total number of queries that can be executed by the users (or roles) in that queue. By assigning all of your database roles to the appropriate resource queue, administrators can control concurrent user queries and prevent the system from being overloaded.



- **Contention Between Concurrent Workloads** - Contention arises when two or more users or workloads try to use the system in a conflicting way. For example, if two transactions are trying to update the same table at once. A transaction seeking either a table-level or row-level lock will wait indefinitely for conflicting locks to be released. This means it is a bad idea for applications to hold transactions open for long periods of time (e.g., while waiting for user input). Are indexes being used correctly?
- **Inaccurate Database Statistics** - Greenplum Database uses a cost-based query planner that relies on database statistics. Accurate statistics allow the query planner to better estimate the number of rows retrieved by a query in order to choose the most efficient query plan. Without database statistics, the query planner can not estimate how many records might be returned, and therefore cannot assume it has sufficient memory to perform certain operations such as aggregations. In this case, the planner always takes the safe route and does aggregations by reading/writing from disk, which is significantly slower than doing them in memory. The ANALYZE command collects statistics about the database needed by the query planner.
- **Uneven Data Distribution** - When you create a table in Greenplum Database, it is important to declare a distribution key that allows for even data distribution across all segments in the system. Because the segments work on a query in parallel, Greenplum Database will always be as fast as the slowest segment. If the data is unbalanced, the segments that have more data will return their results slower.

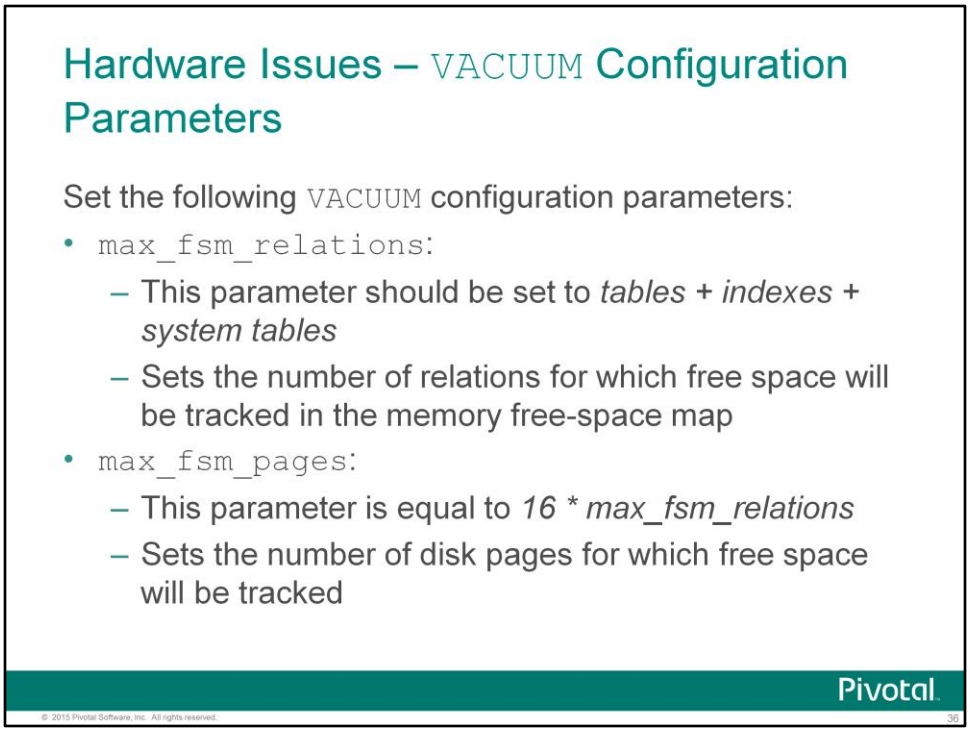


- **Poor SQL Formulation** - Many application developers simply write SQL statements off the top of their heads without giving much thought to designing them for efficient processing. Statements that are not designed can introduce enormous performance issues. For example, are you retrieving more data than you actually need to produce the desired result set? Are you using expensive operations that can be written in a more efficient way?
- **Poor Database Design** - Many performance issues are caused by poor database design. Examine your database design and ask yourself the following: Does the schema reflect the way the data is accessed? Can larger tables be broken down into smaller tables or partitions? Are you using the smallest data type possible to store column values? Are columns used to join tables of the same data type? Are indexes being used correctly?



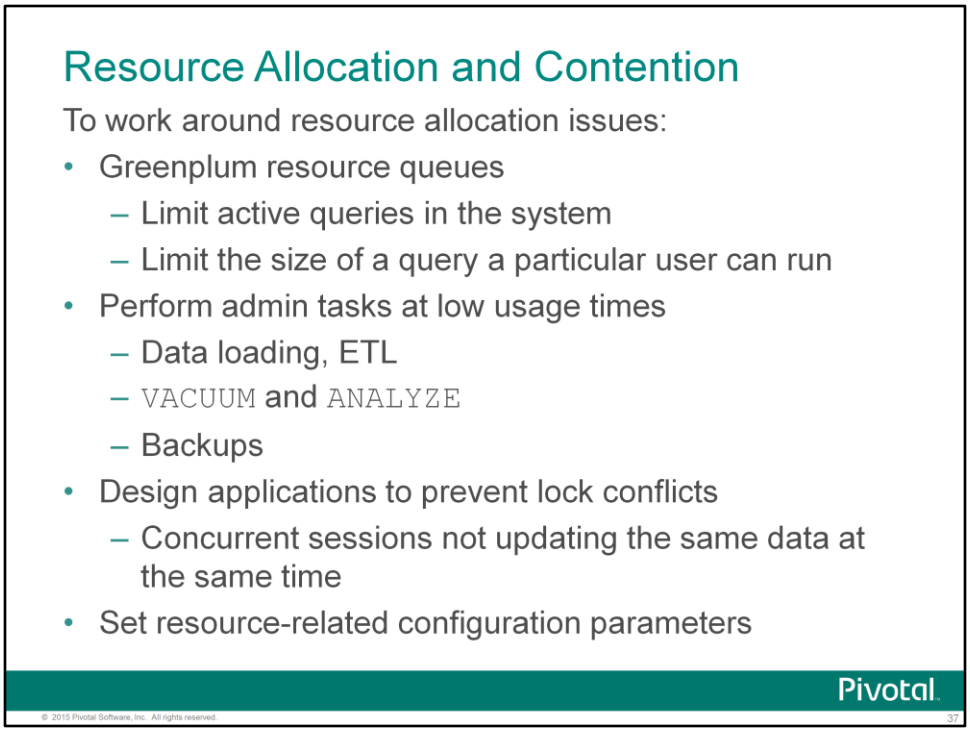
Hardware failures can cause segments to fail or degrade in performance, including:

- **Disk failure** – Although a single disk failure should not dramatically effect database performance if you are using RAID – there is some impact caused by disk resynching consuming resources on the host with failed disks.
- **Host failure** – When a host is offline (power failure, etc) the segments on that host are out of operation. This means that other hosts in the array are doing double duty as they are running both the primary segments and a number of mirrors. If mirrors are not enabled – service is interrupted. There is a temporary interruption of service to recover failed segments.
- **Network failure** – Failure of a NIC, switch, or DNS server can bring down segments.
- Capacity issues – Disk capacity on segment hosts should be at the maximum 70% capacity for optimal performance. To reclaim space after load or updates run VACUUM on a regular basis. Monitor capacity to make sure you do not run out of disk space on a segment host.
- **Not using `VACUUM`** – Because of the MVCC transaction concurrency model, data rows that are deleted or updated still occupy physical space on disk even though they are not visible to any new transactions. If you have a database with lots of updates and deletes, you will generate a lot of dead rows. Dead rows are held in what is called the free space map. The free space map must be sized large enough to cover the dead rows of all tables in your database. If not sized large enough, space occupied by dead rows that overflow the free space map cannot be reclaimed by a regular `VACUUM` command.



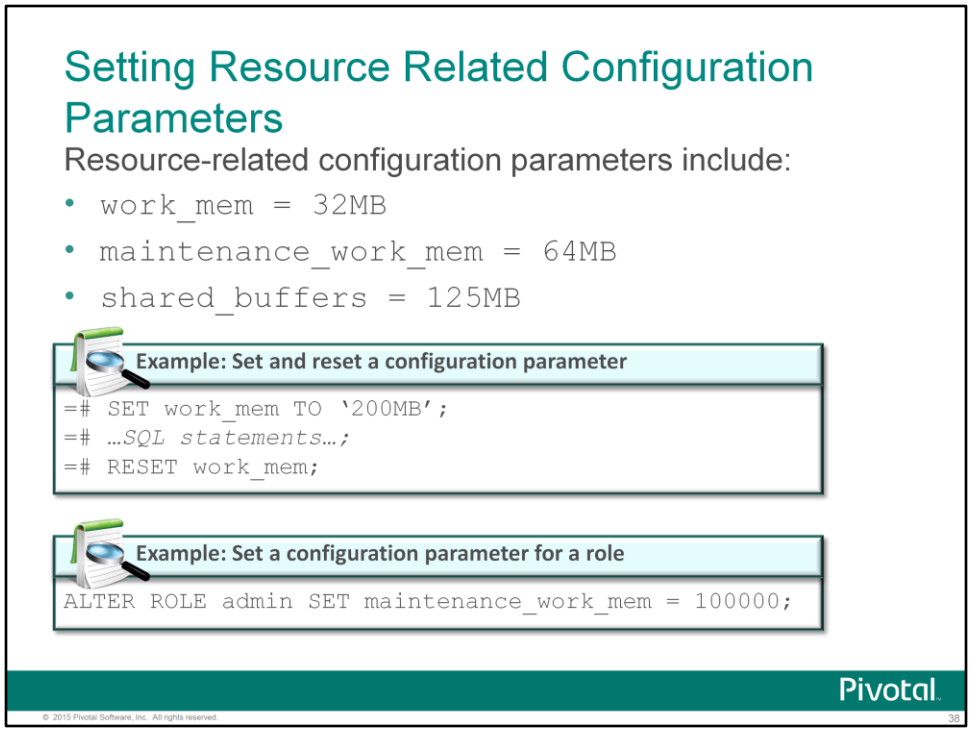
The parameters, `max_fsm_pages` together with `max_fsm_relations` control the size of the shared free space map, which tracks the locations of unused space in Greenplum Database. An undersized free space map may cause Greenplum Database to consume increasing amounts of disk space over time, because free space that is not in the map cannot be reused; instead Greenplum Database will request more disk space from the operating system when it needs to store new data.

Note that for the autovacuum daemon, `VACUUM` cannot run in a transaction. It creates and commits its own transactions as it goes so it can release locks as soon as possible. This is a bit tricky in Greenplum Database in keeping the segment databases in synch with each other. Because `VACUUM` runs asynchronously between the segment instances, it is not possible to protect tables on the segment by taking a table-level lock on the master. This is an issue with the autovacuum daemon as it acts locally on the segment without taking into account the state of the entire array.



To limit resource contention for both hardware and database resources requires some knowledge of your workload, users and peak usage times. To reduce contention you can:

- Create resource queues that limit the number of active queries and/or size of queries let into the system. Then you can assign all of your database roles to a queue.
- Do not compete with database users by performing admin tasks at peak usage times. Do admin operations and load after hours.
- Do not run workloads that try to update the same resources at the same time. Lock conflicts will minimize throughput for competing workloads.

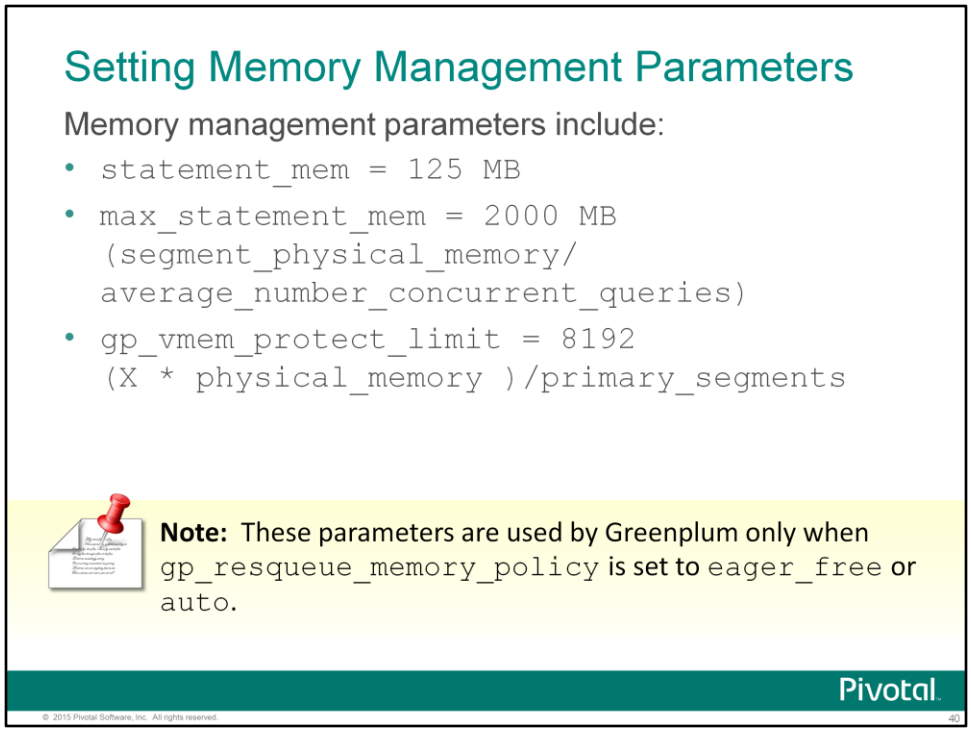


The following resource-related configuration parameters can be used for resource allocation and contention:

- The `work_mem` parameter specifies the amount of memory to be used by internal sort operations and hash tables before switching to temporary disk files. Note that for a complex query, several sort or hash operations might be running in parallel; each one will be allowed to use as much memory as this value specifies before it starts to put data into temporary files. Several running sessions could be doing such operations concurrently. Therefore, the total memory used could be many times the value of `work_mem`. It is necessary to keep this fact in mind when choosing the value.
  Sort operations are used for `ORDER BY`, `DISTINCT`, and merge joins. Hash tables are used in hash joins, hash-based aggregation, and hash-based processing of IN subqueries. Note that `work_mem` is only held for the duration of an operation, and then is immediately freed. A recommended approach is to increase `work_mem` on a per-session or per-query basis as needed.
- `maintenance_work_mem` – Specifies the maximum amount of memory to be used in maintenance operations, such as `VACUUM` or `CREATE INDEX`. Since only one of these operations can be executed at a time by a database session, and an installation normally does not have very many of them happening concurrently, it is safe to set this value significantly larger than `work_mem`. Larger settings may improve performance for vacuuming and for restoring database dumps.

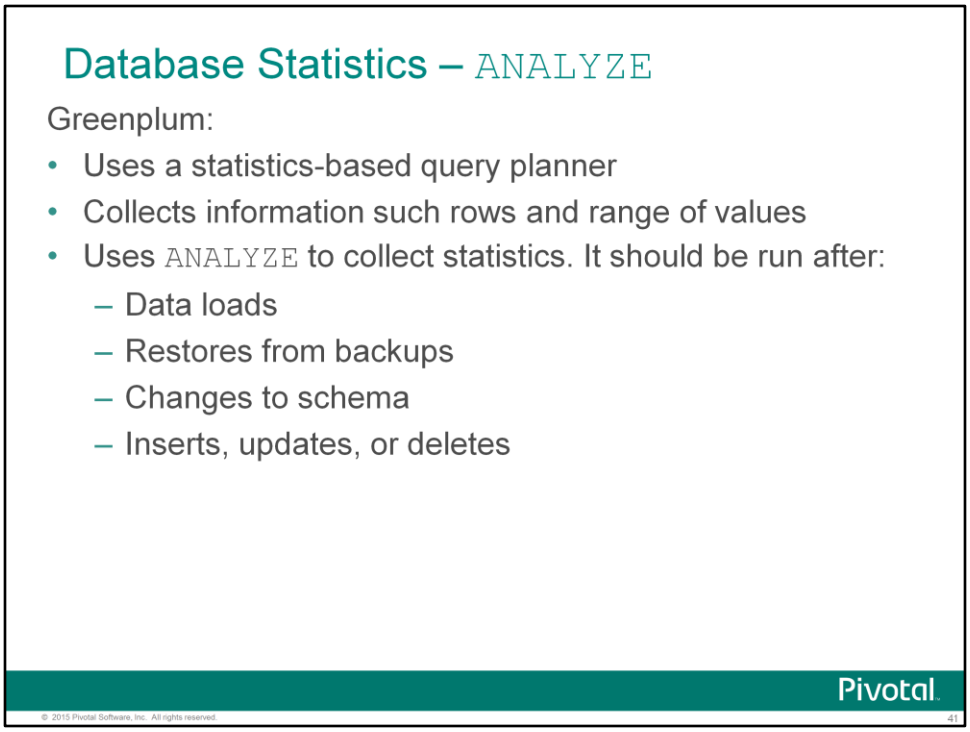
Setting Resource Related Configuration Parameters (Continued)

- `shared_buffers` – Sets the amount of memory a Greenplum server instance uses for shared memory buffers. This setting must be at least 128 kilobytes and at least 16 kilobytes times max_connections. Shared buffers define a block of memory that Greenplum Database will use to hold requests that are waiting for kernel buffer and CPU. Greenplum Database relies mostly on the OS to cache data files, therefore this parameter is set relatively low when compared to the total RAM available on a host.



The new workload management system, enabled by setting the configuration parameter, `gp_resqueue_memory_policy` to `eager_free` or `auto`, ignores the `work_mem` and `maintenance_work_mem`. The resource queues allow you to specify limitations on memory and CPU resources. However, you can set configuration parameters to ensure that queries have enough resources, despite the settings on the resource queues. Configuration parameters that can be used to affect resources outside of the resource queues include:

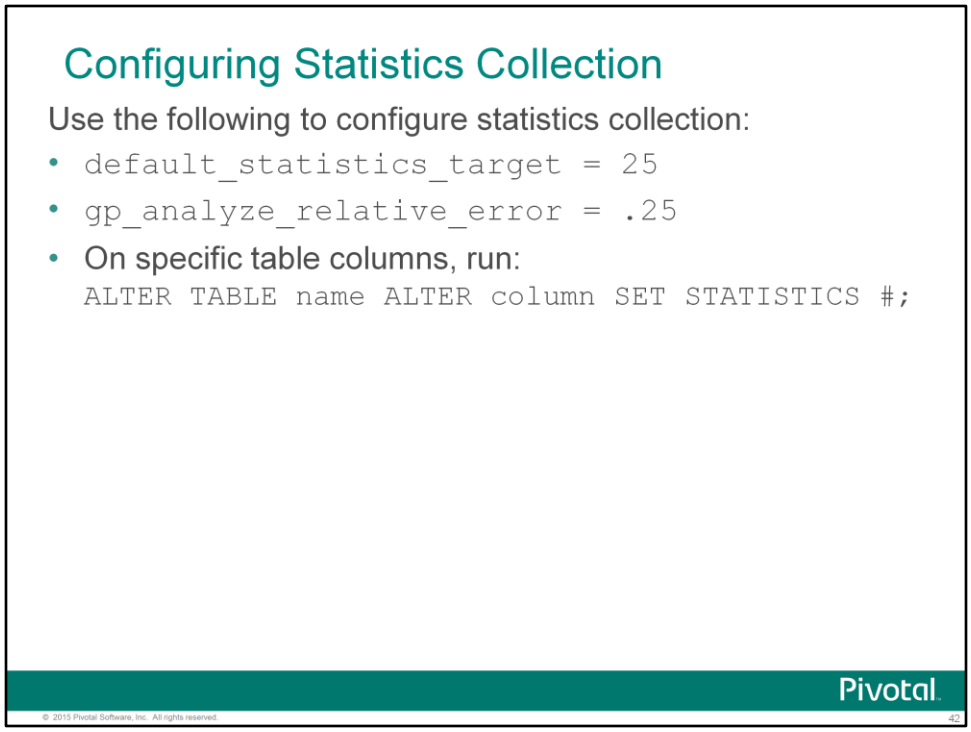
- **statement_mem** – The default memory allotment defined by the resource queue can be overridden on a per query basis by setting the `statement_mem` parameter. The query can be allocated memory up to the amount defined in the max_statement_mem parameter. These values should never exceed the physical memory of a segment host. Setting this parameter for a specific query prevents out of memory errors that could occur on a segment for that specific query.
- **max_statement_mem** – The parameter defines the maximum amount of memory that can be allocated to a query. This value is used both by the statement_mem configuration parameter and the resource queue memory limits as a ceiling for the amount of memory that can be allocated. This parameter can also be used to avoid out of memory errors if the `statement_mem` or memory limits in the resource queue are set too high.
- **`gp_vmem_protect_limit`** – This parameter sets the maximum amount of memory that all postgres processes of an active segment instance can consume.



Greenplum uses a cost-based query planner that relies on database statistics. Statistics such as number of rows and the range of values in a column are needed by the query planner for it to choose the most optimal plans.

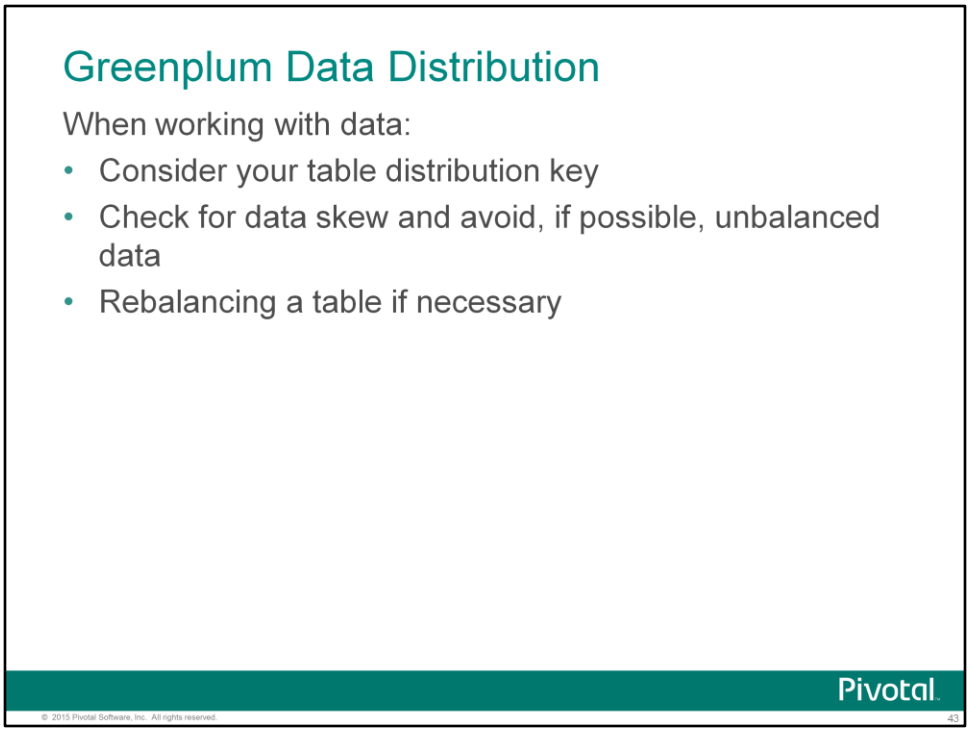
The `ANALYZE` command collects statistics about the database needed by the query planner. It is a good idea run `ANALYZE` after any changes to the database are made, including:

- Performing data loads
- Restore data from backups
- Making changes to the schema
- Changing data with insertions, updates, or deletions



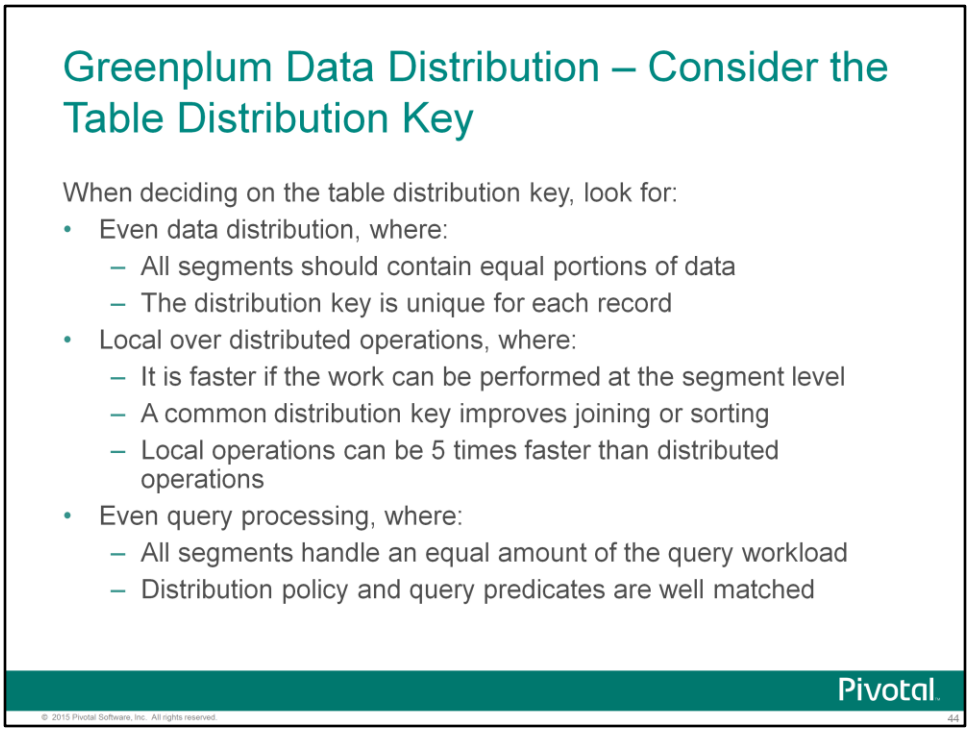
The following can be used to configure statistics collection for the query planner:

- **The `default_statistics_target` parameter** – Sets the default statistics target for table columns that have not had a column-specific target set using `ALTER TABLE SET STATISTICS`. Larger values increase the time needed to perform `ANALYZE`, but may improve the quality of the query planner's estimates. Specifically, this sets the maximum number of most common values and histogram bounds, or the number of value groups of approximately equal population, that are sampled. Most queries retrieve only a fraction of the rows in a table, due to having `WHERE` clauses that restrict the rows to be examined. The planner makes an estimate of the selectivity of `WHERE` clauses, or the fraction of rows that match each condition in the `WHERE` clause. A higher value, especially for columns with irregular data patterns, may allow for more accurate query plans.
- **The `gp_analyze_relative_error` parameter** – Sets the estimated acceptable error in the cardinality of the table. A value of 0.5 is equivalent to an acceptable error of 50%, the default value used in PostgreSQL. If the statistics collected during `ANALYZE` are not producing good estimates of cardinality for a particular table attribute, decreasing the relative error fraction, or accepting less error, tells the system to sample more rows to determine the number of distinct non-null data values in a column (as stored in the `n_distinct` column of the `pg_stats` table).
- **`ALTER TABLE SET STATISTICS`** – This is similar to setting `default_statistics_target` higher for a particular table column. This is recommended for columns frequently used in query predicates and joins.



There are several key points to consider when working with your data:

- Consider what your table distribution key should be. In a Greenplum Database table, one or more columns are used as the distribution key, meaning those columns are used by the hashing algorithm to divide the data amongst all of the segments. The benefit of data distribution is to get your data as flat as possible.
- Check for data skew. Your data may become unbalanced, causing one segment to work harder than another. This ties into the table distribution key selection.
- If necessary, rebalance the table to achieve a more balanced solution.

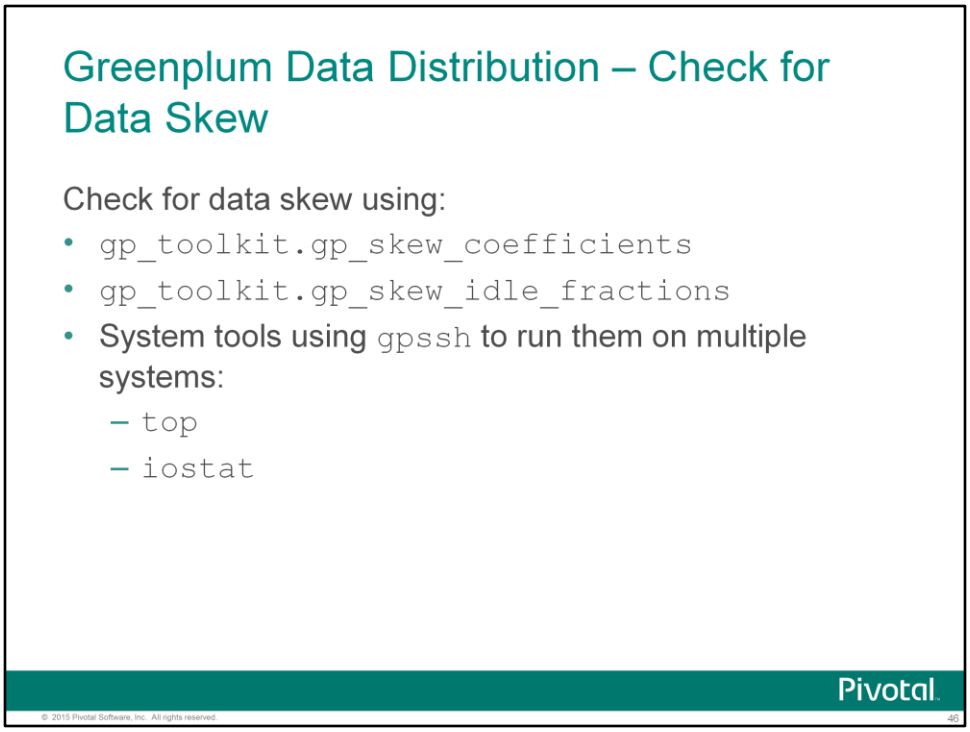


The following considerations should be taken into account when declaring a distribution key for a table (listed in order of importance):

- **Even data distribution** — For the best possible performance, all of the segments should contain equal portions of data. If the data is unbalanced or skewed, then the segments with more data will have to work harder to perform their portion of the query processing. To ensure an even distribution of data, you want to choose a distribution key that is unique for each record, such as the primary key.
- **Local and distributed operations** — During query processing, it is faster if the work associated with join and aggregation operations can be done locally at the segment-level rather than at the system-level (distributing tuples amongst the segments). When tables share a common distribution key in Greenplum Database, joining or sorting on their shared distribution key columns will result in the most efficient query processing, as the majority of the work is done locally at the segment-level. Local operations are approximately 5 times faster that distributed operations.

Greenplum Data Distribution – Consider the Table Distribution Key (Continued)

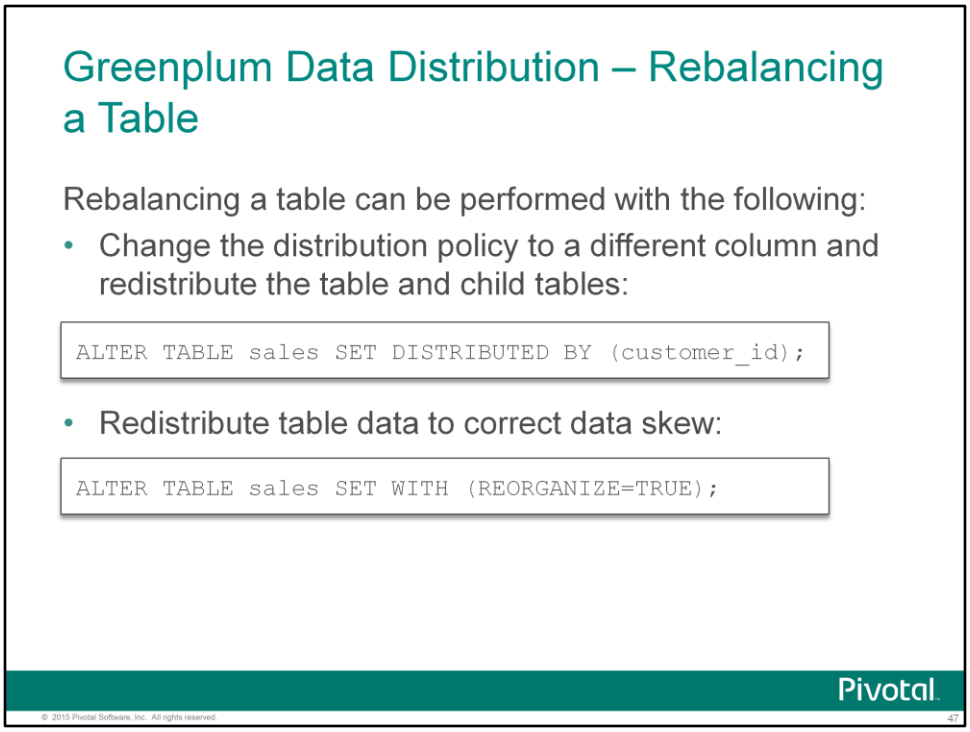
- **Even query processing** — When a query is being processed, you want all of the segments to handle an equal amount of the query workload to get the best possible performance. In some cases, query processing workload can be skewed if the table's data distribution policy and the query predicates are not well matched. For example, suppose you have a table of sales transactions. The table is distributed based on a column that contains corporate names as values. The hashing algorithm distributes the data based on the values of the distribution key, so if a predicate in a query references a single value from the distribution key, the work in the query will run on only one segment. This may be a viable distribution policy if your query predicates tend to select data on a criteria other than corporation name. However, for queries that do use corporation name in their predicates, you can potentially have just one segment instance handling all of the query workload.



One indication of uneven table distribution is when one segment is taking significantly longer to complete its portion of a query. You can view individual segment host performance during query processing by accessing:

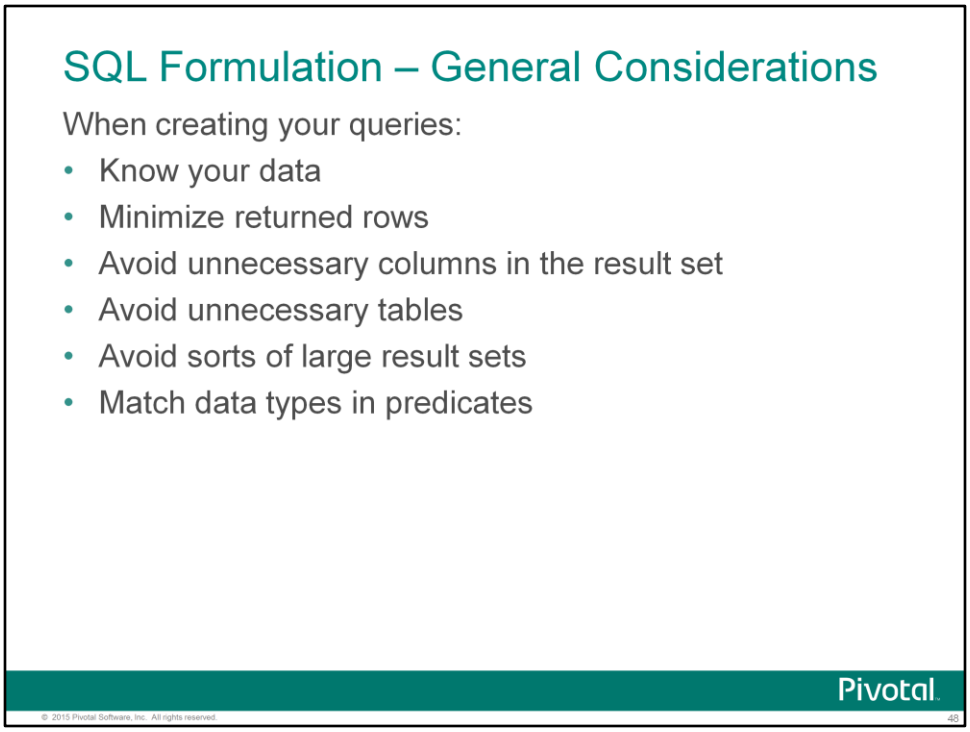
- **`gp_toolkit.gp_skew_coefficients` –** This view shows data distribution skew by calculating the coefficient of variation (CV) for the data stored on each segment. This view is accessible to all users, however non-superusers will only be able to see tables that they have permission to access.
  The column, skccoeff, shows the coefficient of variation (CV). This is calculated as the standard deviation divided by the average. It takes into account both the average and variability around the average of a data series. The lower the value, the better. Higher values indicate greater data skew.
- **`gp_toolkit.gp_skew_idle_fractions` –** This view shows data distribution skew by calculating the percentage of the system that is idle during a table scan, which is an indicator of processing data skew. This view is accessible to all users, however non-superusers will only be able to see tables that they have permission to access.
  The column, siffraction, shows the percentage of the system that is idle during a table scan. This is an indicator of uneven data distribution or query processing skew. For example, a value of 0.1 indicates 10% skew, a value of 0.5 indicates 50% skew, and so on. Tables that have more than 10% skew should have their distribution policies evaluated.
- System tools such as `top` and `iostat` can be run on multiple systems with `gpssh`.

The `gp_toolkit` schema is an administrative schema that can be used to query system catalogs, log files, and the operating system for system status information.



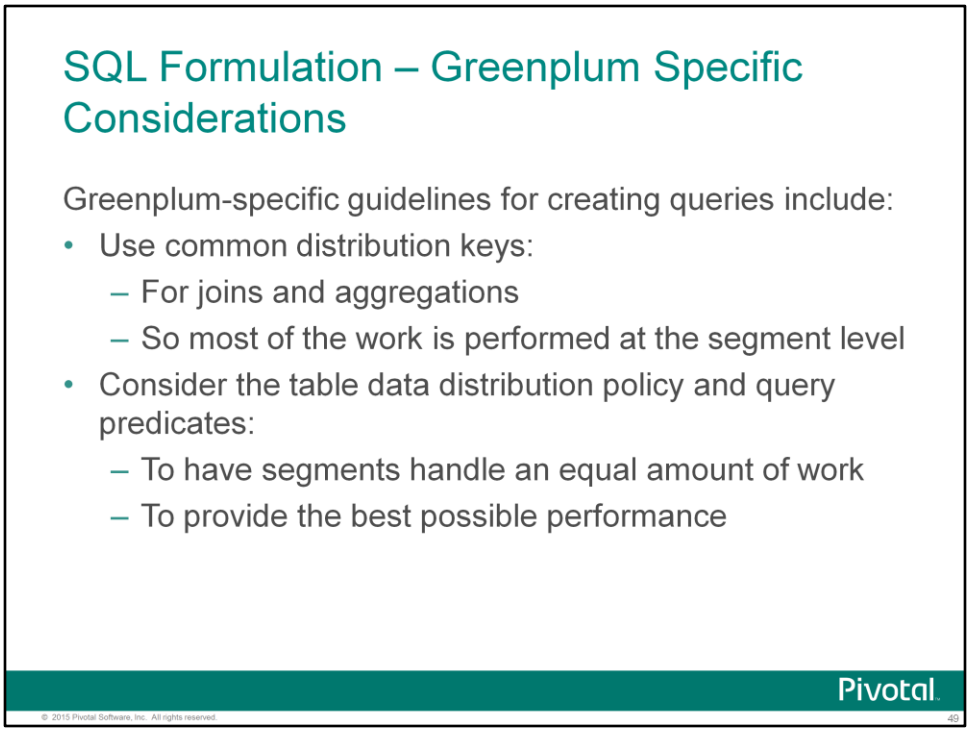
You can rebalance a table to change the distribution policy that previously existed on the table. If you need to change the distribution policy, use the `ALTER TABLE` SQL command and the `SET DISTRIBUTED BY` clause. This automatically redistributes the data.

To redistribute the data for tables with a random policy or to simply rebalance the table, use `REORGANIZE=TRUE` as part of the `ALTER TABLE` SQL command.



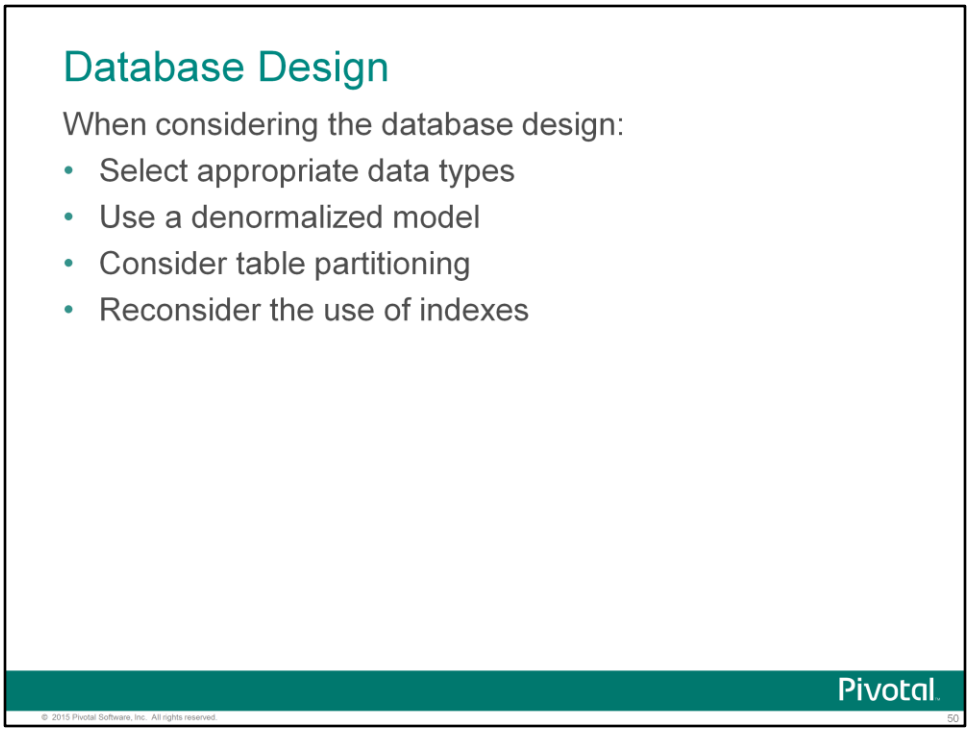
When creating queries:

- **Know your data** - When writing the SQL statement, you should have some idea of how many rows are in each table, how selective your WHERE predicates are, and how many rows you expect in the result set. The larger the number of rows involved, the more time you should spend thinking about the best way to write the SQL statement.
- **Minimize returned rows** – The fewer the rows in the result set, the more efficiently the query will run. Verify your query only fetches the data you need. If returning a large number of rows is unavoidable, consider using cursors to fetch a subset of rows at a time. Avoid multiple `DISTINCT` aggregates if possible.
- **Avoid unnecessary columns in the result set** – The wider the data in each row in the result set, the more disk space and memory that is required for intermediate operations such as sorts to hold the result set.
- **Avoid unnecessary tables** – The fewer the tables, the more efficient the query.
- **Avoid sorts of large result sets if possible** – Sorts are expensive, especially when the rows being sorted will not fit in memory. Try to limit the amount of data that needs to be sorted.
- **Match data types in predicates** – When a query predicate compares two column values as is done with joins, it is important for the data types to match. When the data types do not match, the DBMS must convert one of them before performing the comparison, and while the work to do this is relatively small, it can add up when it has to be done for a large number of rows.



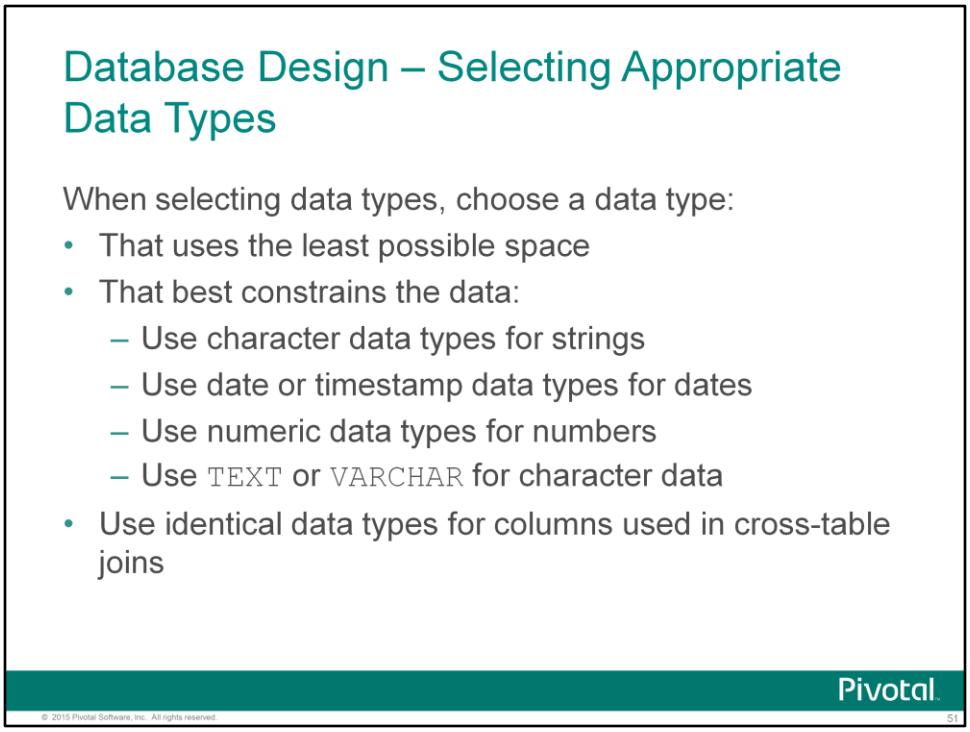
Greenplum-specific considerations for creating queries include:

- Using common distribution key columns for joins and aggregations when possible. When tables share a common distribution key in Greenplum Database, joining or sorting on their shared distribution key columns will result in the most efficient query processing. This allows the majority of the work to be done at the segment-level, which is significantly faster than redistributing data across the segments at the system-level.
- Considering the table data distribution policy when determining query predicates. When a query is being processed, you want all of the segments to handle an equal amount of the query workload to get the best possible performance. In some cases, query processing workload can be skewed if the table's data distribution policy and the query predicates are not well matched. For example, suppose you have a table of sales transactions. The table is distributed based on a column that contains corporate names as values. The hashing algorithm distributes the data based on the values of the distribution key, so if a predicate in a query references a single value from the distribution key, the work in the query will run on only one segment. This may be a viable distribution policy if your query predicates tend to select data on a criteria other than corporation name. However, for queries that do use corporation name in their predicates, you can potentially have just one segment instance handling all of the query workload.



When designing the database, consider the following:

- **Data Types** – The data type of a column determines the types of data values that column can contain.
- **Denormalization** – In relational database theory, normalization is the process of restructuring the logical data model of a database to eliminate redundancy and improve data organization. A denormalized scheme, while introducing redundancy, can improve performance.
- **Table partitioning** – This addresses the problem of large fact tables by dividing the table into more manageable pieces. Partition tables can have a positive effect on query performance.
- **Indexing** – In distributed databases such as Greenplum Database, indexes should be used sparingly, as they do not offer the same level of performance that may be seen in other traditional databases.



The data type of a column determines the types of data values that column can contain. As a general rule, you should choose the data type that:

- Uses the least possible space, yet can still accommodate your data
- That best constrains the data in that column.

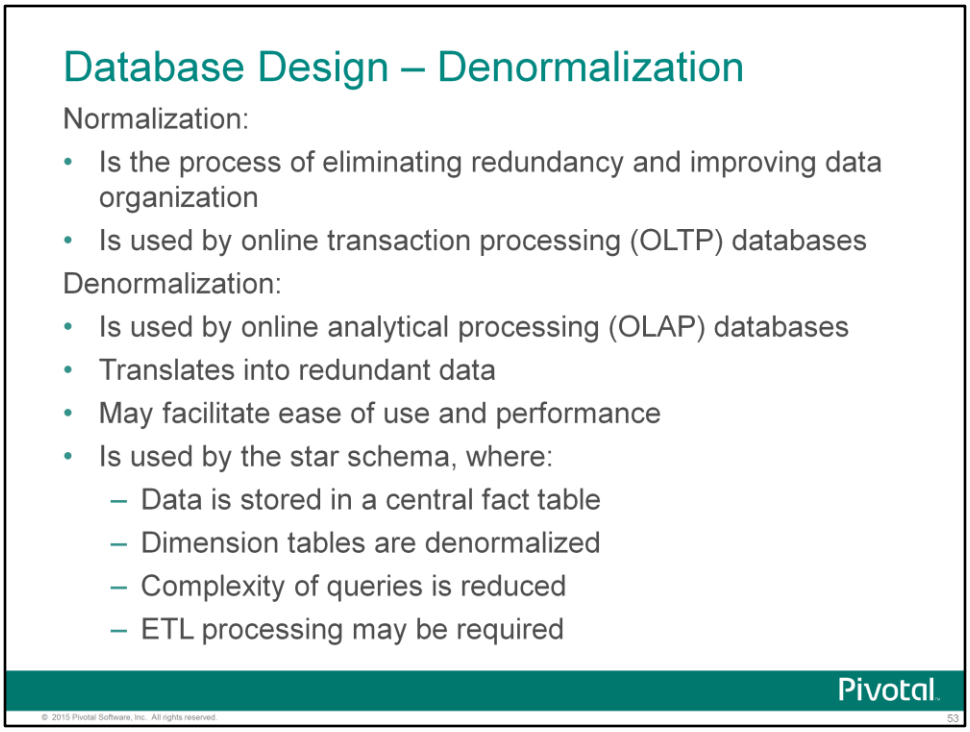
For example, use:

- Character data types for strings
- Date or timestamp data types for dates
- Numeric data types for numbers. Using test data types for numbers introduces more overhead than needed – character set and locale checks. For numeric column data types, use the smallest data type in which the data will fit. A lot of space is wasted if, for example, the `BIGINT` data type is used when the data would always fit in INT or `SMALLINT`.
- For character column data types, there are no performance differences between the use of `CHAR`, `VARCHAR`, and `TEXT` data types, apart from the increased storage size when using the blank-padded type. While `CHAR` has performance advantages in some other database systems, it has no such advantages in Greenplum Database. In most situations, `TEXT` or `VARCHAR` should be used instead.

Database Design – Selecting Appropriate Data Types (Continued)

- Use identical data types for the columns you plan to use in cross-table joins. Joins work much more efficiently if the data types of the columns used in the join predicate (usually the primary key in one table and a foreign key in the other table) have identical data types. When the data types are different, the database has to convert one of them so that the data values can be compared correctly, and such conversion amounts to unnecessary overhead.

Refer to the *Greenplum Database Administrator Guide* for a list of the built-in data types available in Greenplum Database and their associated sizing information.



In relational database theory, normalization is the process of restructuring the logical data model of a database to eliminate redundancy and improve data organization. The goals of normalization are to improve data integrity, eliminate repeating data, and reduce the potential for anomalies during database operations.

Databases intended for Online Transaction Processing (OLTP) are typically more normalized than databases intended for Online Analytical Processing (OLAP). OLTP and OLAP systems have very different requirements. OLTP applications are characterized by a high volume of small transactions, each transaction leaving the database in a consistent state. By contrast, databases intended for OLAP operations are primarily read-only databases. OLAP applications tend to extract historical data that has accumulated over a long period of time. For OLAP databases, redundant or denormalized data may facilitate ease-of-use and performance.

Data warehouses often use denormalized or partially denormalized schemas (such as a star schema) to optimize query performance. In a star schema design, data is stored in a central fact table surrounded by one or more denormalized dimension tables. One advantage of a star schema design is improved performance because the data is pre-joined, reducing the complexity of the queries. Also, because of the simplicity of the design, it is easy for business users to understand and use. One disadvantage of the dimensional approach is that some type of ETL process is required to guarantee data consistency.

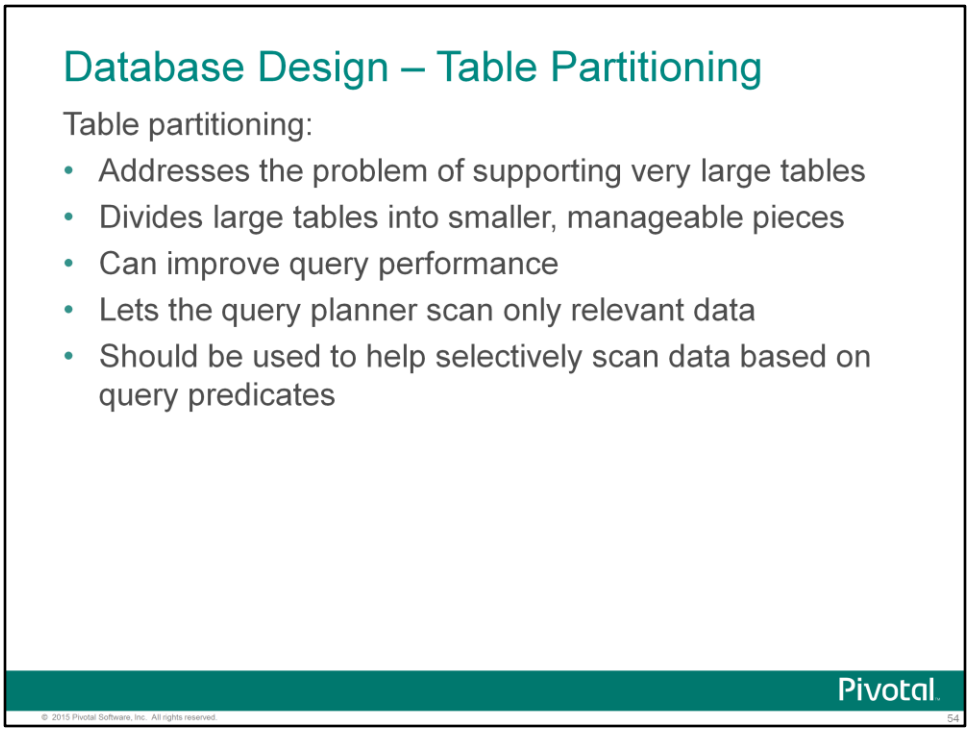
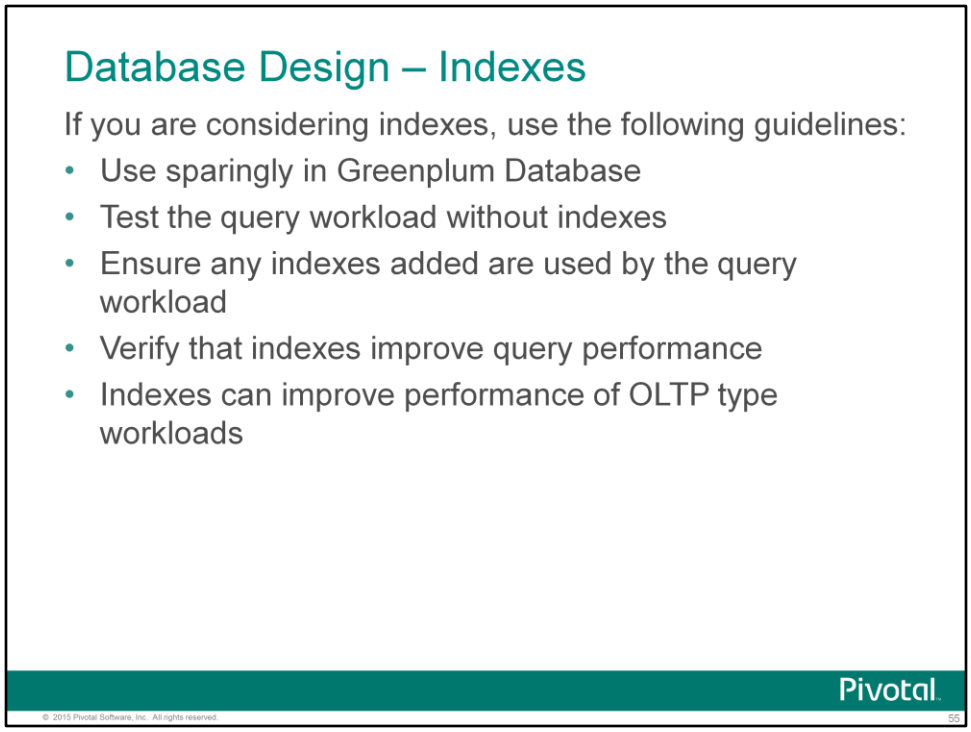


Table partitioning addresses the problem of supporting very large tables, such as fact tables, by allowing you to divide them into smaller and more manageable pieces. Partitioned tables can improve query performance by allowing the Greenplum Database query planner to scan only the relevant data needed to satisfy a given query rather than scanning the entire contents of a large table. Partitioned tables can also be used to facilitate database maintenance tasks, such as rolling old data out of the data warehouse. Keep in mind that partitioning should be used to help selectively scan data based on query predicates. Any query that does not scan according to the partition design will not benefit from this approach and can even slightly lessen performance. In this case, there are more scans performed but the scan results are not used to filter data partitions.

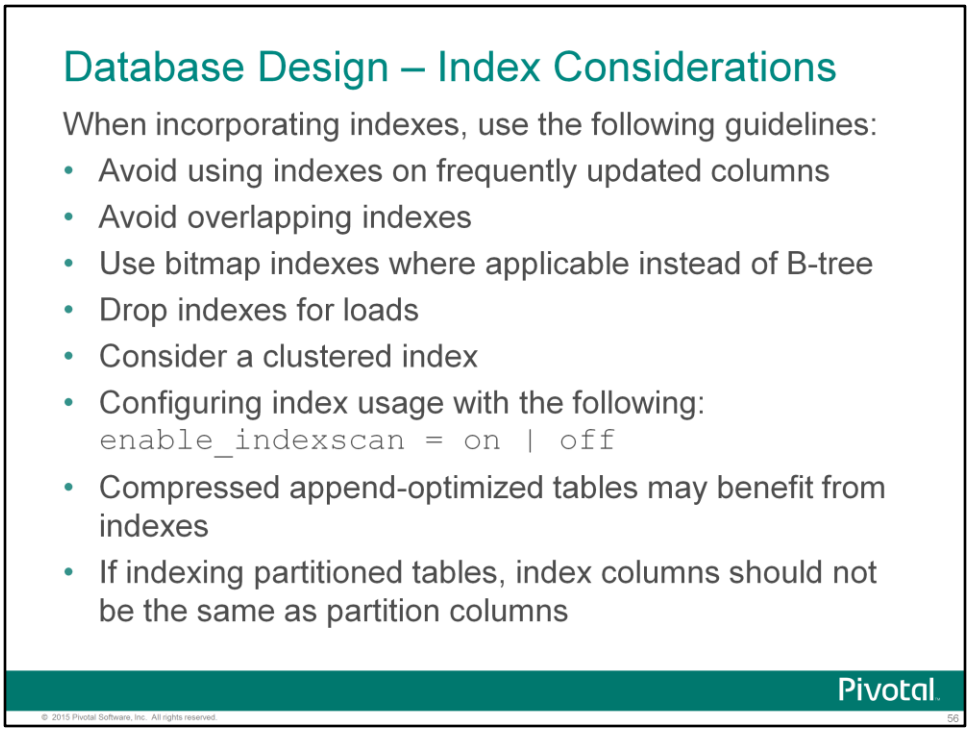


Greenplum Database is very fast at sequential scanning, where as indexes use a random seek pattern to locate records on disk. As the data is distributed across segments, each segment scans a smaller portion of the overall data in order to get the desired result. If you are using table partitioning, the total data to scan may be even a fraction of that.

Follow these guidelines when considering indexes:

- Test your query workload without adding any additional indexes. Greenplum Database will automatically create PRIMARY KEY indexes for tables with primary keys. If you experiencing unsatisfactory performance, then you may try adding indexes to see if performance improves. Indexes do add some database overhead. Indexes:
    - Consume storage space.
    - Must be maintained whenever the table is updated.
- Ensure indexes you create are being used by your query workload.
- Verify that the indexes you add improve query performance, compared to a sequential scan of the table. You can do this by executing `EXPLAIN` on a query to see its plan.
  `EXPLAIN` will be covered later in the course.

Indexes are more likely to improve performance for OLTP type workloads, where the query is returning a single record or a very small data set. Typically, a business intelligence query workload returns very large data sets. It therefore does not make efficient use of indexes. For this type of workload, it is better to use sequential scans to locate large chunks of data on disk rather than to randomly seek the disk using index scans.

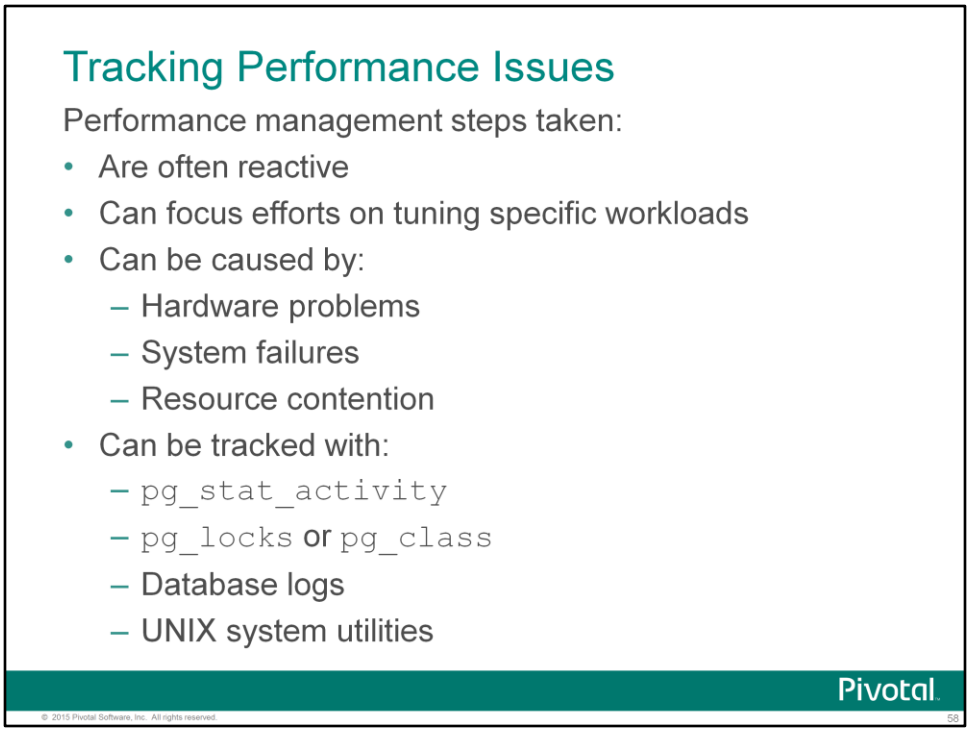


Consider the following when incorporating indexes into your design:

- Avoid indexes on frequently updated columns. Creating an index on a column that is frequently updated increases the amount of writes required when the column is updated.
- Use Bitmap indexes for low selectivity columns. Greenplum Database has an additional index type called a Bitmap index, which is a fraction of the size of B-tree indexes for low cardinality columns. Performance is boosted for queries that use multiple predicates to filter results of bitmap indexed columns. Filtering is done before row is fetched.
- Avoid overlapping indexes. Overlapping indexes (those that have the same leading column) are redundant and unnecessary.
- Drop indexes for loads. For mass loads of data into a table, consider dropping the indexes and re-creating them after the load is complete. This is often faster than updating the indexes.

Database Design – Index Considerations (Continued)

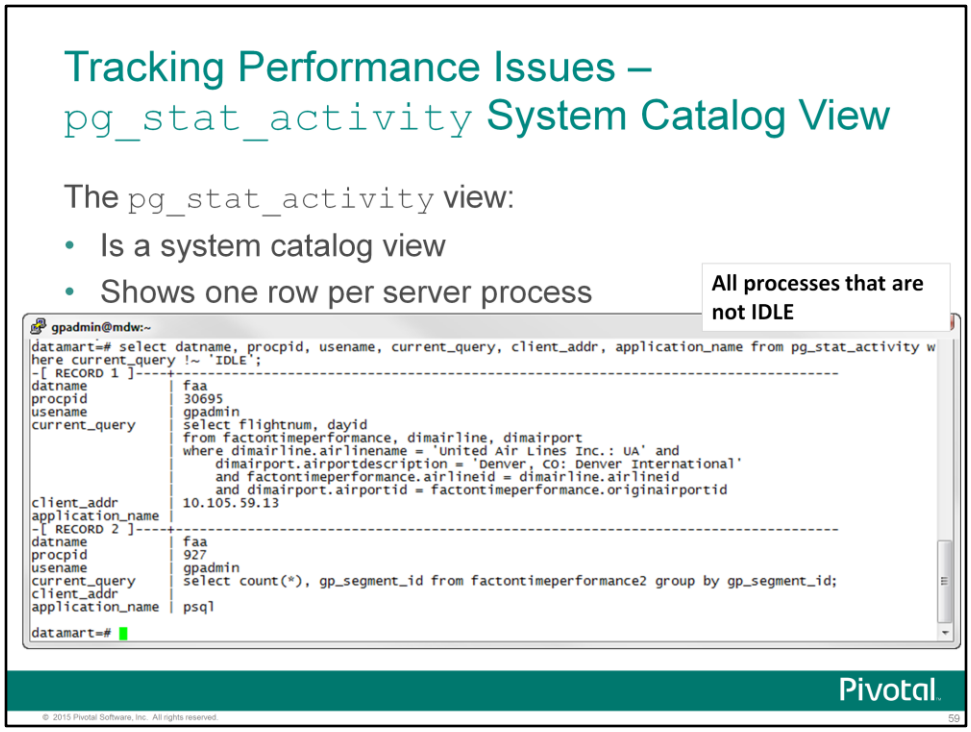
- **Clustered index**. Clustering an index means that the records are physically ordered on disk according to the index. If the records you need are distributed randomly on disk, then the database has to seek across the disk to get the records requested. If those records are stored more closely together, then the fetching from disk is more sequential. A good example for a clustered index is on a date column where the data is ordered sequentially by date. A query against a specific date range will result in an ordered fetch from the disk, which leverages fast sequential access. Postgres has a CLUSTER command that does this, but Greenplum does not recommend its use because it takes a really long time to cluster in this way. Instead do a CREATE TABLE AS SELECT ORDER BY and order the data according to the associated column.
- **Configure index usage** – The enable_indexscan parameter allows you to turn off use of index scan operations in query plans – useful for confirming performance gains (or losses) of using an index.
- **Indexes on append-optimized compressed tables –** Compressed data requires additional CPU cycles to uncompress when retrieving or writing data to the table. An index on a compressed append-optimized table can improve performance as only the rows that are being retrieved will be uncompressed.
- **Indexing partitioned tables –** While it is not considered a good practice to index partitioned tables, should you need to do so, the indexed column must not be the same as the partitioned column. Remember, the goal of partitioning is to reduce the number of rows that need to be scanned. The use of indexes should be to further improve the performance of finding rows in a subset of the already partitioned rows.



Many of the performance management steps taken by database administrators are reactive - a business user calls with a response time problem, a batch job fails, the system suddenly becomes unavailable. If the problem is related to a particular workload or query, then you can focus your efforts on tuning that particular workload. If the performance problem is system-wide, then hardware problems, system failures, or resource contention may be the cause.

Problems can be tracked down using a combination of the following:

- The `pg_stat_activity` view
- The `pg_locks` or `pg_class` views
- Database logs
- UNIX system utilities

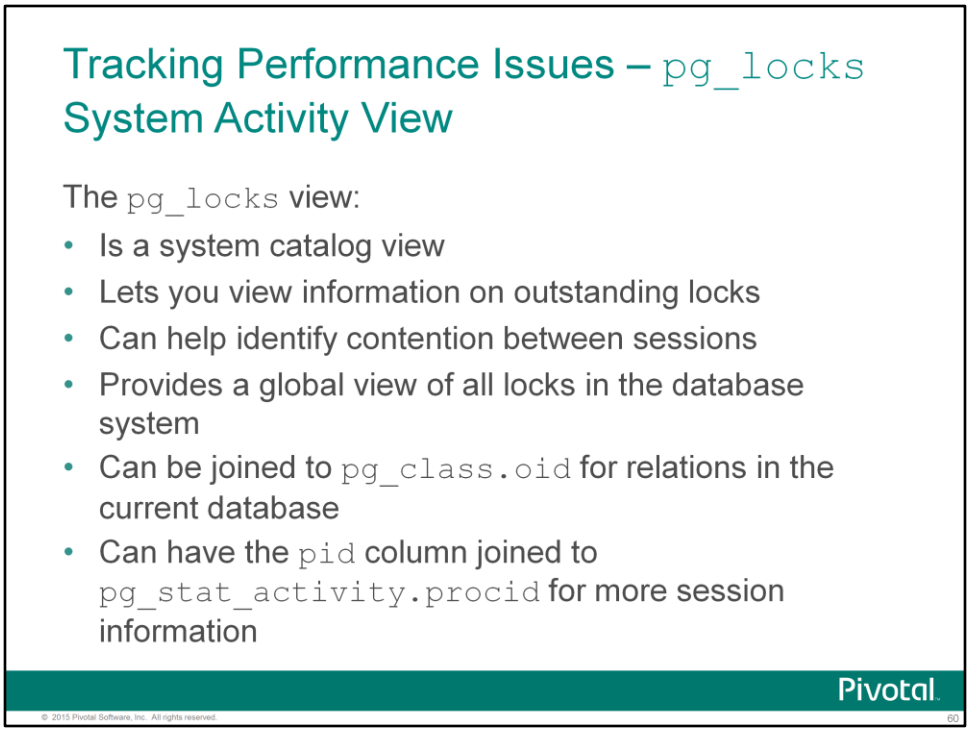


This system view shows one row per server process . The view contains the following fields:

- Database OID
- Database name
- Process ID
- User OID
- User name
- Current query
- Time at which the current query began execution
- Time at which the process was started
- Client address
- Port number
- Application name
- Transaction start time

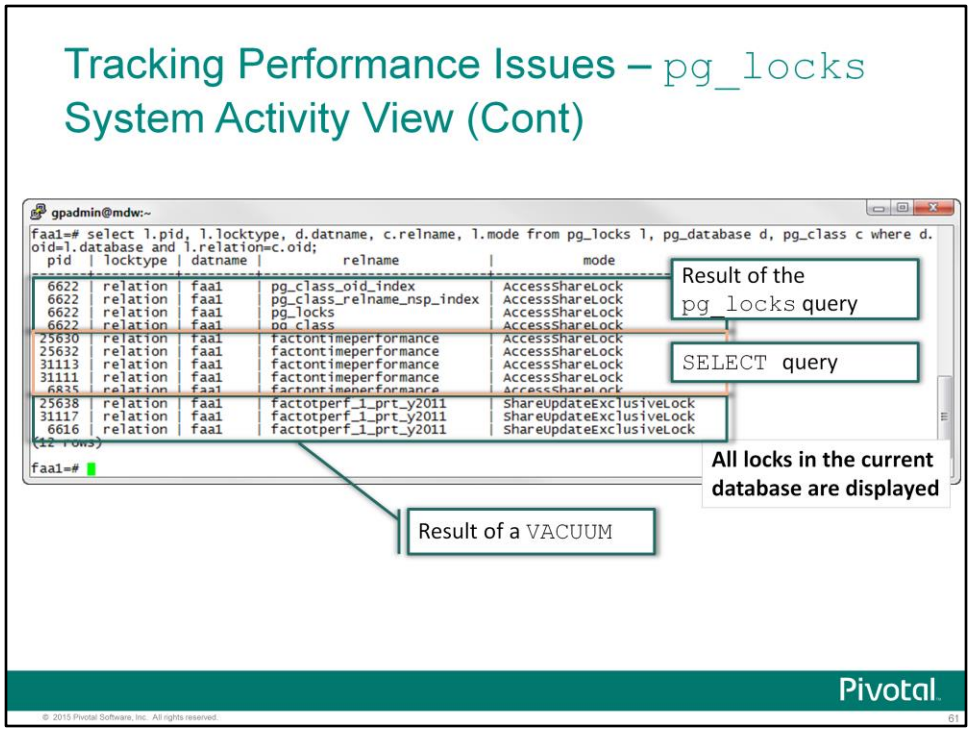
Querying this view can provide more information about the current workload on the system. This view should be queried as the database superuser to obtain the most information possible. Also note that the information does not update instantaneously.

The output shown displays all processes executing that are not idle. The output is formatted with the PSQL metacommand, `\x`.



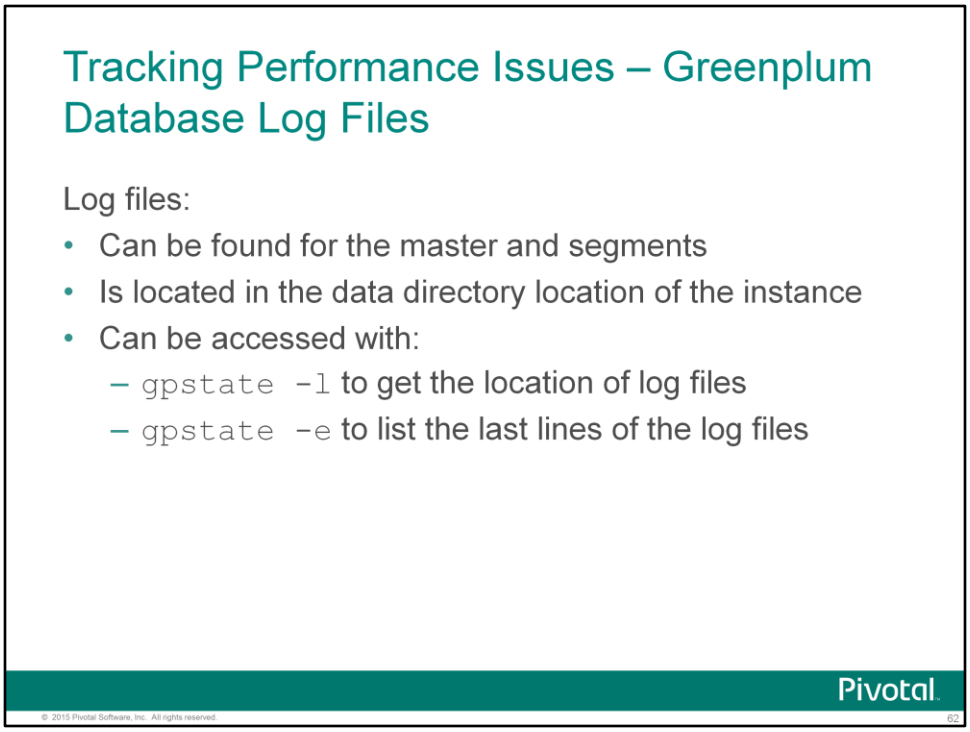
The `pg_locks` system view allows you to view information about outstanding locks. Examining `pg_locks` for ungranted locks can help identify contention between database client sessions. The `pg_locks` view provides a global view of all locks in the database system, not only those relevant to the current database.

Although its relation column can be joined against `pg_class.oid` to identify locked relations, such as tables, this will only work correctly for relations in the current database. The pid column can be joined to the `pg_stat_activity.procpid` to get more information on the session holding or waiting to hold a lock.



The pg_locks system activity view contains the following columns:

- Type of the lockable object.
- OID of the database in which the object exists.
- OID of the relation, if it exists. If it does not exist, the value for the column is NULL.
- Page number within the relation.
- Tuple number within the page.
- Transaction ID
- OID of the system catalog containing the object.
- OID of the object within the system catalog.
- Column number.
- Transaction ID that is holding or awaiting the lock.
- Process ID associated with the server process holding or awaiting the lock.
- Name of the lock.
- Boolean specifying whether the lock is held or is being waited on.
- Client session ID associated with the lock holder.
- Boolean specifying if the lock is being held by the writer process.
- Segment ID where the lock is being held.

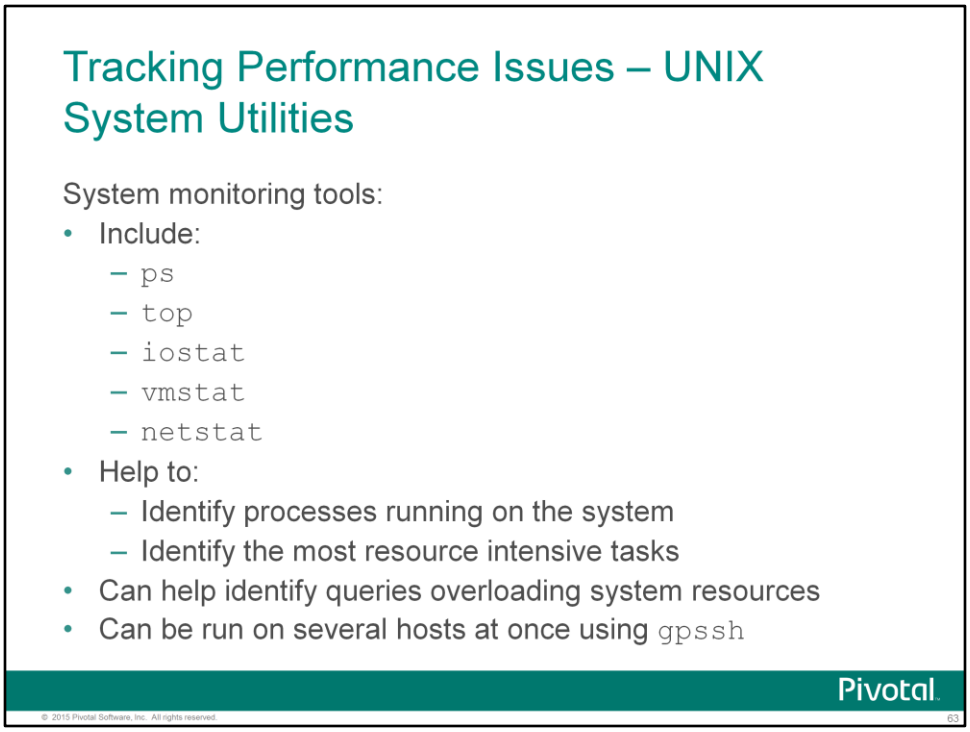


The log files for each Greenplum Database server instance, master and segments, can be found in the data directory location of that instance.

If you are not sure of the location of the data directories, you can use:

- **`gpstate -l`** to determine the location of the log files
- **`gpstate -e`** to view last lines of all log files

The `gpstate` command displays information about a running Greenplum Database system. It can be used to identify failed segments.

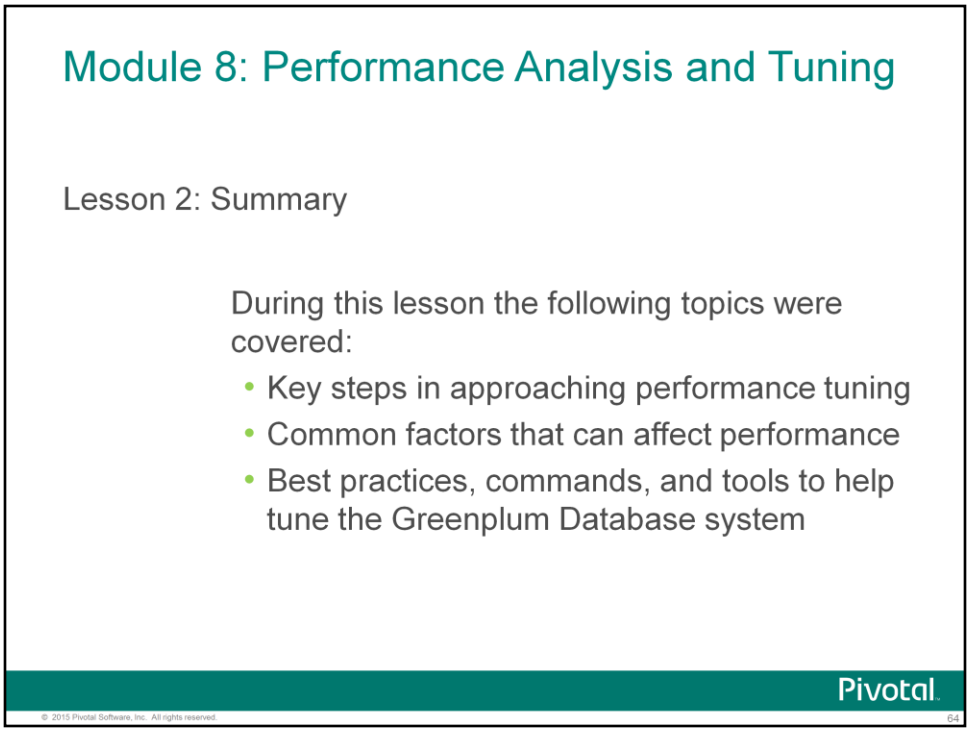


System monitoring utilities such as `ps`, `top`, `iostat`, `vmstat`, and `netstat` can be used to monitor database activity on the hosts in your Greenplum Database array.

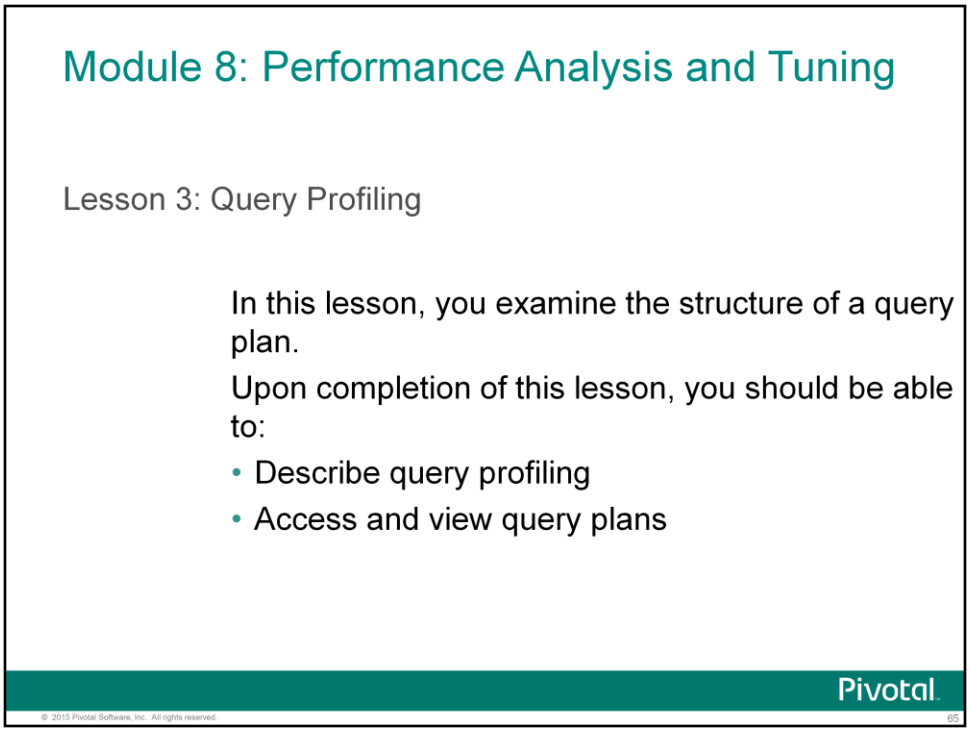
These tools can be used to help:

- Identify Greenplum Database processes, including postmaster/postgres processes, currently running on the system.
- The most resource intensive tasks with regards to CPU, memory, disk I/O, or network activity.

Looking at these system statistics can help identify queries that are overloading the system by consuming excessive resources and thereby degrading database performance. Greenplum Database comes with a management tool called `gpssh`. This allows you to run these system monitoring commands on several hosts at once.



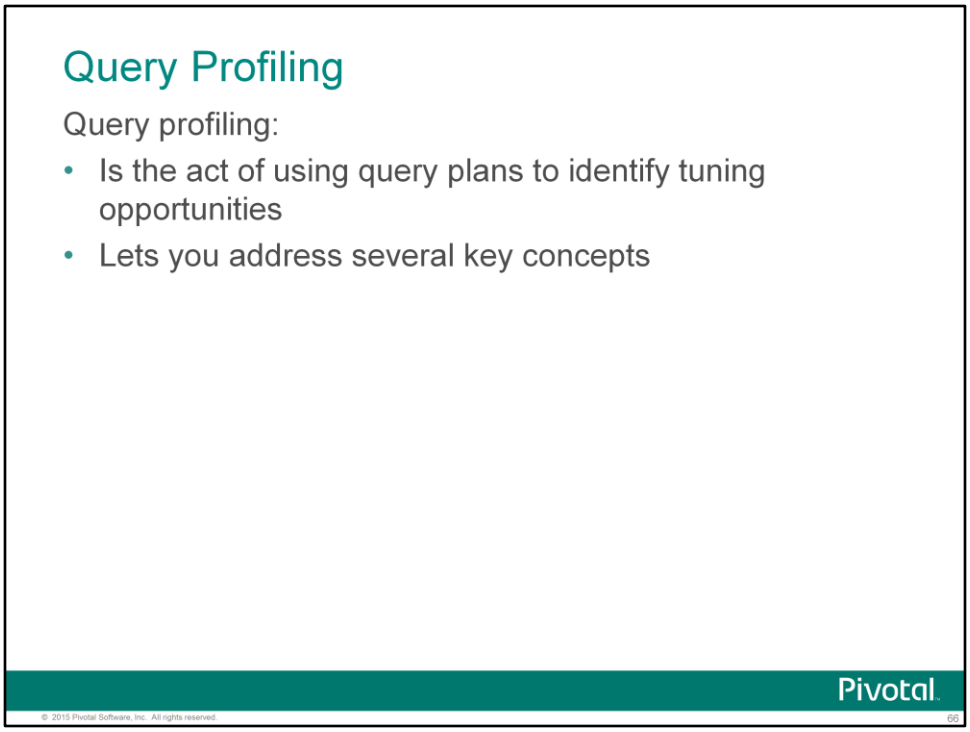
This lesson discussed the key impacts and steps in approaching performance tuning and the common factors that can affect the overall performance of the system and queries. Best practices, commands, and tools, views, and tables that you can use to help tune the system and queries are also provided.



Understanding the structure of a query plan can help you decide how best to tune the database or your queries for the fastest and most efficient response time.

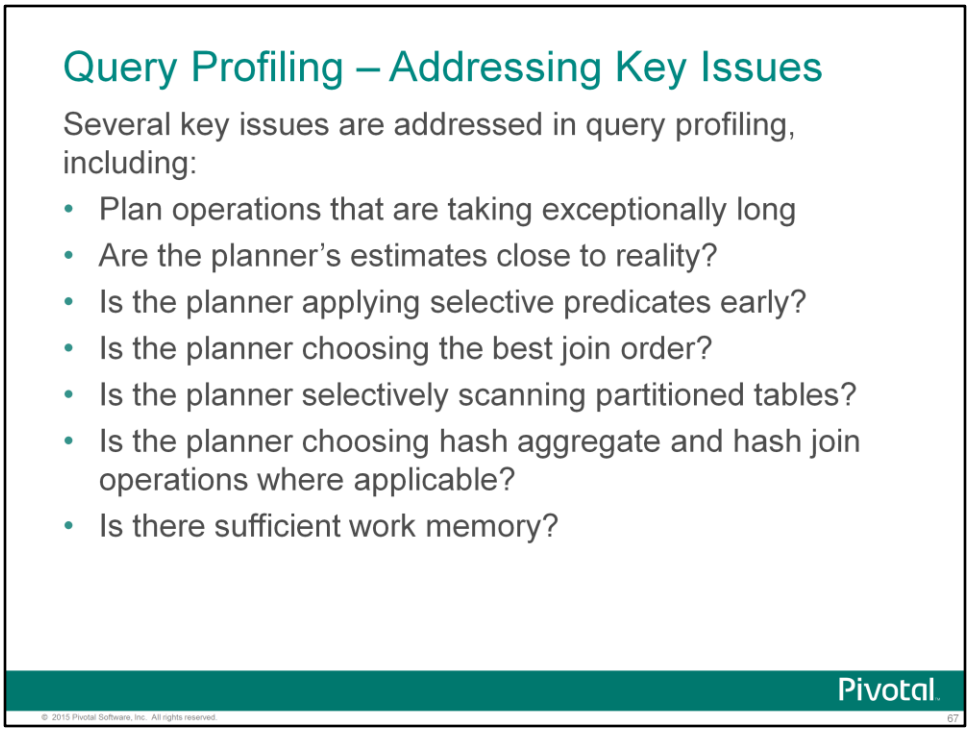
In this lesson, you:

- Describe query profiling.
- Access and view query plans.



Query profiling is the act of looking at the query plan, using `EXPLAIN` or `EXPLAIN ANALYZE`, to identify tuning opportunities of poorly performing queries. You should only tackle query profiling and tuning when you are first designing your database or when you are experiencing poor performance. Query profiling requires that you know your data well. You cannot identify problems you know what to expect.

There are several major issues or key items that query profiling allows you to address.

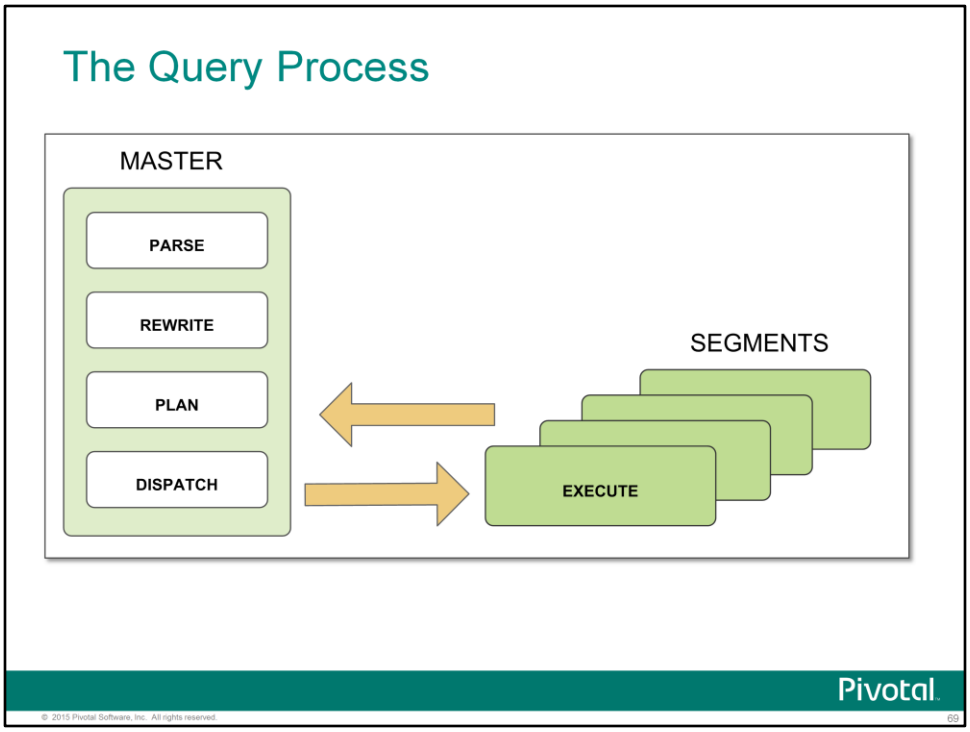


Query profiling lets you look at the following key items:

- Is there one operation in the plan that is taking exceptionally long? When looking through the query plan, is there one operation that is consuming the majority of the query processing time as compared to other operations? For example, is an index scan taking a lot longer than expected? Maybe the index needs to be `REINDEX`ed because its out of date. Maybe see if you can force the planner to choose another operation by turning `enable_indexscan=off` temporarily and then rerunning the query.
- Are the planner's estimates close to reality? Run an `EXPLAIN ANALYZE` and see if the number of rows estimated by the planner is close to the number of rows actually returned by the query operation. If there is a huge discrepancy, you may need to collect more statistics on the relevant columns.
- Are selective predicates applied early in the plan? The most selective filters should be applied early in the plan so that less rows move up the plan tree. If the query plan is not doing a good job at estimating the selectivity of a query predicate, you may need to collect more statistics on the relevant columns. You can also try reordering the `WHERE` clause of your SQL statement.

Query Profiling – Addressing Key Issues (Continued)

- Is the planner choosing the best join order? When you have a query that joins multiple tables, make sure that the planner is choosing the most selective join order. Joins that eliminate the largest number of rows should be done earlier in the plan so that less rows move up the plan tree. If the plan is not choosing the optimal join order you can use explicit JOIN syntax in your SQL statement to force the planner to use a specified join order. You can also collect more statistics on the relevant join columns.
- Is the planner selectively scanning partitioned tables? If you are using table partitioning, is the planner selectively scanning only the child tables required to satisfy the query predicates? Do scans of the parent tables return 0 rows (they should, since the parent tables should not contain any data).
- Is the planner choosing hash aggregate and hash join operations where applicable? Hash operations are typically much faster than other types of joins or aggregations. Row comparison and sorting is done in memory rather than reading/writing from disk.
- Is there sufficient work memory? In order for hash operations to be chosen, there has to be sufficient work memory available to hold the number of estimated rows. Try increasing work memory to see if you can get better performance for a given query. If possible run an `EXPLAIN ANALYZE` for the query, which will show you which plan operations spilled to disk, how much work memory they used, and how much was required to not spill to disk.

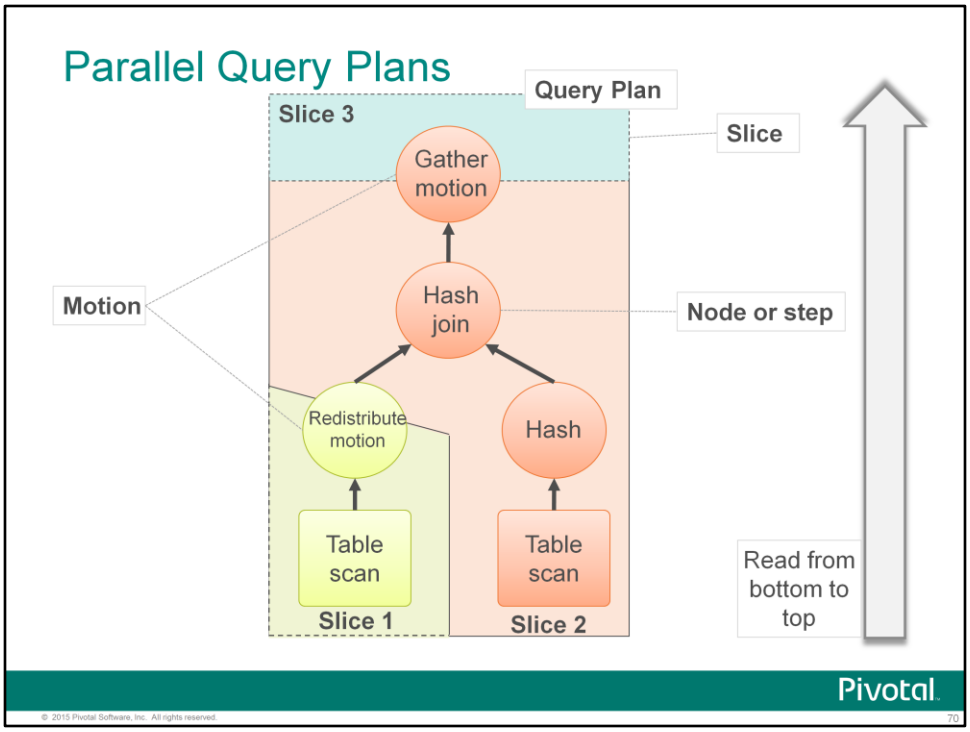


When a query is issued to Greenplum Database, the master:

- Parses the query and checks syntax.
- Sends the query to the query rewrite system where VIEWS and RULES are processed.
- Creates a parallel query execution plan.
- Slices the plan and each segment is dispatched its slice to work on.

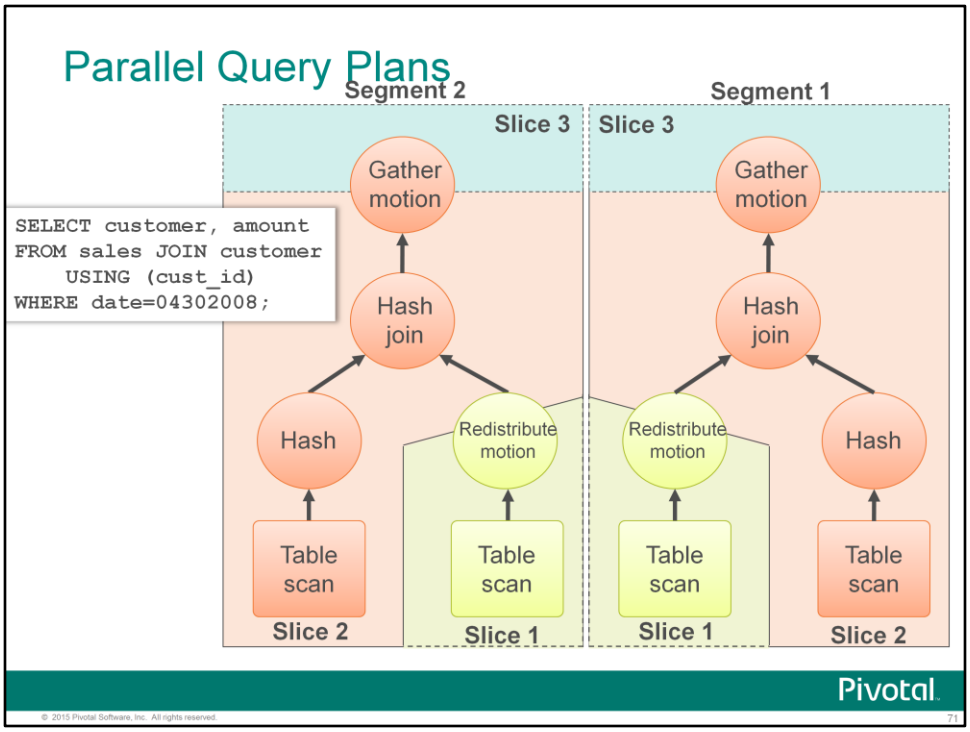
The segments execute the query between themselves, moving data around as needed over the interconnect.

When execution is complete, the segments return the results to the master. Sometimes the master may do final aggregations for small data sets only and then the master sends the result to the client. Most of the tuning of queries is focused on the planning stage of this process. You want the planner to give the best plan, and thereby the best performance possible.



A query plan:

- Is the set of operations the Greenplum Database performs to produce the answer to a given query.
- Consists of nodes or steps which represent a database operation such as a table scan, join, aggregation or sort.
- Is read and executed from bottom to top of the query plan. Typical database operations , such as tables scans, and joins include an additional operation called a *motion*.
- May include a motion operation which involves moving tuples, or rows, between the segments during query processing.
- Is divided into slices to achieve maximum parallelism during query execution. A slice is a portion of the plan that can be worked on independently at the segment level.
- Is sliced wherever there is data movement in the query plan, one slice on each side of the motion. If a join occurs on distributed tables and there is no data movement, there are no slices created. If there is data movement, slices are created for each data movement point.
- Can contain a gather motion, which is when the segments send results back up to the master for presentation to the client.



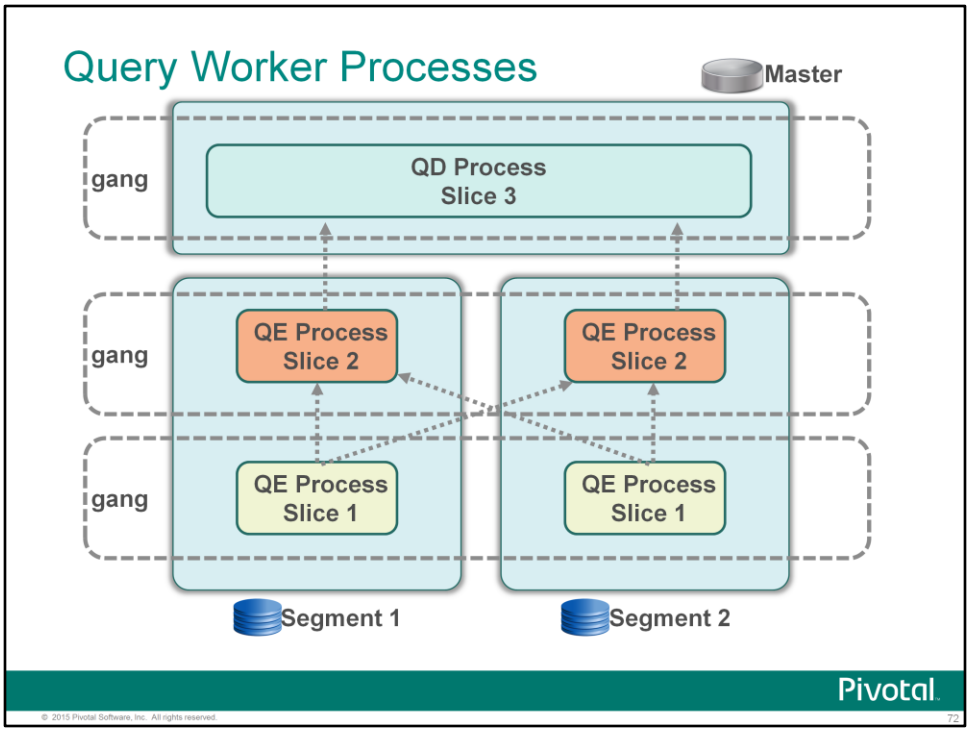
Each segment gets a copy of the query plan and works on it in parallel.

Consider the following simple query involving a join between two tables:

```
SELECT customer, amount
FROM sales JOIN customer USING (cust_id)
WHERE date=04302008;
```

There is a redistribute motion that moves tuples between the segments to complete the join. The plan is sliced on either side of the redistribute motion, creating slice 1 and slice 2. This query plan also includes a gather motion. Since a query plan is always sliced wherever a motion occurs, this plan also has an implicit slice at the very top of the plan, slice 3. Not all query plans involve a gather motion. For example, a CREATE TABLE x AS SELECT... statement would not have a gather motion (tuples are sent to the newly created table, not to the master).

Note: If a SORT is performed on the data, that is performed before the Gather Motion action occurs.

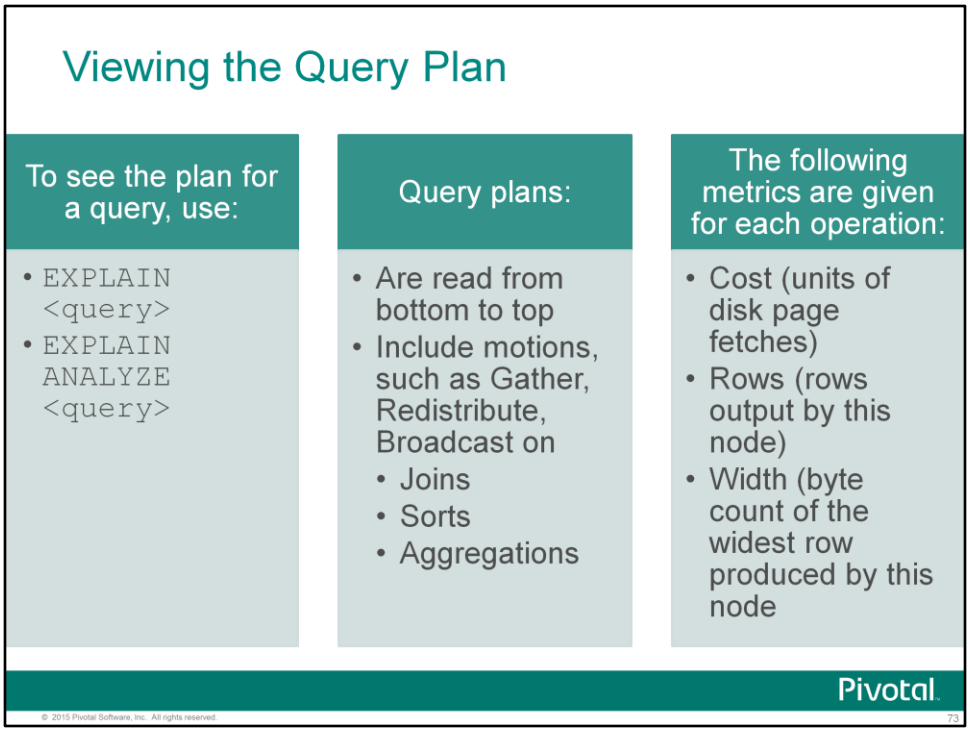


Greenplum creates a number of database processes, postgres, to handle the work of a query:

- **Query dispatcher** – On the master, the query worker process is called the query dispatcher (QD). The QD is responsible for creating and dispatching the query plan, and for accumulating and presenting the final results.
- **Query executor** – On the segments, a query worker process is called a query executor (QE). A QE is responsible for completing its portion of work and communicating its intermediate results to the other worker processes. For each slice of the query plan, there is at least one worker process assigned. A worker process works on its assigned portion of the query plan independently. During query execution, each segment will have a number of processes working on the query in parallel.

Related processes that are working on the same portion of the query plan are referred to as *gangs*. Gangs are connected by a connection identifier that can be retrieve by issuing a process listing command. Process identifiers are prefixed with the term, con. For example, all the gather slices in this example may show up in the process list as con13.

As a portion of work is completed, tuples flow up the query plan from one gang of processes to the next. This inter-process communication between the segments is what is referred to as the interconnect component of Greenplum Database.



Use the following commands to see the query plan:

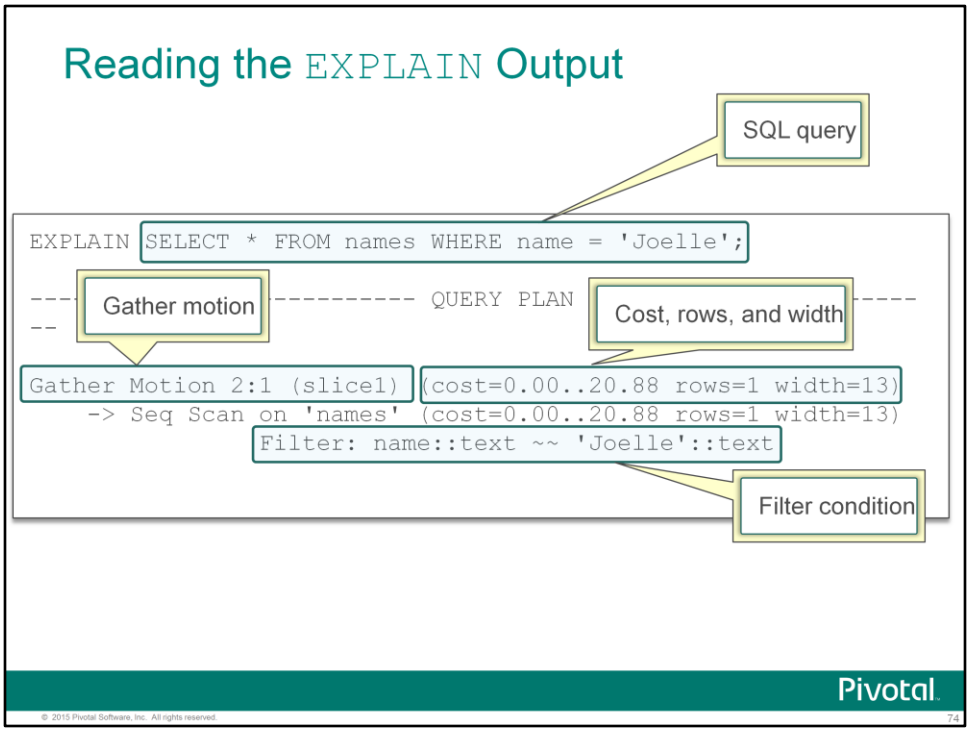
- **`EXPLAIN`** – The EXPLAIN command lets you view the query plan for a query.
- **`EXPLAIN ANALYZE`** – This command runs the query and give you the actual metrics rather than just the estimates.

A query plan is a tree plan of nodes. Each node represents a database operation such as a table scan, join, or sort. Query plans are read from bottom to top.

Greenplum Database has an additional query plan node type called a motion node, which are the operations that move tuples between segments.

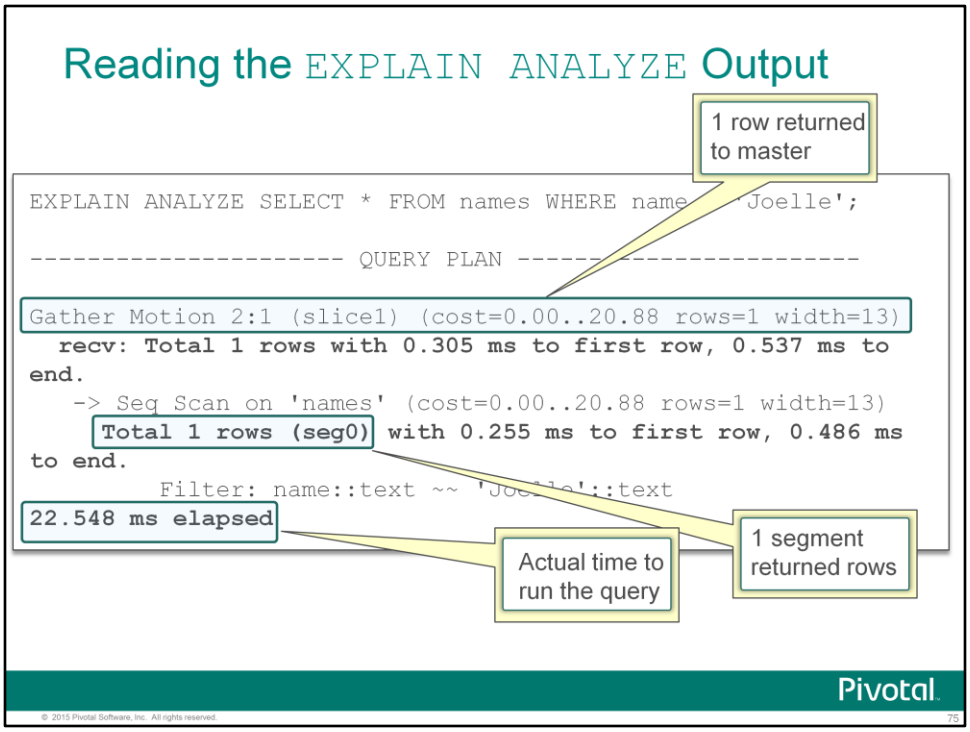
In all query plan nodes or operations, the following metrics are given:

- Cost is measured in units of disk page fetches; that is, 1.0 equals one sequential disk page read. The first estimate is the start-up cost and the second is the total cost.
- Rows is the total number of rows output by this plan node.
- Width is the width (in bytes) of the widest row produced by this plan node.



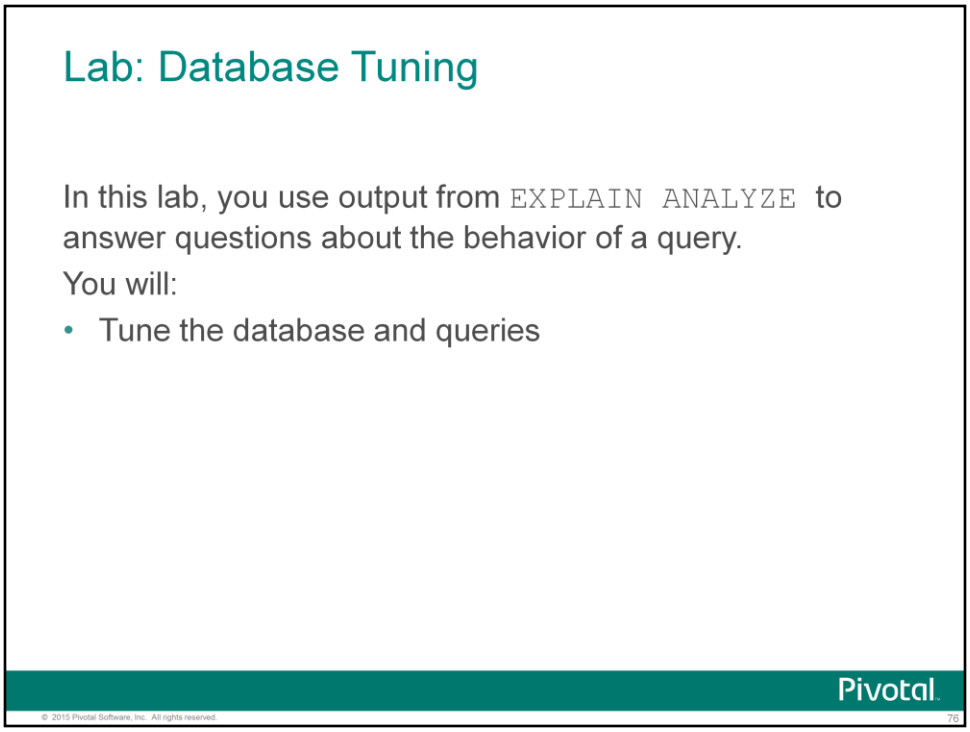
To illustrate how to read an EXPLAIN query plan, consider the following example for a very simple query:

- When reading the plan from the bottom of the output, the query planner starts by doing a sequential scan of the names table.
- Note that the `WHERE` clause is being applied as a filter condition. This means that the scan operation checks the condition for each row it scans, and outputs only the ones that pass the condition.
- The results of the scan operation are passed up to a gather motion operation. In Greenplum Database, a gather motion is performed when segments send rows up to the master. In this case we have 2 segment instances sending to 1 master instance (2:1). This operation is working on slice1 of the parallel query execution plan. In Greenplum Database a query plan is divided into slices so that portions of the query plan can be worked on in parallel by the segments.
- The estimated startup cost for this plan is 00.00 (no cost) and a total cost of 20.88 disk page fetches. The planner is estimating that this query will return one row and the widest row is 13 bytes.

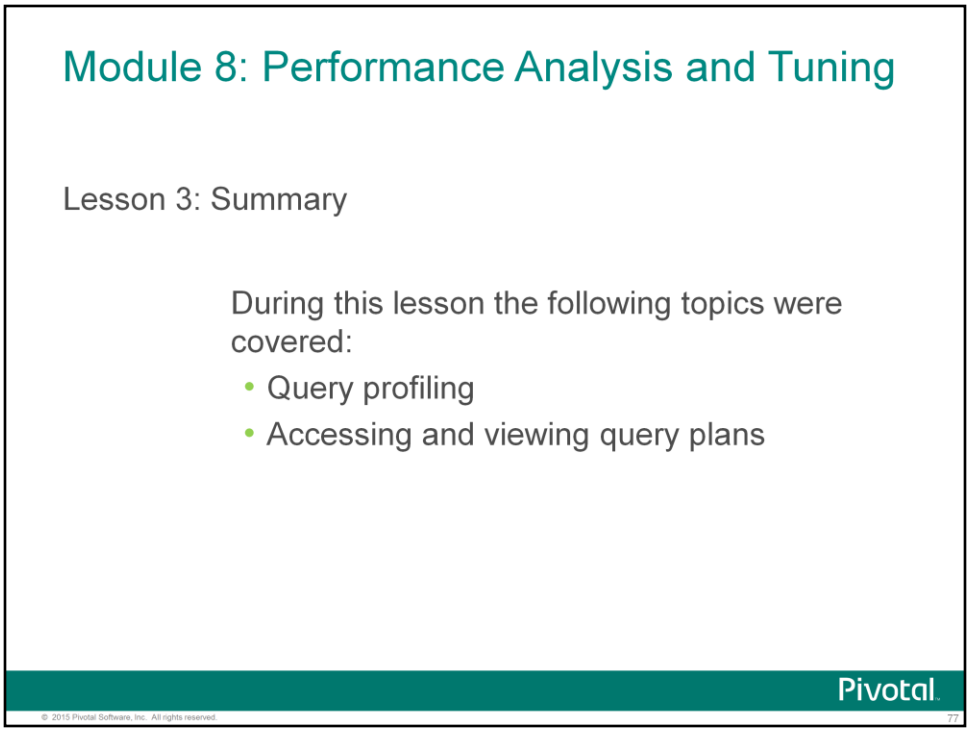


To illustrate how to read an EXPLAIN ANALYZE query plan, let us examine the same simple query we used in the EXPLAIN example:

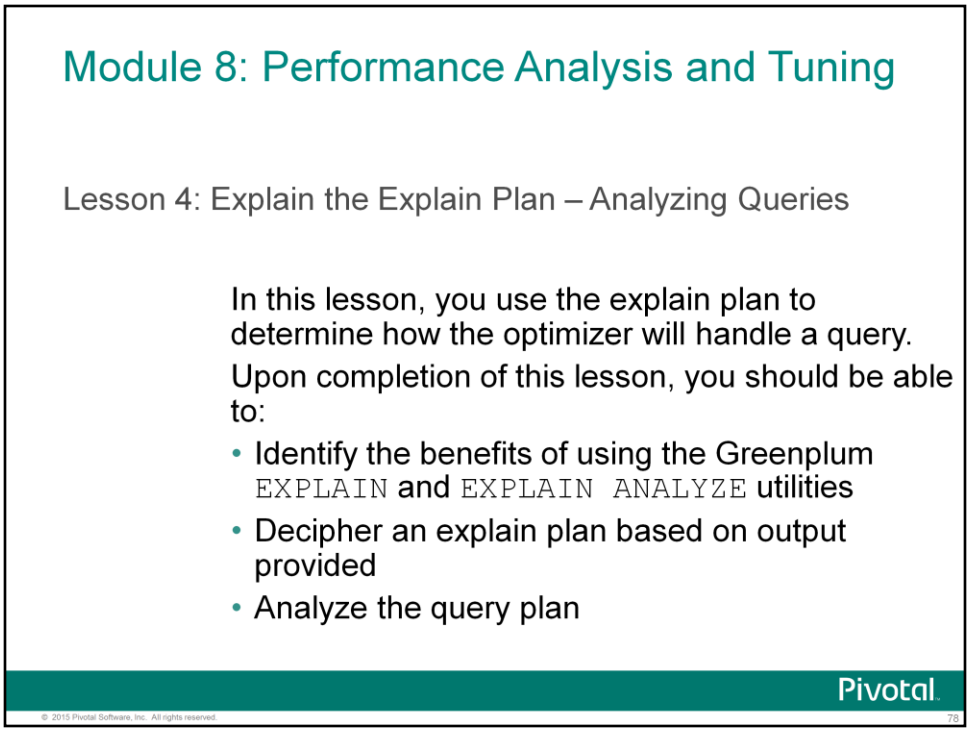
- Notice that there is some additional information in this plan that is not in a regular EXPLAIN plan. The parts of the plan in bold show the actual timing and rows returned for each plan node.
- Reading the plan from the bottom up, you will see some additional information for each plan node operation. The total elapsed time it took to run this query was 22.548 milliseconds.
- The sequential scan operation had only one segment (seg0) that returned rows, and it returned just 1 row. It took 0.255 milliseconds to find the first row and 0.486 to scan all rows. Notice that this is pretty close to the planner's estimate - the query planner estimated that it would return one row for this query, which it did.
- The gather motion operation then received 1 row (segments sending up to the master). The total elapsed time for this operation was 0.537 milliseconds.



In this lab, you use output from `EXPLAIN ANALYZE` to answer questions about the behavior of a query.



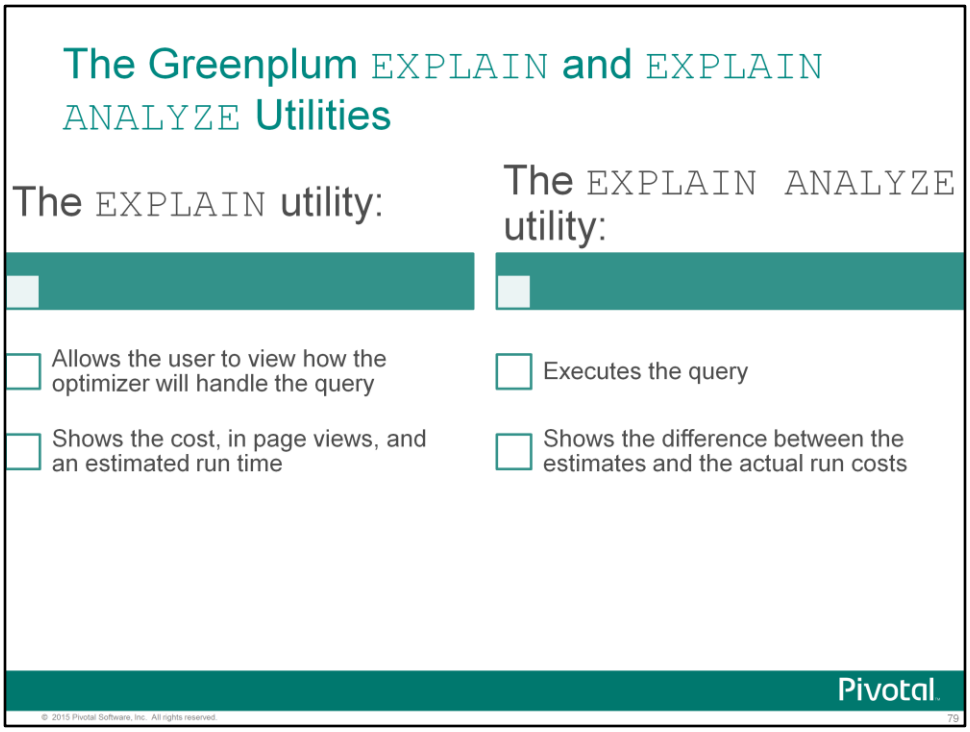
This lesson discussed query profiling, the major components of the query process, and the ability to access and view query plans. For every query issued to the system, the planner creates a plan based on the query, parameter settings, the environment, and the object structure. There are plenty of opportunities to tune the query to ensure that it is optimized for execution.



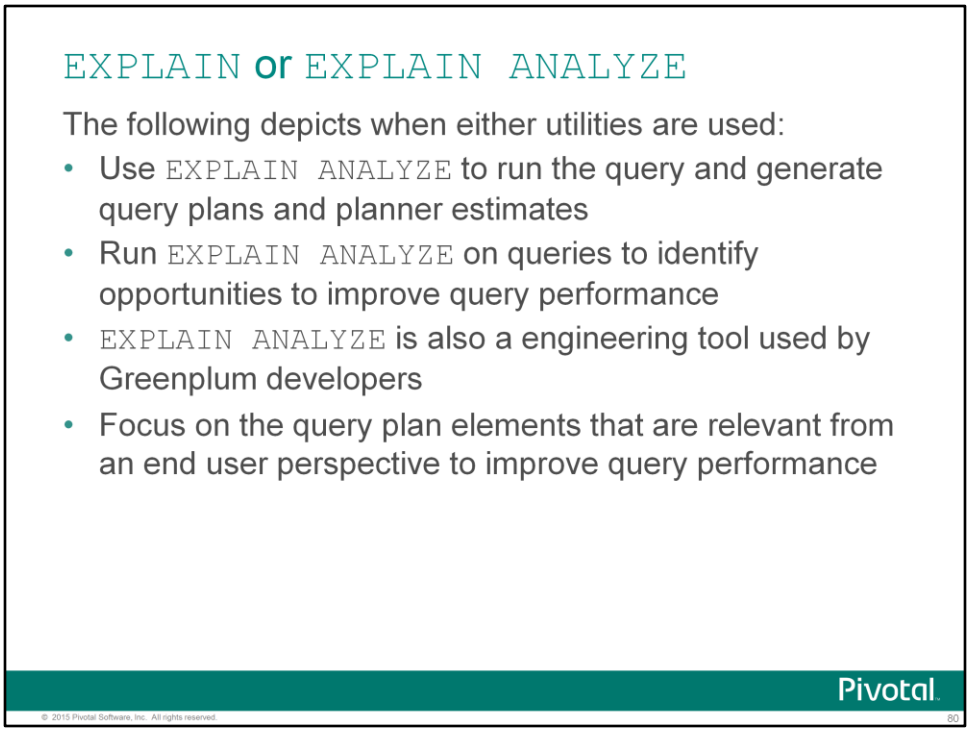
In this lesson, you explore the explain plan in greater detail to see how the optimizer handles a query. By analyzing exactly what occurs with the query planner, you can see how providing the most accurate statistics can affect what the optimizer generates for the query plan.

In this lesson, you will:

- Identify the benefits of using the Greenplum `EXPLAIN` and `EXPLAIN ANALYZE` utilities.
- Decipher an explain plan based on output provided.
- Analyze the query plan.



EXPLAIN gives you a guess as to what it's going to do, based on statistics. EXPLAIN ANALYZE runs the query and tells you what it did. This does give an accurate depiction of the time taken to run the query, whereas EXPLAIN produces an estimate of the time it will take to execute the command.



The query plan will define how the query will be executed in the Greenplum database environment. The query planner uses the database statistics to determine the query plan with the lowest possible cost. Cost is measured in disk I/O, CPU and memory. The goal is to minimize the total execution cost for the plan.

Use `EXPLAIN ANALYZE` to execute the query and generate the query plan with estimates. Run `EXPLAIN ANALYZE` on queries to identify opportunities to improve query performance.

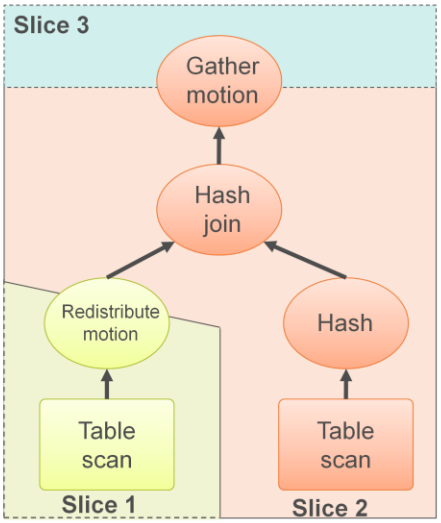
`EXPLAIN ANALYZE` is also an engineering tool used internally by Greenplum developers in the product development lifecycle. Query plans have information that assists the developers but also provides information from an end user perspective to analyze query execution and improve query performance.

Therefore, as an end user it is very important to focus on the query plan elements that are relevant to improving query performance.

EXPLAIN Output

In query plans:

- Each node feeds its results to the node directly above
- There is one line for each node in the plan tree
- Each node represents a single operation
- Motion nodes are responsible for moving rows between segment instances



Pivotal

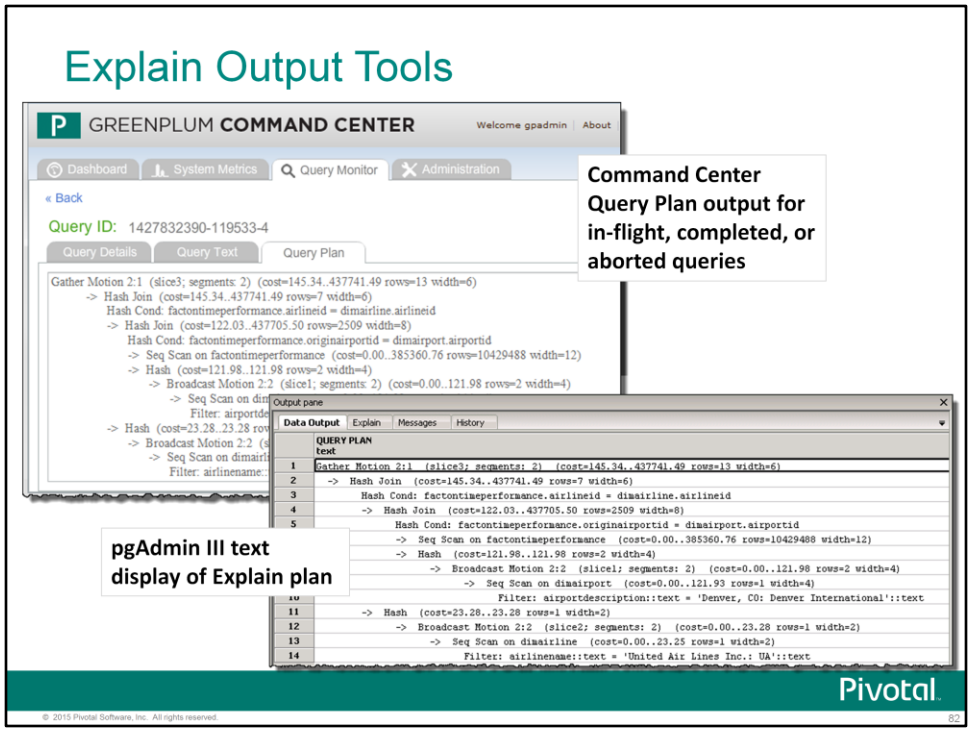
81

Query plans are a right to left tree plan of nodes that are read from bottom to top with each node passing its result to the node directly above. There is one line for each node in the plan tree that represents a single operation, for example scan, join, aggregation or sort operation.

The node will identify the method used to perform the operation. For example a scan operation may perform a sequential scan or index scan. A join operation may perform a hash join or nested loop join.

The query plan will also include motion nodes. The motion operations are responsible for moving rows between segment instances required to process the query. The node will identify the method used to perform the motion operation for example, redistribute or broadcast motion.

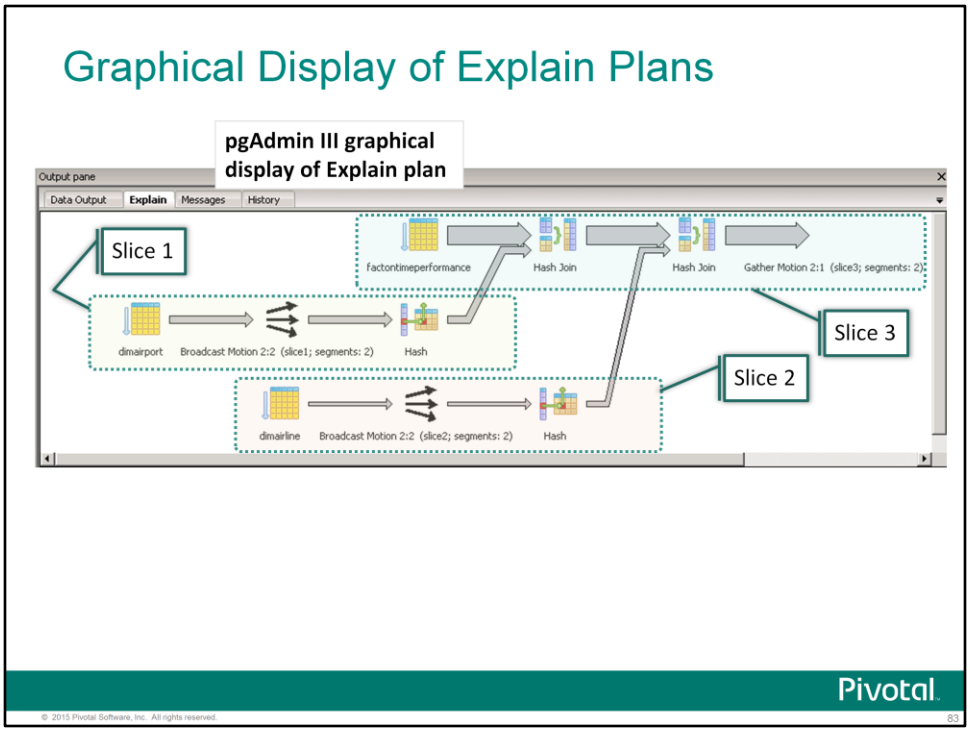
A gather motion is when segments send the resulting rows to the master host. The last operation for most query plans will be a gather motion.



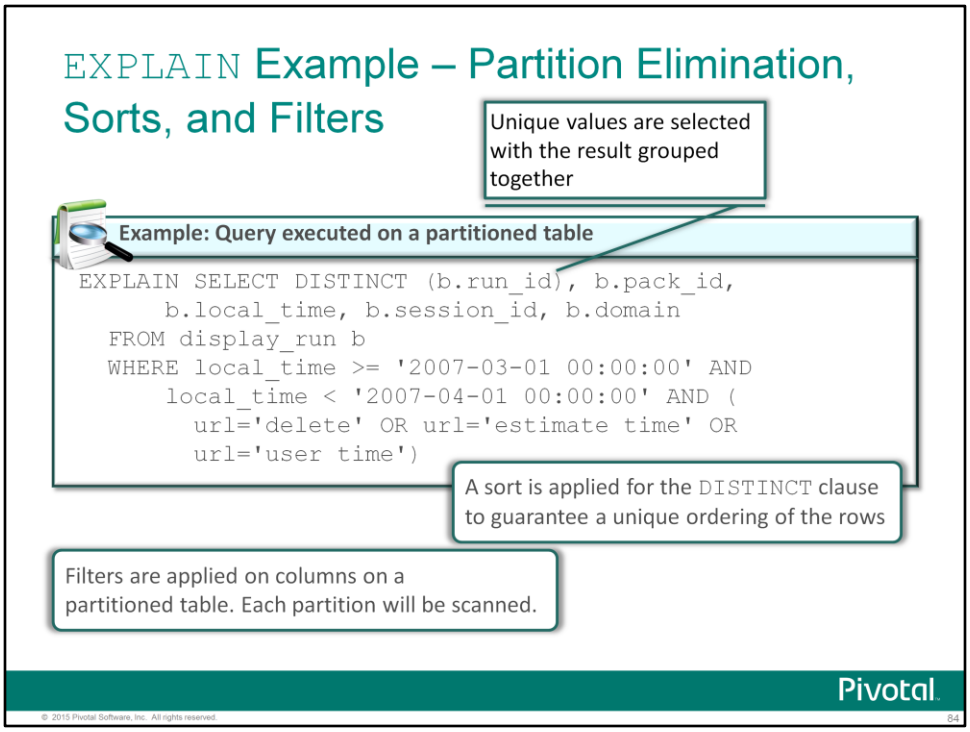
Explain output can be rather lengthy and more difficult to read directly in PSQL. You can save the output to text files from PSQL by using the `\o <filename>` meta-command in PSQL.

You can view the explain plan in Greenplum Command Center by accessing the Query Monitor tab and clicking on the **Show Query Plan** button for an in-flight or completed query. The Query Plan button lets you view the full query details, the query text, and the query plan for the executing, completed, or aborted query.

pgAdmin III provides both a textual display and a graphical display. You click the **Explain query** button in pgAdmin III to view the plan analysis. While Command Center provides the explain plan in text for an in-flight query, pgAdmin III does not actually execute the query, but instead performs the same behavior as using the `EXPLAIN` syntax as part of the query. You do not need to prepend the `EXPLAIN` clause to the query to obtain this output.

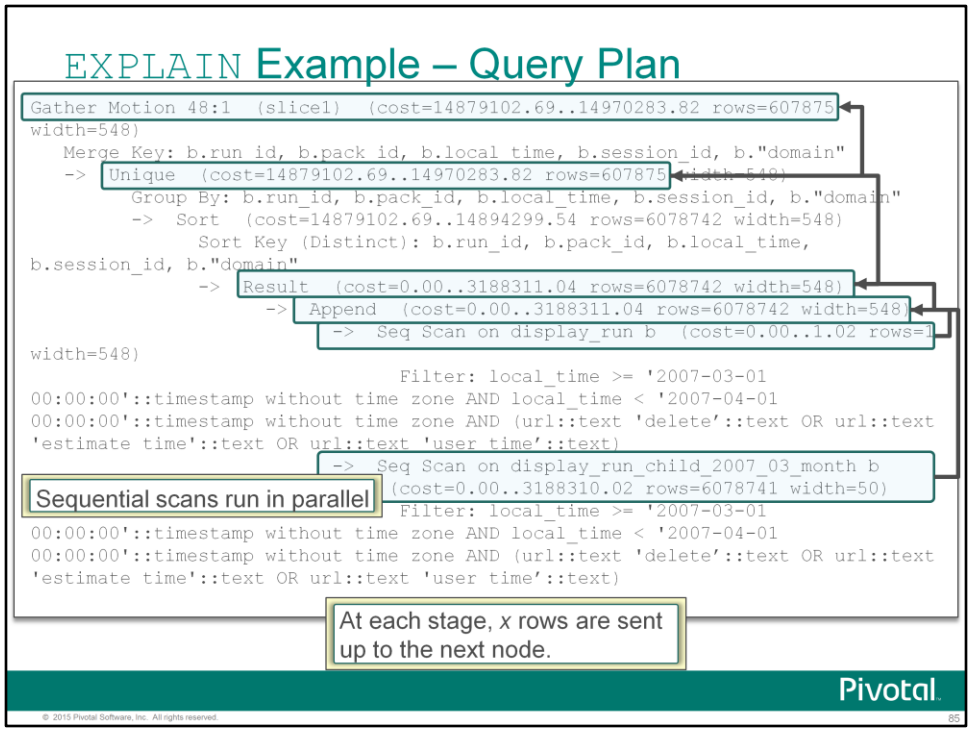


The graphical display shown here is similar to the graphical description provided earlier. Here, the plan is read from left to right and bottom to top. Slice 1 starts the operation with a scan on a table and performs a broadcast motion. In parallel, slice 2 starts a scan of another table and performs a broadcast motion of that table to the other segments to complete the operation. Once complete, the results are fed to another slice where another table is scanned and an hash join is performed between the results of slice 2 and slice 3. The results of slice 1 are hash joined with the results of slice 3 and a gather motion is then performed.



Let us examine the query shown on the slide. The query shown on the slide was executed against a partitioned table, aliased `b`. There are several restrictions applied to the query, which includes a restriction on the `local_time` column and the `url` column.

The corresponding query plan is displayed on the next slide.

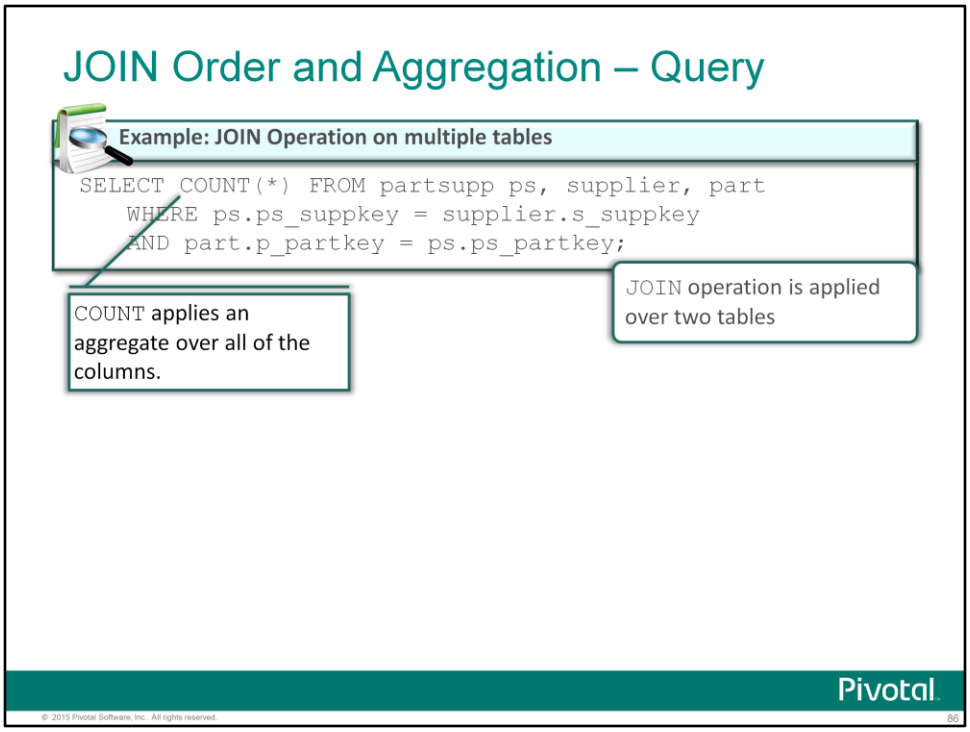


The example query plan displayed consists of a tree plan with multiple nodes as indicated by the arrows symbol (->). Each node is executed in parallel across all segment instances. In the examples used, some elements of the plans may be omitted for readability.

As query plans are analyzed it is important to focus on the query plan elements that are relevant to improving query performance. Only those elements relevant for the end user will be discussed and analyzed.

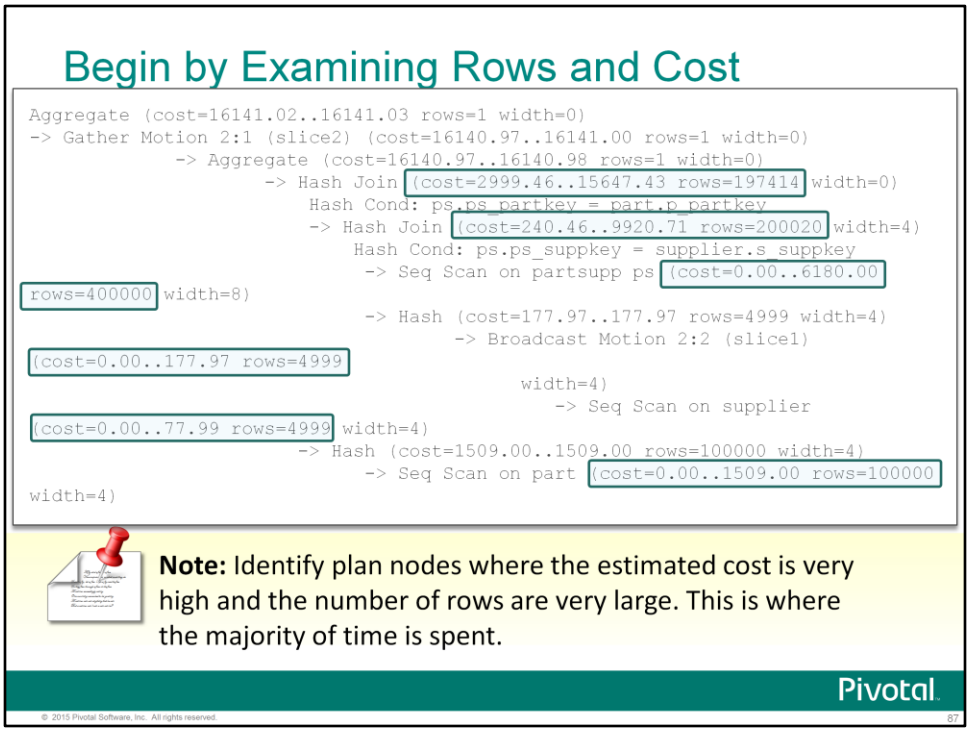
Read the query plan from the bottom up:

- The query displayed begins with a sequential scan of the `display_run_child_2007_03_month` table.
- The `WHERE` clause is applied as a filter condition.
- The scan operation checks the condition for each row it scans, and outputs 6078741 rows that pass the condition.
- The sequential scan operation cost began at 0.00 to find the first row and 3188310.02 0.255 to scan all rows.
- A second sequential scan of the `display_run_b` table results in one row.
- The results of the scan operations are passed up to a sort operation which is required in order to perform the group operation.
- The output of the sort operation will pass 6078742 rows to the next plan node to perform a GROUP BY resulting in 607875 rows.
- Lastly a gather motion operation is performed where the segment instances send their rows to one of the segment instances.



In this example, multiple JOIN operations are performed on several tables. The result is an aggregation, in the form of a count, on the resulting rows.

Let us examine the query plan for this query.

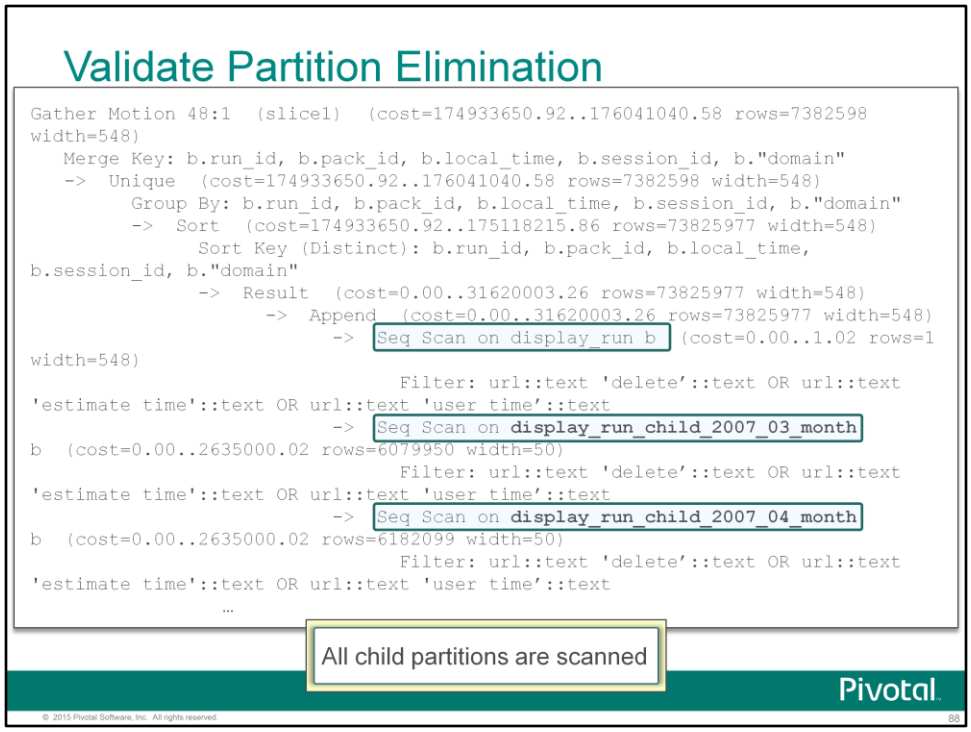


To analyze query plans and identify opportunities for query improvement:

- Begin by examining the rows and cost for the plan nodes.
- Identify plan nodes where the estimated number of rows are very large.
- Identify plan nodes where the estimated cost to perform the operation is very high.
- Determine if the estimated number of rows seem reasonable and the cost relative to the number of rows for the operation performed.
- Ensure that database statistics are up to date. High quality, up to date database statistics is required for the query planner's estimates.

In the example displayed:

- A sequential scan of the `supplier` table is performed.
- This is followed by a sequential scan of the `partsupp` table.
- To perform the hash join of the `supplier` and `partsupp` tables, a broadcast of the supplier table is performed. The broadcast motion notion of 2:2 indicates that there are two segment instances that will broadcast its rows in the `supplier` table to the other segment instances.
- The final gather motion is when segment instances send its rows to the master host. In this example there are two segment instances sending to one master instance (`2:1`).



If using table partitioning, it is important to validate that partitions are being eliminated or pruned. Only the partitions that contain data to satisfy the query should be scanned. The `EXPLAIN ANALYZE` output will display the child partitions that will be scanned.

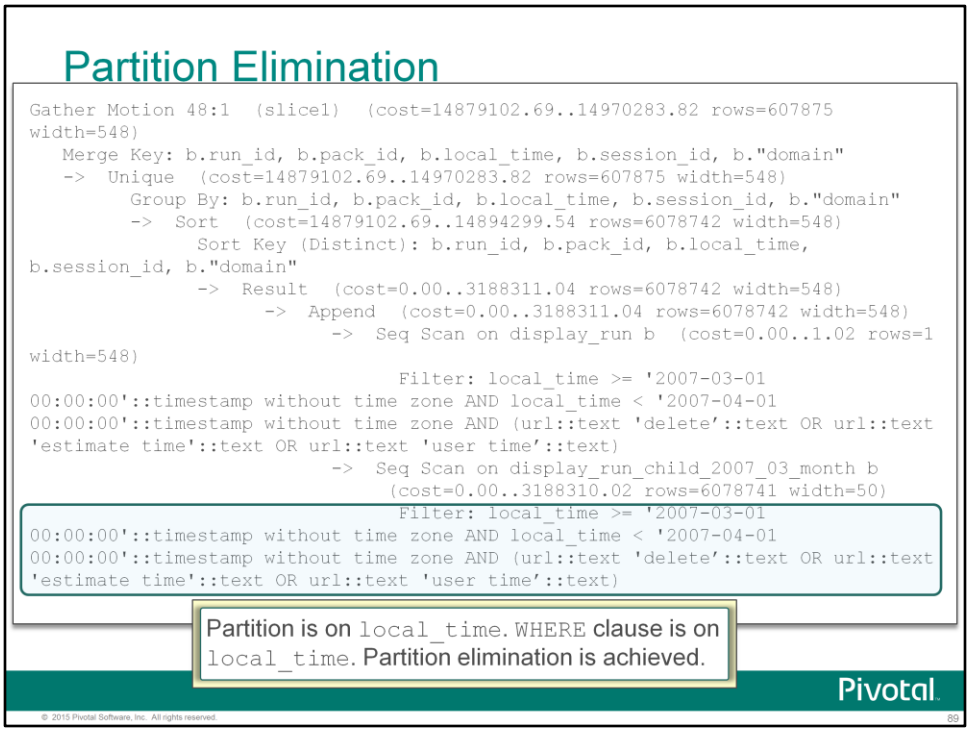
In the example displayed, all ten child partitions will be scanned to satisfy the query. Note that only two partitions are displayed for brevity. A sequential scan is performed for the `display_run` parent table, but keep in mind the parent table contains no data.

It is important to remember that to eliminate partitions, the partitioning criteria, `CHECK` clause, for the child tables is the same criteria used in the query predicate, `WHERE` clause. If the query access (`WHERE` clause) does not match the partitioning definition, the benefit of partition elimination is not achieved.

In the example displayed, the `display_run` table was partitioned on `local_time` and was not used in the query predicate.

The SQL for the query displayed is as follows:

```
SELECT DISTINCT run_id, pack_id, local_time, session_id,
domain
FROM display_run b
WHERE (url LIKE '%.delete%' OR
     url LIKE '%.estimated time%' OR
     url LIKE '%.user time%' );
```



To achieve partition elimination, the query predicate must be the same as the partitioning criteria.

It is also important to note that to eliminate partitions, the `WHERE` clause must contain an explicit value. The `WHERE` clause can not contain a subquery if partitions are to be eliminated.

For example, `WHERE date = '1/30/2008'` is an explicit value. However, `WHERE date IN (SELECT…. WHERE)` is an example of a subquery.

Building upon the same example, the `display_run` table was partitioned on `local_time` and the `WHERE` clause contained an explicit value that eliminated all but one partition. This partition contained the data required to satisfy the query.

The SQL for the query displayed is as follows:

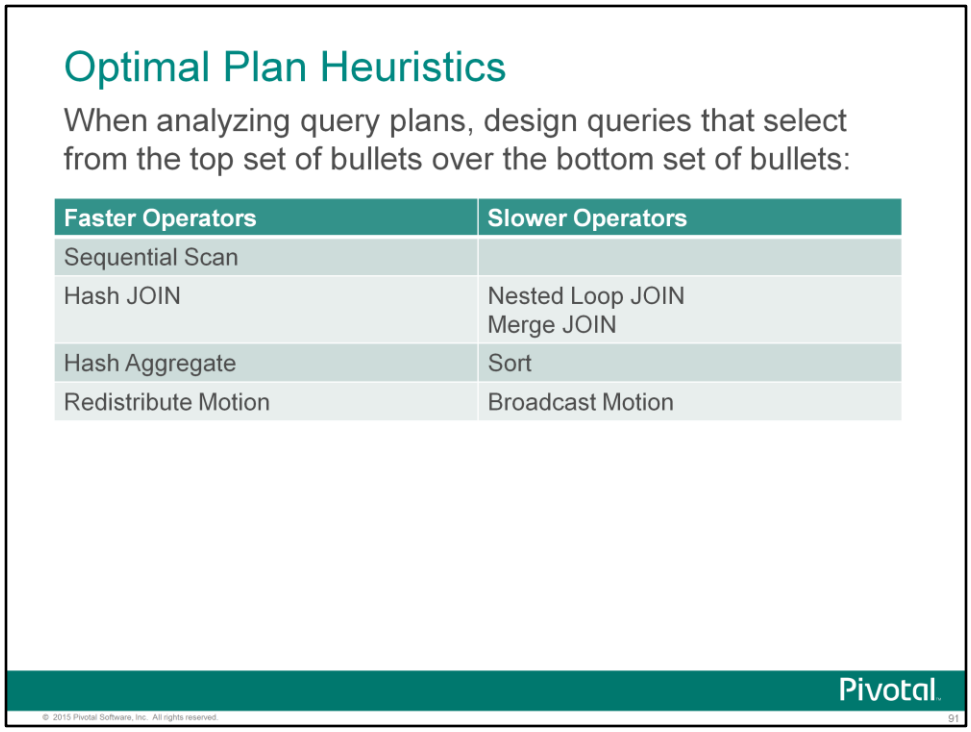
```
SELECT DISTINCT run_id, pack_id,
local_time,session_id,  domain
FROM display_run b
WHERE (b.local_time >= '2007-03-01 00:00:00'::timestamp
without time zone AND b.local_time < '2007-04-01
00:00:00'::timestamp without time zone)
AND (url LIKE '%.delete%' OR url LIKE '%.estimated time%'
OR url LIKE '%.user time%' );
```

Partition Elimination (Cont)

Note: To eliminate partitions, the `WHERE` clause must be the same as the partition criteria AND must contain an explicit value (no subquery).

Pivotal

Partition Elimination (Continued)



The second set of bullets shown on the slide represents actions that are considered slower than the top set of bullets.

The `EXPLAIN ANALYZE` contains one line for each node in the plan tree. That line represents a single operation, such as scan, join, aggregation, or sort operation.

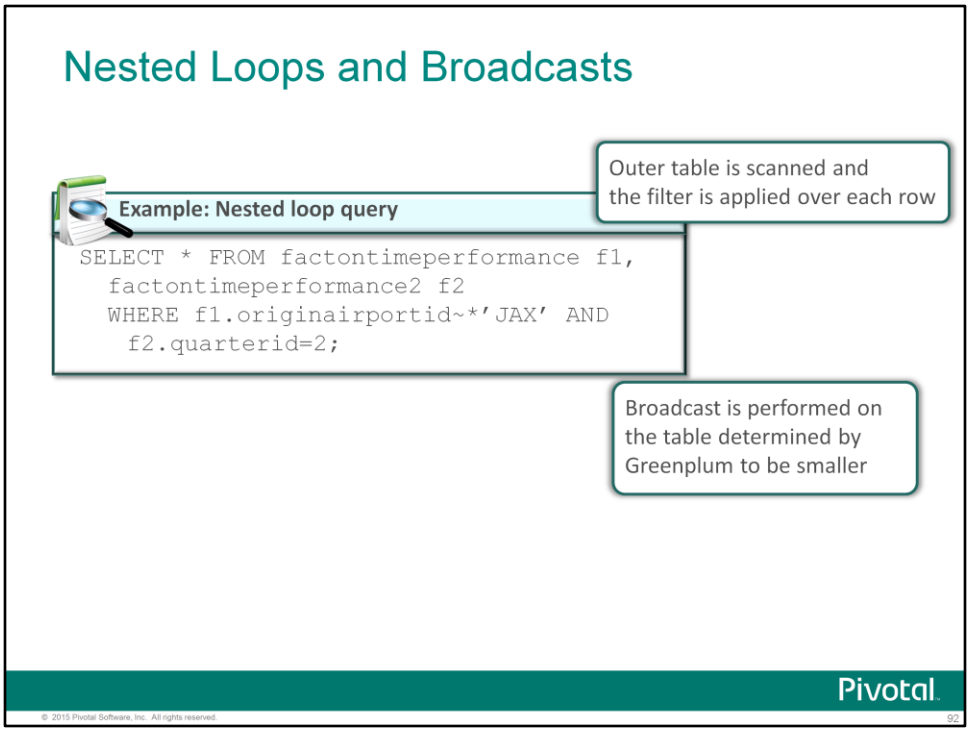
The node will identify the method used to perform the operation. For example:

- A scan operation may perform a sequential scan or index scan.
- A join operation may perform a hash join or nested loop join. A sort operation may perform a hash aggregation or sort.

When analyzing query plans, identify the nodes with slow operators including nested loop join, merge join, sort and broadcast motion operators. Begin by examining the rows and cost for the plan nodes.

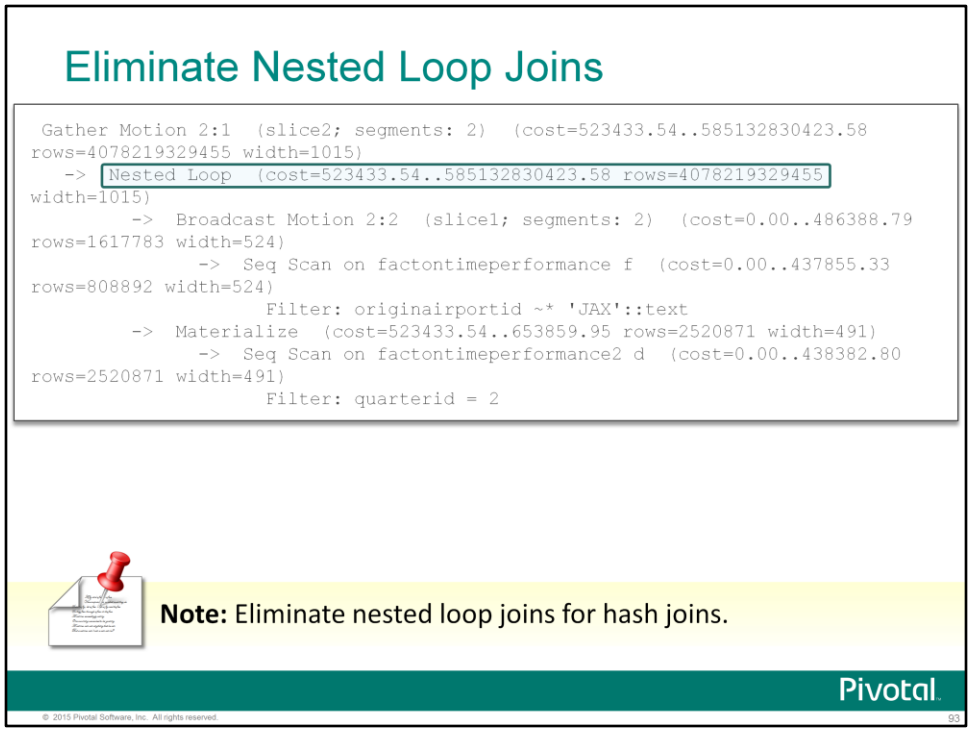
For these plan nodes, review the estimated number of rows and cost. If the estimated number of rows are large and the cost to perform the operation is high, then there are opportunities to improve query performance.

A sequential scan in an MPP environment is an efficient method of reading data from disk as each segment instance works independently in the shared nothing environment. This is unlike a traditional SMP environment where queries compete for resources. Data warehouse environments typically access large volumes of data for analysis. However, an index scan may be beneficial when the selectivity of queries are high accessing a single or few rows.



In this example, a nested loop performs a scan on the outer table and iterates over each row of the outer table, applying a filter, if necessary, to those rows.

The smaller table is broadcast across segments, if the JOIN is not on the distribution key of both tables.



A nested loop join requires:

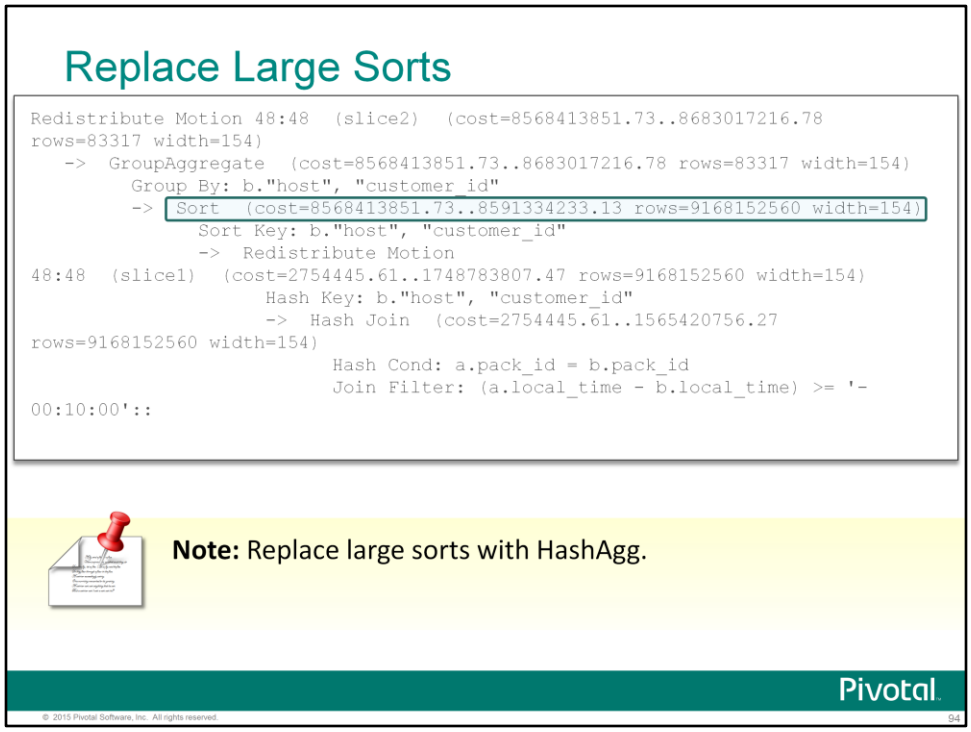
- Each outer tuple is compared with each inner tuple that might join to it.
- The broadcast of one of the tables so that all rows in one table can be compared to all rows in the other table. The performance is therefore affected by the size of the tables.

Nested loop joins may perform well for small tables. There are performance implications when joining two large tables.

In the example displayed, the estimated cost to perform the join is very high relative to the estimated number of rows.

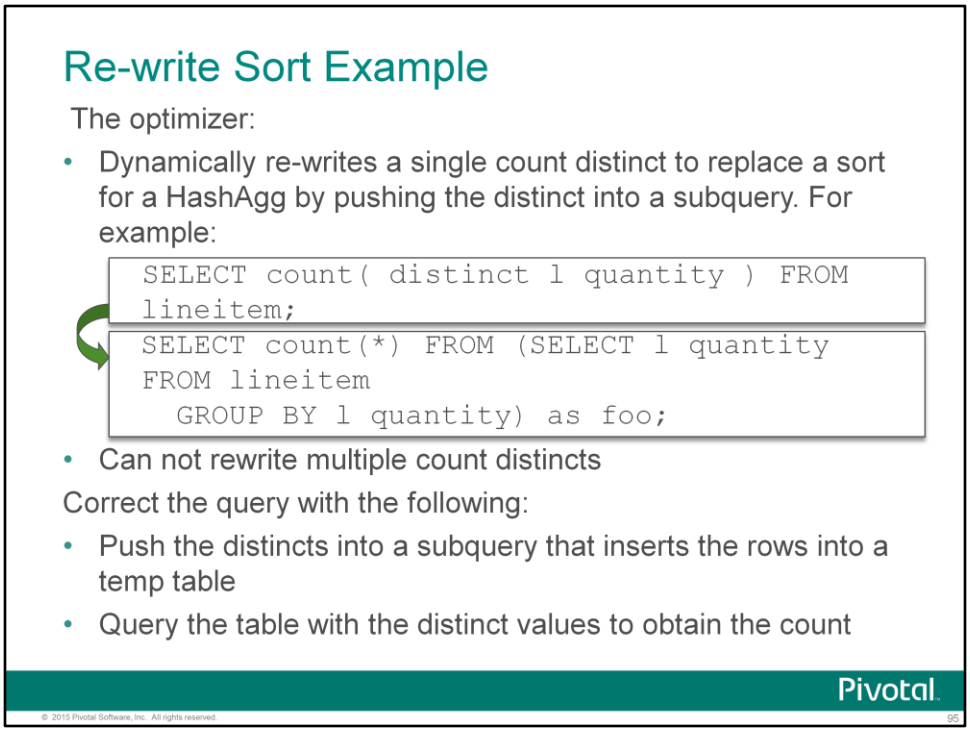
For plan nodes that contain a nested loop join operator, validate the SQL and ensure that the results are what is intended. Poorly written or incorrect SQL is often times the primary offender affecting query performance.

In addition, the `enable_nestloop` system configuration parameter should be set to `off` to favor a hash join over a nested loop join when possible. By default, the parameter is set to `off`.



Identify plan nodes where a sort or aggregate operation is performed. Hidden inside an aggregate operation is sort. If the sort or aggregate operation involves a large number of rows, there is opportunity to improve query performance. A HashAggregate operation is preferred over sort and aggregate operations when a large number of rows are required to be sorted.

Usually, a sort operation is chosen by the optimizer due to the SQL construct, that is, due to the way the SQL is written. Most sort operations can be replaced with a HashAgg if the query is rewritten.



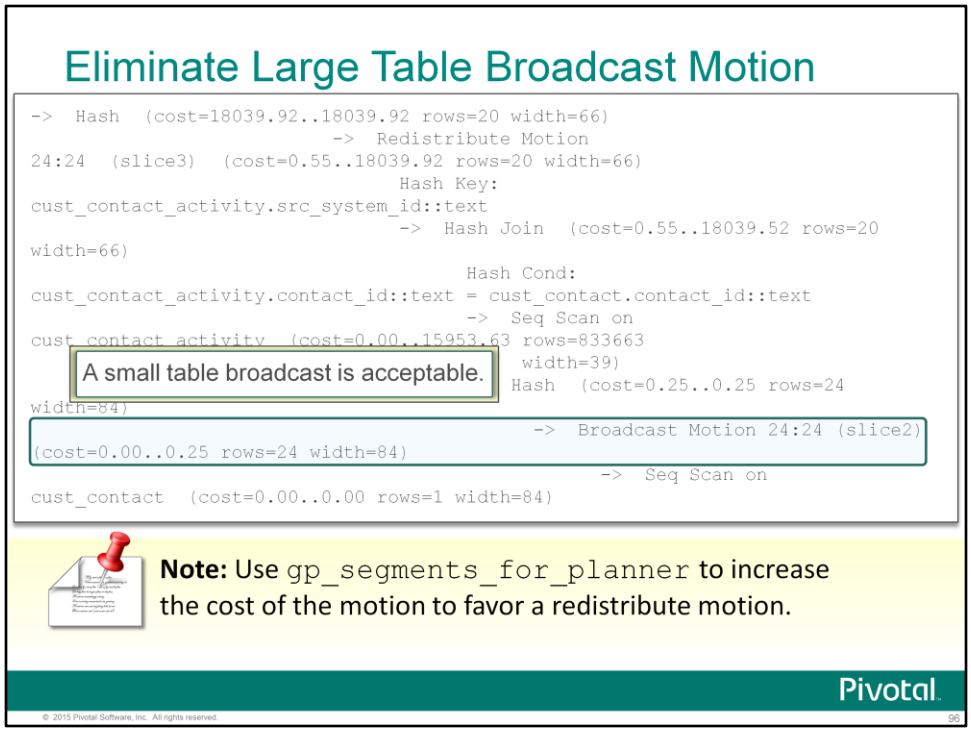
A sort is performed when the SQL involves a count distinct SQL construct.

The optimizer will dynamically re-write a query that contains a count distinct to obtain a HashAgg operation instead of a sort operation. The optimizer re-write basically pushes the distinct into a subquery.

However, the optimizer can not re-write an query that contains multiple count distincts so a sort operation will be performed.

To replace the sort operation for a HashAgg operation the query will need to be rewritten. The re-write is similar to the example displayed, except a temp table will be used to store the distinct values.

In this scenario, re-write the query to push the distinct into a subquery that inserts the resulting rows into a temp table. Then select from the temp table that contains the distinct values to obtain the count.



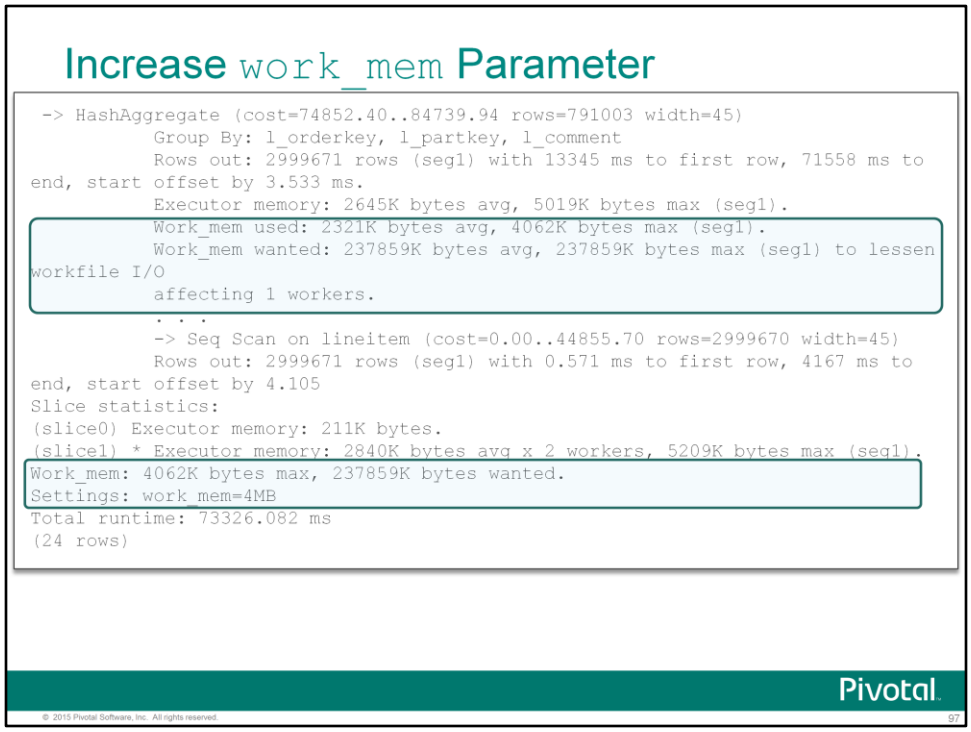
In some cases, a broadcast motion will be performed rather than a redistribute motion.

In a broadcast motion, every segment instance performs a broadcast of 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. A broadcast motion is not acceptable for large tables. When analyzing query plans, identify broadcast motion nodes. If the number of rows are large, then consider using the `gp_segments_for_planner` parameter to increase the cost of the motion.

The `gp_segments_for_planner` sets the number of primary segment instances for the planner to assume in its cost and size estimates. If `gp_segments_for_planner` is set to zero, then the value used is the actual number of primary segments.

This variable affects the planner's estimates of the number of rows handled by each sending and receiving process in motion operators. Increasing the number of primary segments increases the cost of the motion, thereby favoring a redistribute motion over a broadcast motion. For example setting `gp_segments_for_planner = 100000` tells the planner that there are 100000 segments.



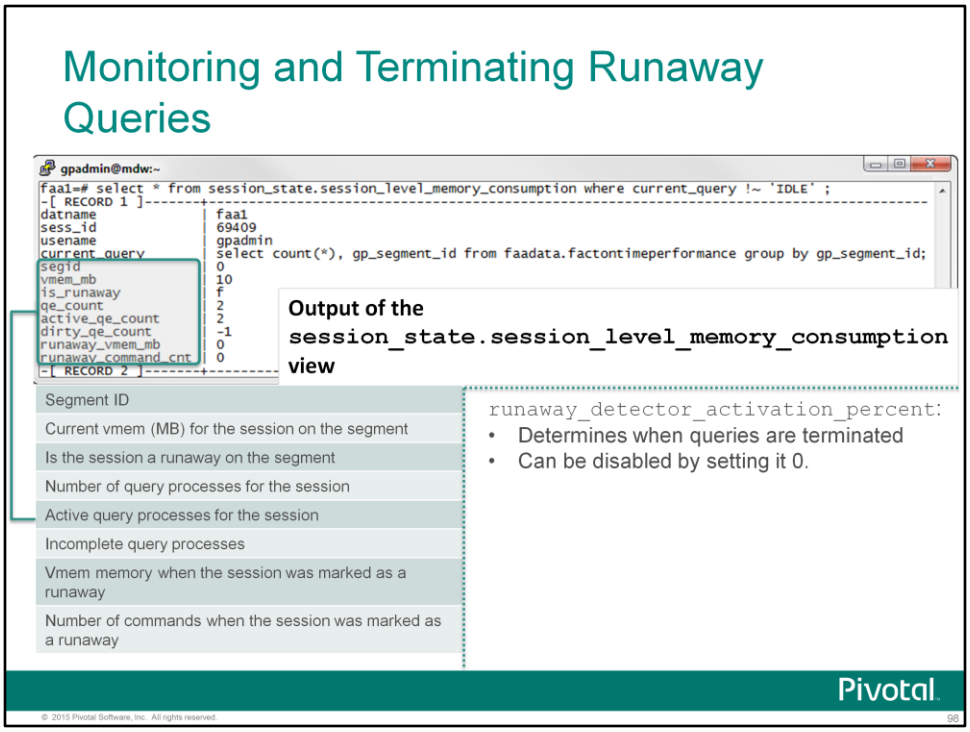
Identify plan nodes where the memory used versus the memory wanted for the specified operation is significant. At the bottom of the plan file, the `work_mem` parameter setting is also displayed.

The `work_mem` variable specifies the amount of memory to be used by internal sort operations and hash tables before spilling to file system. If there are several sort or hash operations running in parallel for a given query, each operation will be permitted to use the memory specified before spilling to file system.

If the memory used versus the memory wanted for a plan node operation is significant, then increase `work_mem` for the query using the SQL `SET` statement to give the query additional memory.

Arbitrarily setting `work_mem` to a very large number for a given query may:

- Decrease query performance due to the overhead of allocating and managing memory.
- Impact on other concurrent queries with sort and hash operations.

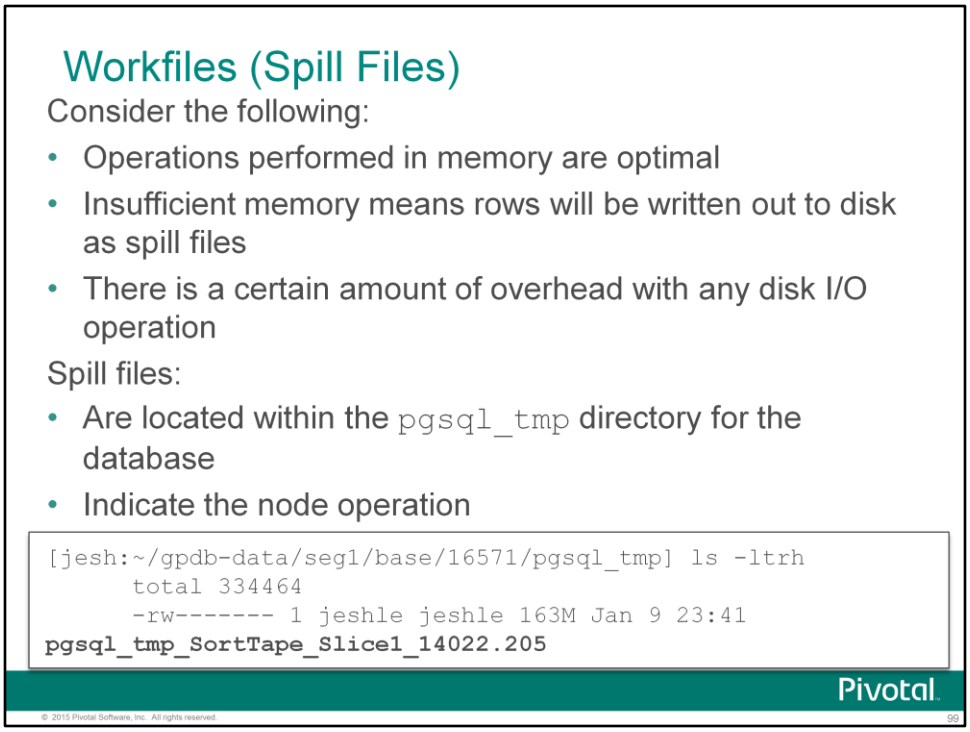


While manipulating the `work_mem` parameter can work for individual queries, the workload resource management tool lets you manage queries for other users. If not defined appropriately, a query can potentially consume large amounts of memory and block other sessions from executing their queries.

The `session_level_memory_consumption` view, defined in the `session_state` schema, provides information on the amount of memory that a session is using to execute its queries. The view provides a view of the query consumption at the segment level for the associated query. The segment listing includes the master.

The view is installed per database using the script, `$GPHOME/share/postgresql/contrib/gp_session_state.sql`.

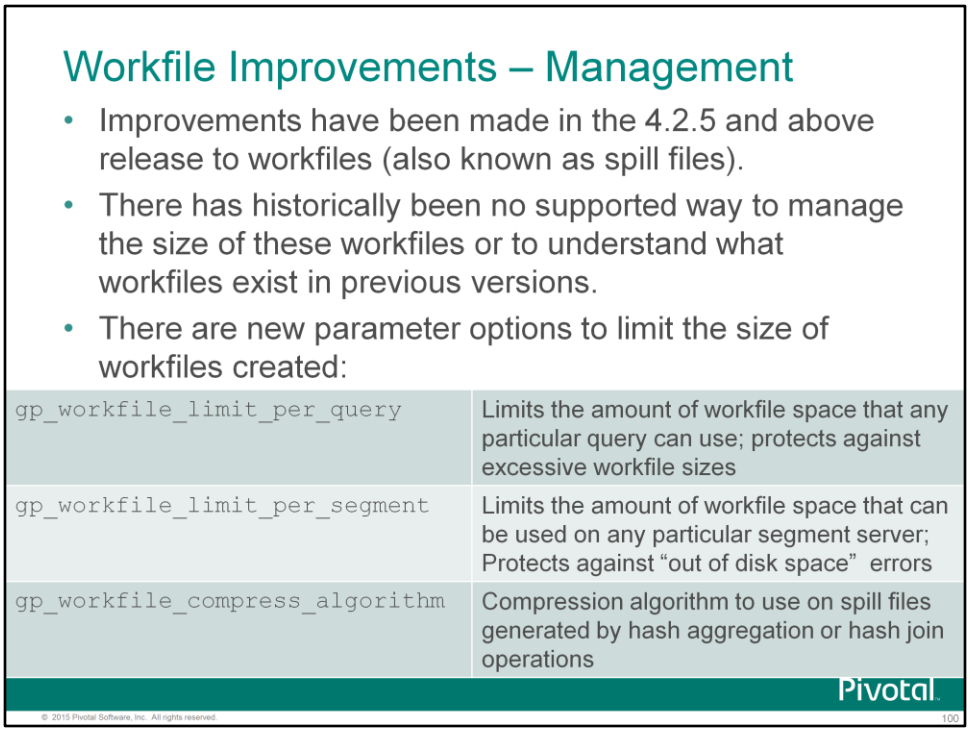
In addition to providing information on the amount of memory the query is consuming with the `vmem_mb` field, the `is_runaway` field provides an indication of whether or not Greenplum considers the session a runaway. A session is considered a runaway when the amount of vmem memory it is consuming exceeds the value defined in the `runaway_detector_activation_percent` parameter. Greenplum will terminate the query, starting with the largest consumer. By default, this parameter is set to 90%. It can be disabled by changing the value to 0.



Plan node operations performed in memory is optimal for query performance. Join, aggregate, and sort operations that can not be performed in memory will impact query performance.

For example, a hash join performed in memory is the optimal method for joining tables. However, if the build step of a hash join involves a very large number of rows, there may not be sufficient memory. In this scenario the build rows will be written out to disk as spill files. With any disk I/O operation there is a certain amount of overhead.

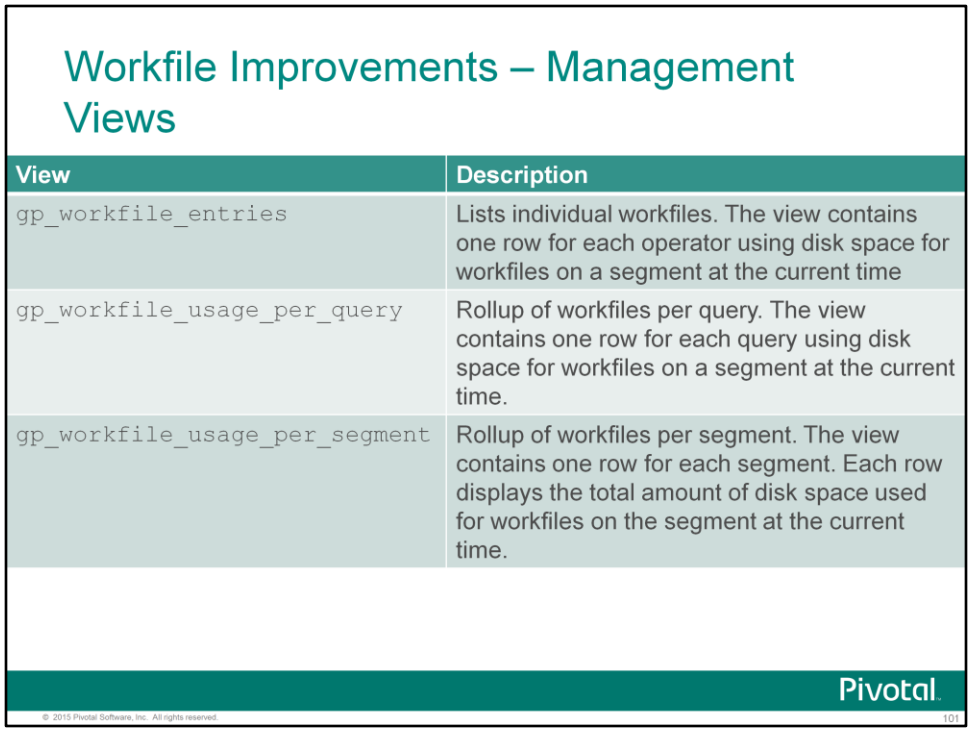
Inside the directory, `/data_directory/base`, you will find several directories whose names represent the OIDs of the databases within a segment instance. Within any of these directories, there is a `pgsql_temp` directory where the spill files are located. The filenames will indicate the node operation.



These following parameters help to control how large workfiles can get for queries that will generate them and how to handle the size of these spill files. This effectively helps you to control queries that process very large data sets that cannot be perform totally in memory. This can be used to help protect against bad user queries that use excessive workfile sizes and can also be used to help protect against "out of disk space" conditions.

The workfile parameters introduced are:

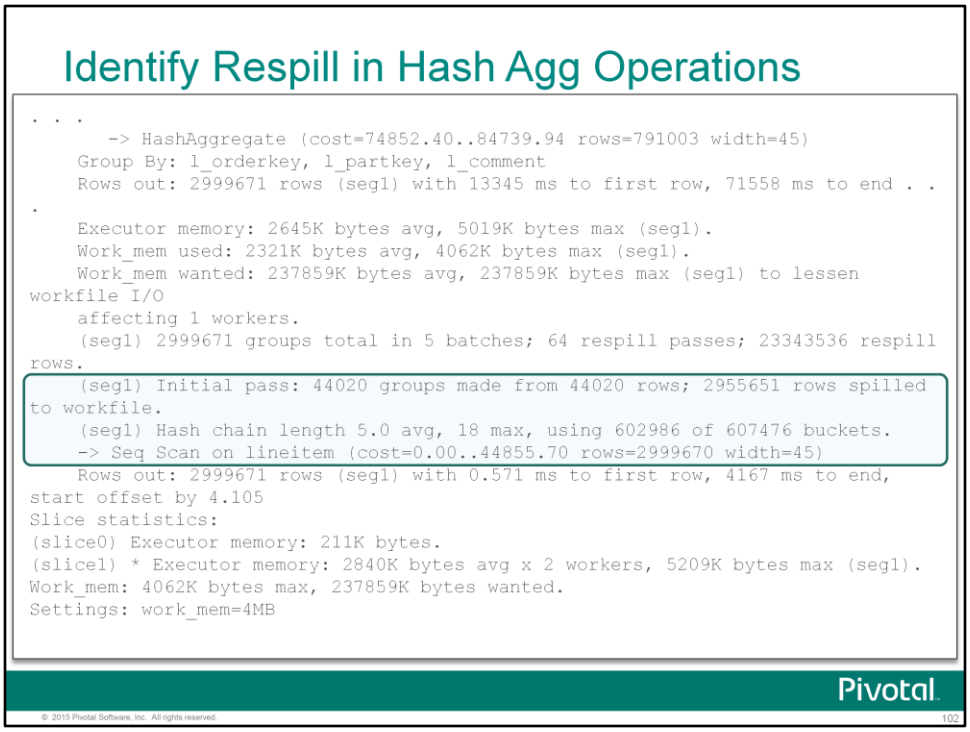
- `gp_workfile_limit_per_query` – This sets the maximum disk size (in kilobytes) an individual query is allowed to use for creating temporary spill files at each segment. The default value is 0, which means a limit is not enforced. This parameter can be used to protect the environment against queries that can consume excessive workfile sizes for execution.
- `gp_workfile_limit_per_segment` – Sets the maximum total disk size (in kilobytes) that all running queries are allowed to use for creating temporary spill files at each segment. The default value is 0, which means a limit is not enforced.
- `gp_workfile_compress_algorithm` – This parameter is used to define the algorithm that can be used to compress spill files generated by either a hash aggregation or hash join. It can be set to `zlib` or disabled with `none`, the default setting.



The `gp_workfile*` views are available in the gp_toolkit schema. These views are:

- `gp_workfile_entries` – This view contains one row for each operator that uses disk space for workfiles at the current time. If an operator uses disk space for workfiles on multiple segments, the view contains one row for each segment where that occurs. If multiple operators in a query use disk space for workfiles, the view contains one row for each operator using disk space for workfiles for each segment where that occurs.
- `gp_workfile_usage_per_query` – This view contains one row for each query that uses disk space for workfiles at the current time. If a query uses disk space for workfiles on multiple segments, the view contains one row for each segment where that occurs.
- `gp_workfile_usage_per_segment` – This view contains one row for each segment. Each row displays the total amount of disk space used for workfiles on the segment at the current time.

This information can be used for troubleshooting and tuning queries. The information in the views can also be used to specify the values for the Greenplum Database configuration parameters `gp_workfile_limit_per_query` and `gp_workfile_limit_per_segment`.



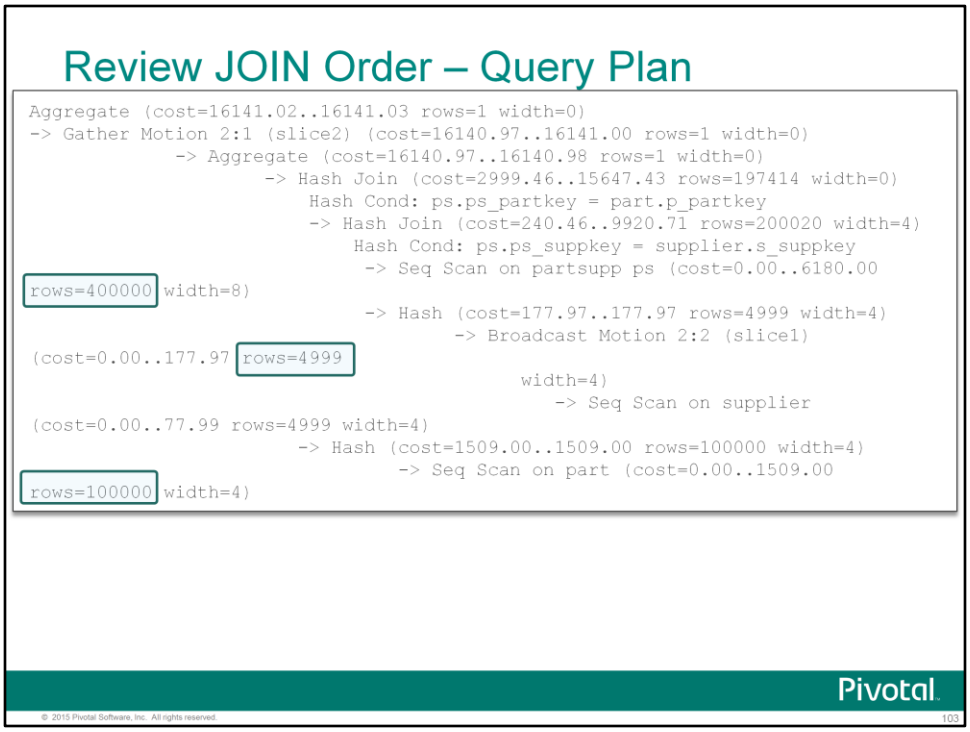
A HashAgg attempts to build the hash table in memory. If the hash table size exceeds the available memory, it will start to spill, that is, write rows out to temporary files, called batches.

Later, the HashAgg will process batch by batch. If the batch still cannot fit in memory, each batch will spill to more batches. This is respill condition.

Identify HashAggregate node plans with respill conditions. If the number of respill passes and the number of respill rows are large, than there is opportunity to increase query performance.

In order to minimize respilling, the HashAggregate node uses the planners estimates to subdivide the data into batches and will spill a separate data file per batch.

If the estimated number of batches is smaller than the real optimal value, then respilling of data will be required when processing batches. This respilling behavior can be quite expensive if the difference between the estimated and optimal batch value is great.



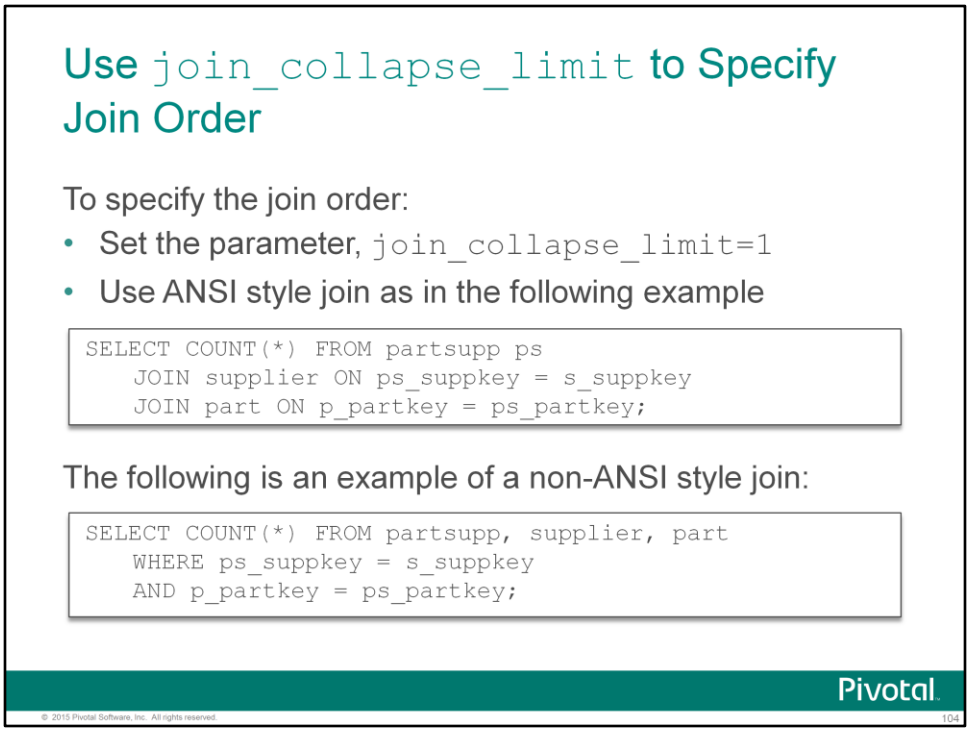
For a join, the execution order to build on the smaller tables or hash join result and probe with larger tables.

Review:

- The execution order of the query plan tree.
- The estimated number of rows.

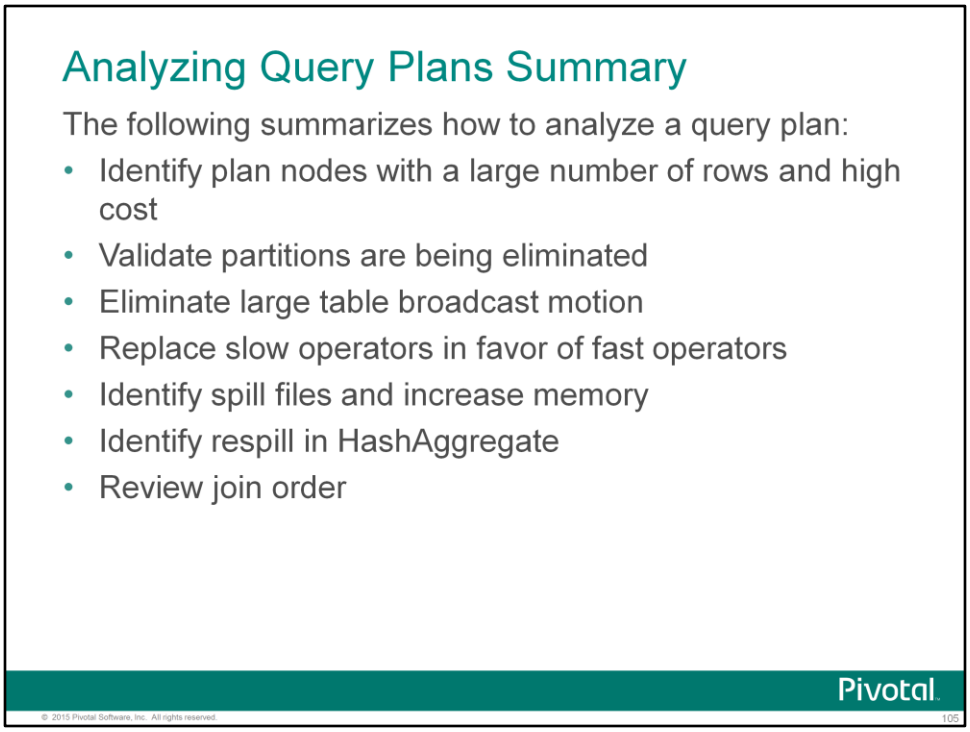
Optimally the largest table is used for the final join or probe to reduce the number of rows being passed up the tree to the topmost plan nodes.

If the analysis reveals that the order of execution builds and/or probes on the largest table first ensure that database statistics are up to date.



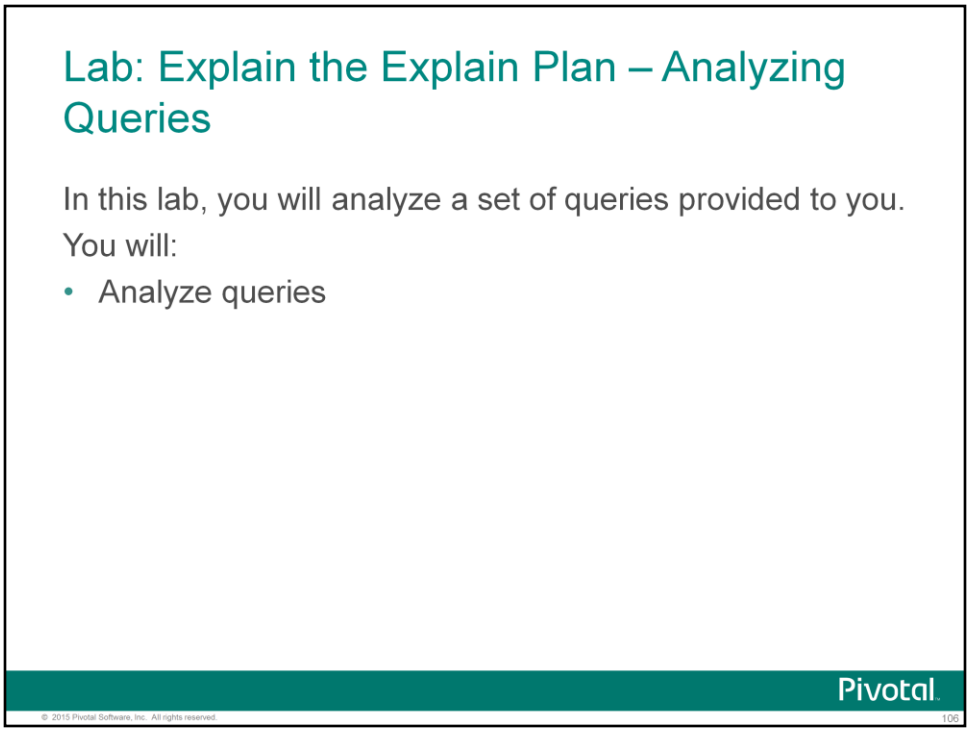
If the analysis reveals that the order of execution builds and/or probes on the largest table first and database statistics are up to date, consider using the Global User Configuration (GUC) variable, also known as the configuration parameter, `join_collapse_limit`, to specify the join order for ANSI style joins. This parameter will not affect non-ANSI style joins.

Set `join_collapse_limit = 1` to disable the join ordering as determined by the optimizer. It is then the user's responsibility to write the query in a way, using ANSI style joins, that sets the preferred join ordering.

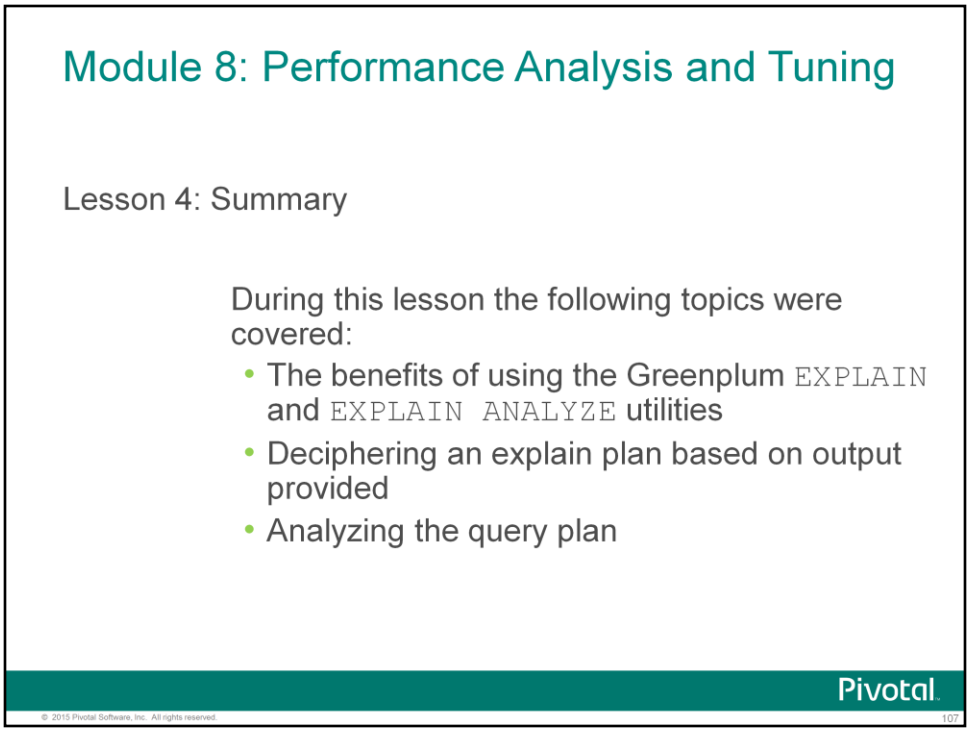


Analyzing `EXPLAIN` plan files is the best possible way to understand query execution and query performance:

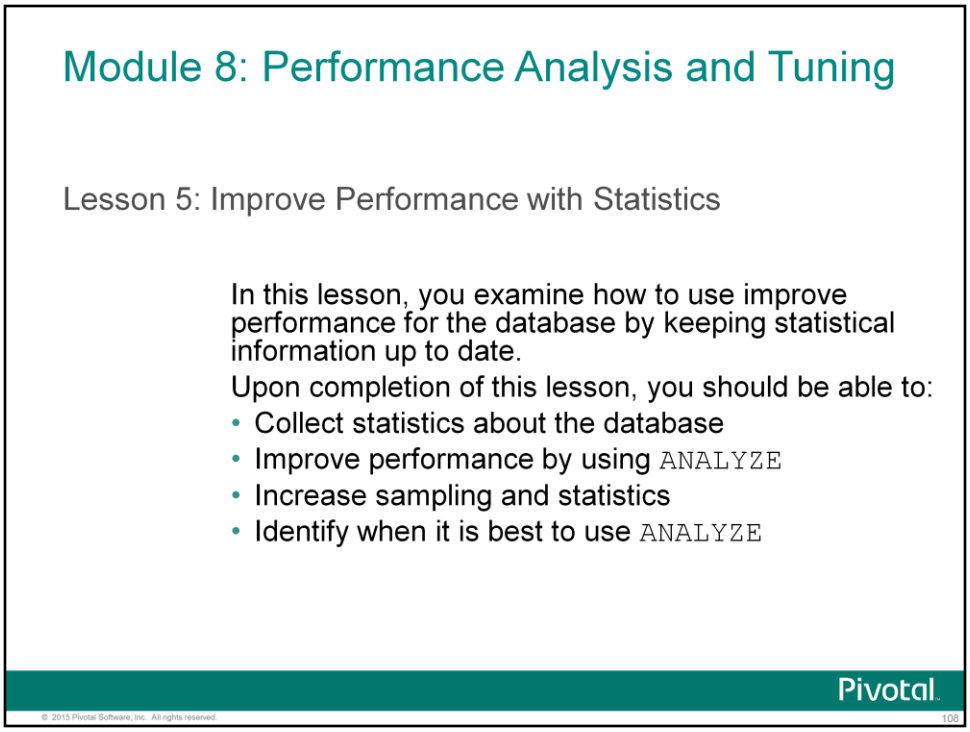
- Begin by reviewing the plan nodes where there majority of time is spent in executing the query. For these plan nodes review the estimated number of rows and cost. Does the estimated number of rows and relative cost seem reasonable? If not, validate database statistics are up to date.
- Validate partitions are being eliminated. Only the partitions that contain data to satisfy the query should be scanned.
- Eliminate large table broadcast motion for redistribute motion.
- Replace slow operators for fast operators.
- Identify plan nodes where the memory used versus the memory wanted for the specified operation is significant. If the memory used versus the memory wanted for a plan node operation is significant then increase `work_mem` for the query.
- Review the execution order of the query plan tree. If the analysis reveals that the order of execution builds and/or probes on the largest table first ensure that database statistics are up to date.
- And most importantly review the SQL and ensure that the results are what is intended. Poorly written or incorrect SQL is often times the primary offender affecting query performance.



In this lab, you will analyze a set of queries provided to you.



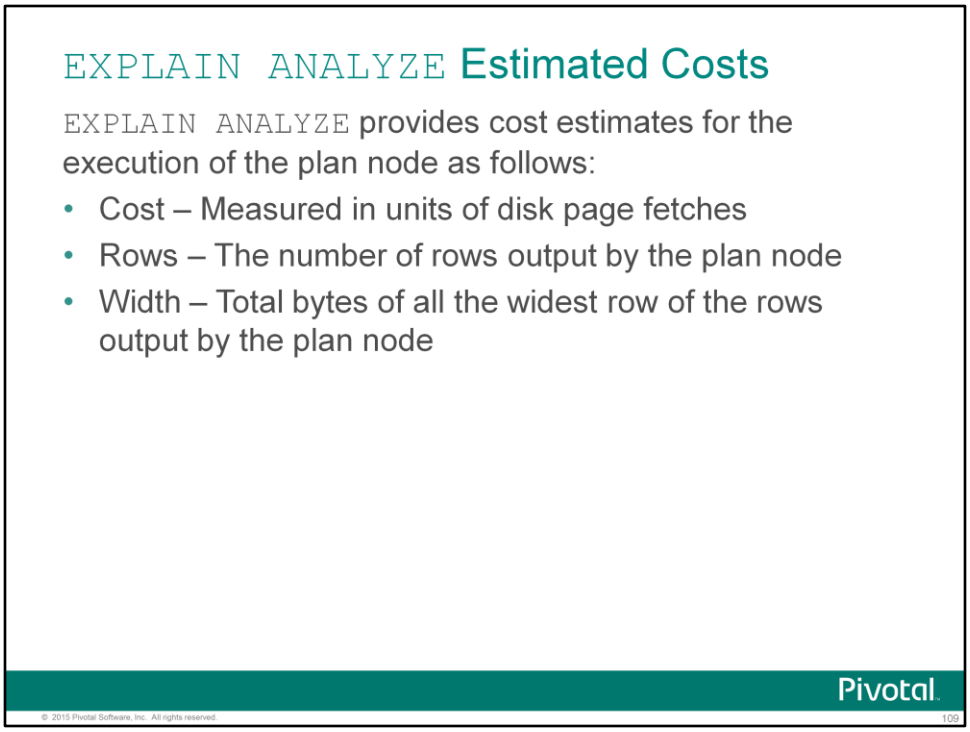
This lesson focused on the query plan and how to use the EXPLAIN and EXPLAIN ANALYZE syntax to not only view the query plan, but, as in the latter syntax, improve overall statistics and the performance of the query based on the optimizer's ability to more accurately predict what is needed. A description of the explain plan was provided along with examples of major nodes and operations to look for when analyzing the query plan.



Keeping statistical information up on tables to date directly impacts how the optimizer creates the query plan.

In this less, you will:

- Collect statistics about the database
- Improve performance by using `ANALYZE`
- Increase sampling and statistics
- Identify when it is best to use `ANALYZE`



Query plans display the node type along with the planner estimates including cost, rows and width for the execution of that plan node.

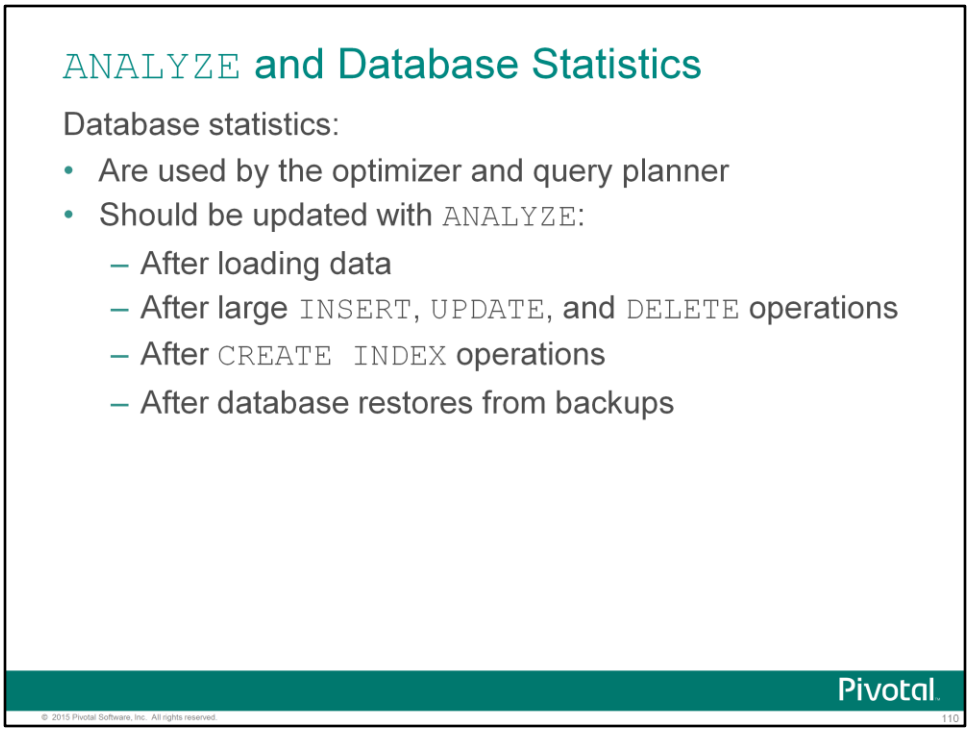
Cost is a relative measured in units of disk page fetches. The value 1.0 equals one sequential disk page read. The first estimate is the cost of getting the first row and the second is the total cost of getting all rows in milliseconds. The total cost assumes that all rows will be retrieved, which may not always be the case if using a `LIMIT` clause for example.

Rows are the total number of rows output by the plan node. This is usually less than the actual number of rows processed or scanned by the plan node, reflecting the estimated selectivity of any WHERE clause conditions.

The width is the total bytes of the widest row of all the rows output by the plan node.

It is important to note that the cost of an upper-level node in the plan tree includes the cost of all its child nodes. The topmost node of the plan has the estimated total execution cost for the plan.

The cost does not consider the time spent transmitting results to the client.



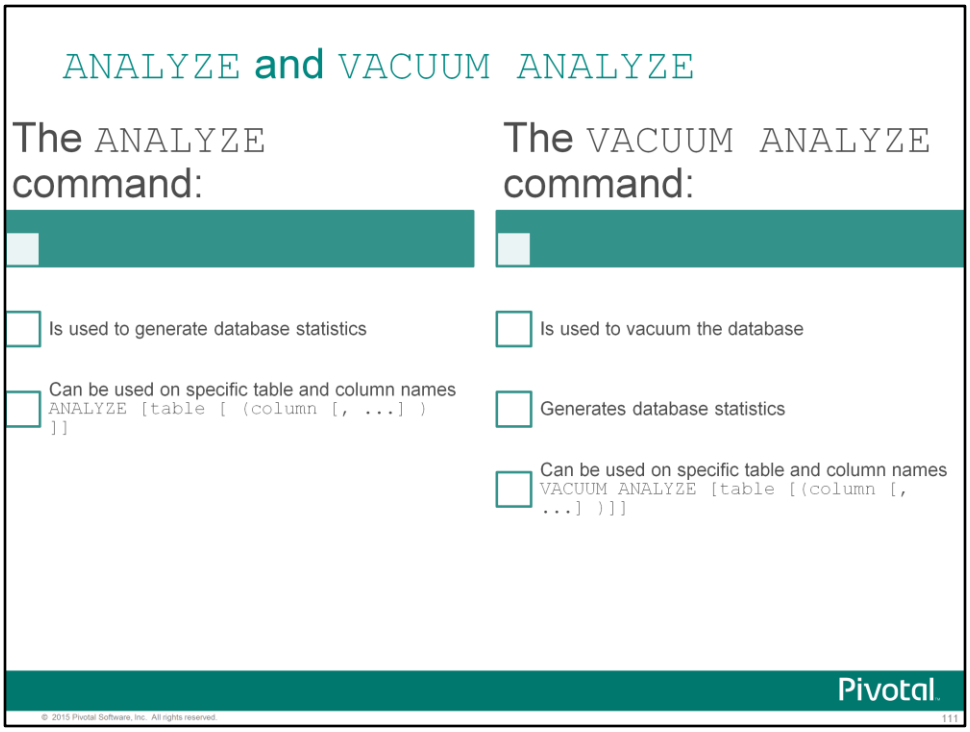
The Greenplum database uses a cost-based query planner that relies on database statistics. The ANALYZE command collects statistics about the database needed by the query planner to determine the most efficient way to execute a query.

It is very important that the database has up-to-date statistics for optimal query performance. As a database or system administrator this is the most important ongoing administrative database maintenance task.

It is highly recommended to run `ANALYZE`:

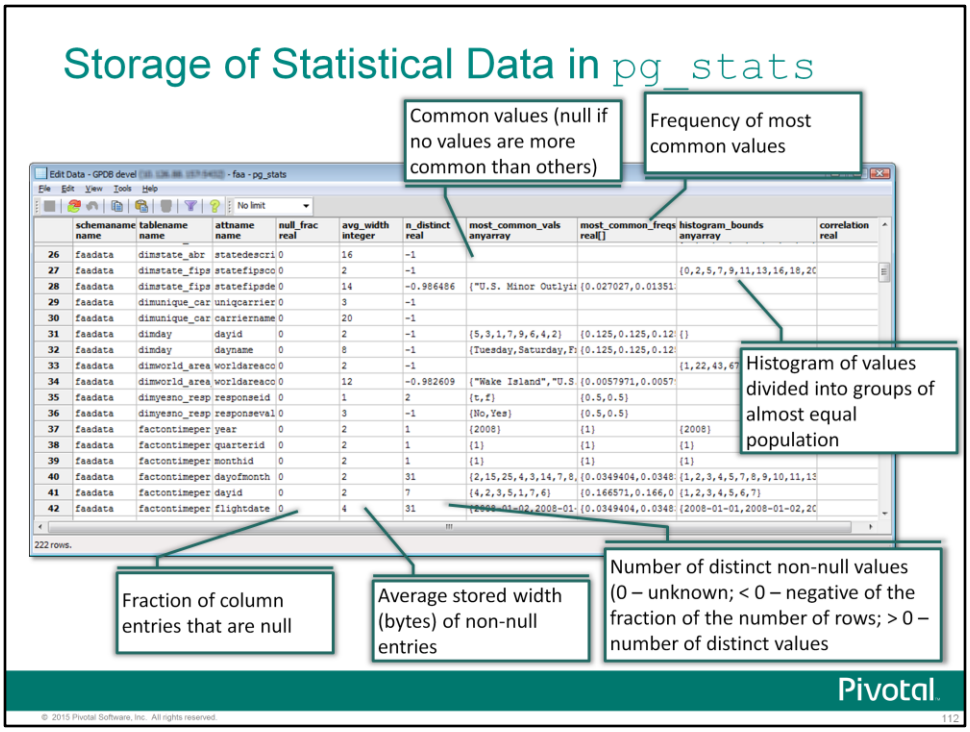
- After data loads
- After large `INSERT`, `UPDATE`, and `DELETE` operations
- After `CREATE INDEX` operations
- After completing a database restore from backups

The `pg_statistic` system catalog table stores statistical data about the contents of the database. Entries are created by `ANALYZE` and subsequently used by the query planner. There is one entry for each table column that has been analyzed. The `pg_statistic` system catalog also stores statistical data about the values of index expressions.



Use the ANALYZE command to generate database statistics. An ANALYZE may be run on the entire table or on specified columns in a table. In large data warehouse environments it may not be feasible to run ANALYZE on an entire database or table due to time constraints. Specifying particular columns to analyze provides the flexibility to generate optimal database statistics for large databases.

Use VACUUM ANALYZE to vacuum the database and generate database statistics. VACUUM ANALYZE performs a VACUUM and then an ANALYZE for each selected table. This is a convenient combination for routine database maintenance scripts.



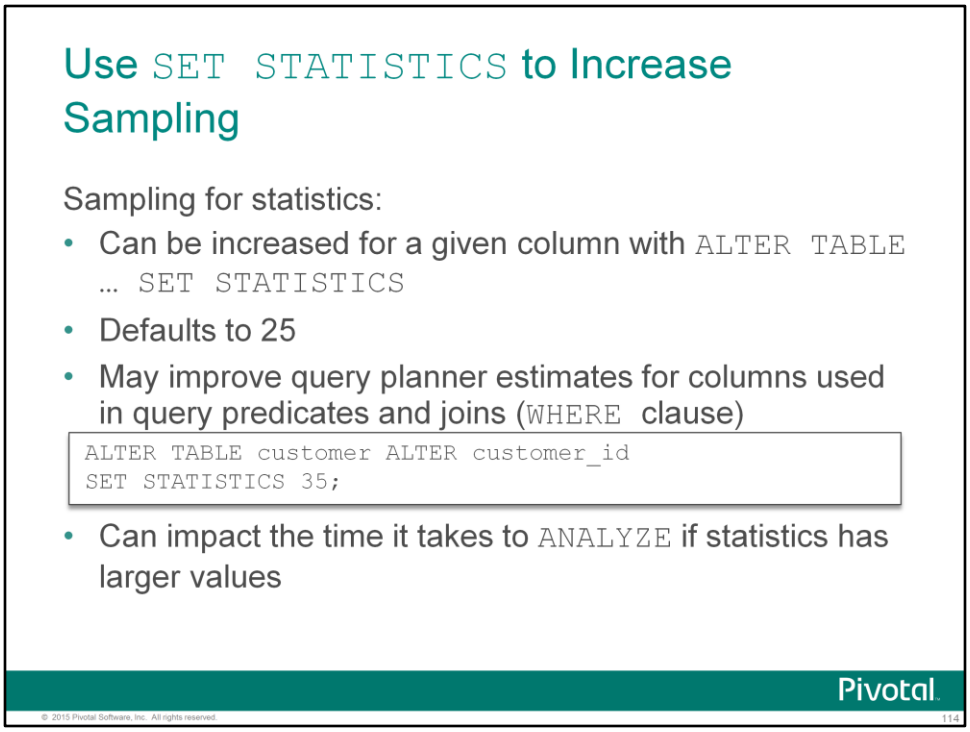
Statistical information used by the optimizer is stored in the `pg_catalog.pg_statistic` table. This table stores statistical data for each entry of a table column that has been analyzed with the `ANALYZE` command. The `pg_statistic` table is accessible to database superusers as it provides information on all tables within the database. The `pg_stats` view is a publicly readable view which gives the current user information on tables they have access to. This protects sensitive information by only displaying information on tables the user has read access to.

The statistical columnar data is as follows:

- `null_frac_real` – This column displays the fraction of column entries that are null.
- `avg_width` – The average width of the non-null values of the column is displayed.

Storage of Statistical Data in pg_stats

- `n_distinct` – The number of distinct non-null values is displayed. If the value displayed is 0, the number of distinct values is unknown. If it is less than 0, the value displayed is the negative of the number distinct values divided by the number of rows in the table. The negative value indicates that `ANALYZE` believes the number of distinct values will increase as the number of rows increases. A positive value indicates the number of possible values for the column.
- `most_common_vals` – The list of most common values is displayed in this column. This column is null if there are no values which are more common than other values.
- `most_common_freqs` – The frequency of the most common values is displayed. This represents the number of occurrences of each divided by the number of rows. The value is null if `most_common_vals` is null.
- histogram_bounds – The column displays the histogram of the list of values that divide the values into groups of approximately equal population. This does not include the values in the `most_common_vals` column, if it is populated. If the histogram_bounds field is null, this is likely a result of the `most_common_vals` field representing the entire population.

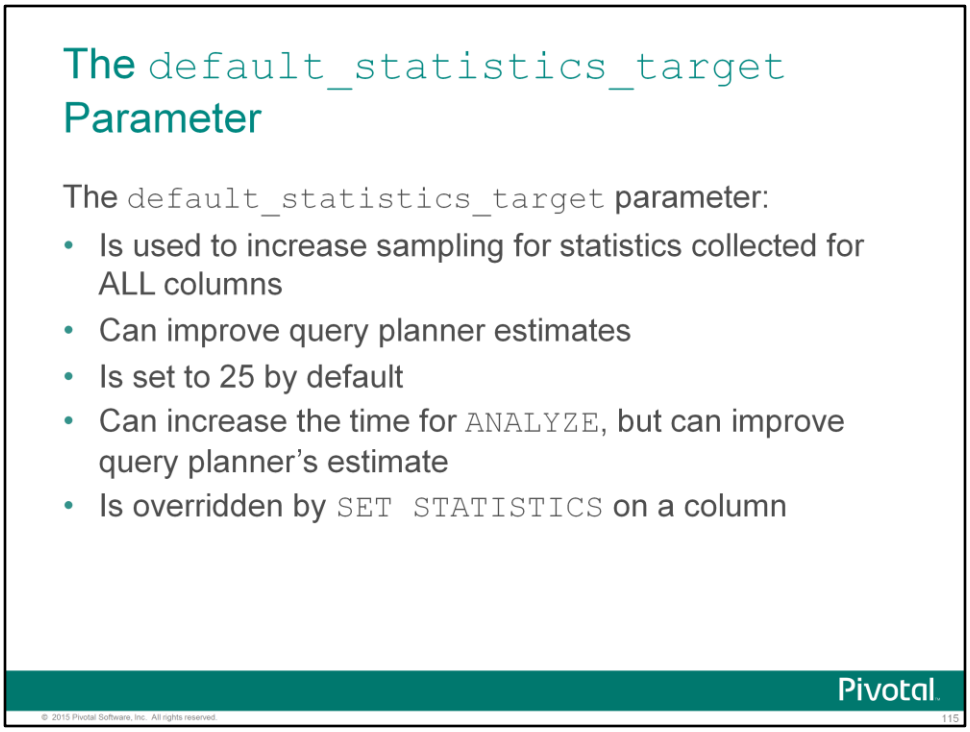


For large tables, `ANALYZE` takes a random sample of the table contents, rather than examining every row to allow large tables to be analyzed in a small amount of time. To increase sampling and statistics collected by `ANALYZE` use the `ALTER TABLE…SET STATISTICS` command.

`SET STATISTICS` can be used to alter the statistics value for a particular table column. This is recommended for columns frequently used in query predicates and joins. The default statistics value is 25. This default value is set with the `default_statistics_target` parameter.

Setting the statistics value to zero disables collection of statistics for that column. It may be useful to set the statistics value to zero for columns that are never used as part of the `WHERE`, `GROUP BY`, or `ORDER BY` clauses of queries, since the query planner has no use for statistics on such columns.

Larger values increase the time needed to do `ANALYZE`, but may improve the quality of the query planner's estimates.

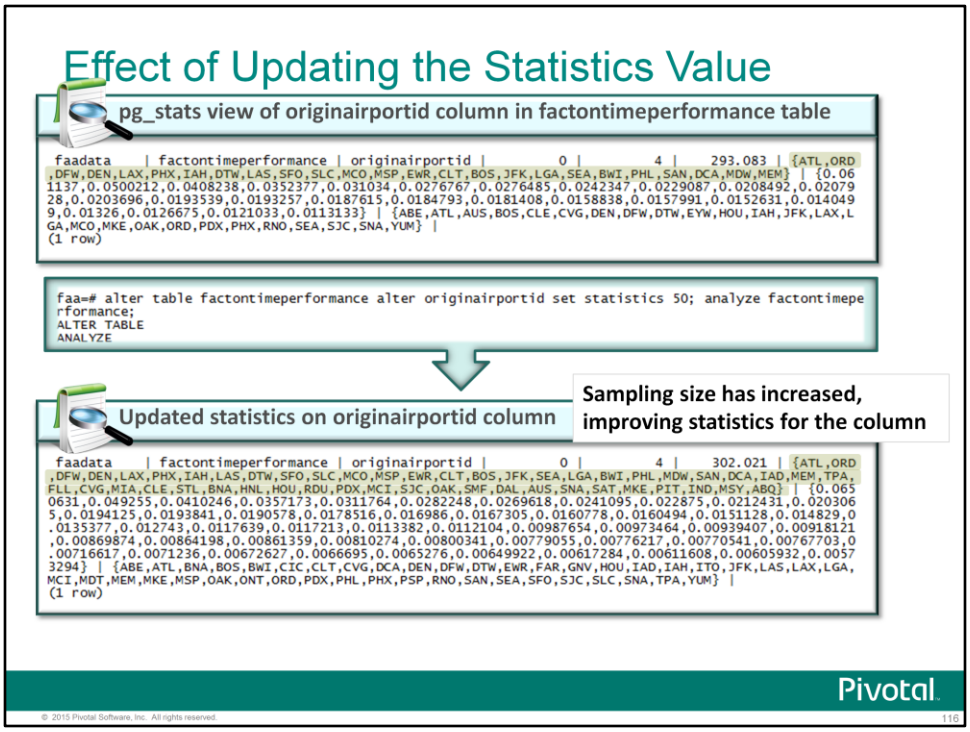


For large tables, rather than examining every row to allow large tables to be analyzed in a small amount of time, `ANALYZE` takes a random sample of the table contents.

To increase sampling for all table columns, adjust the `default_statistics_target` configuration parameter.

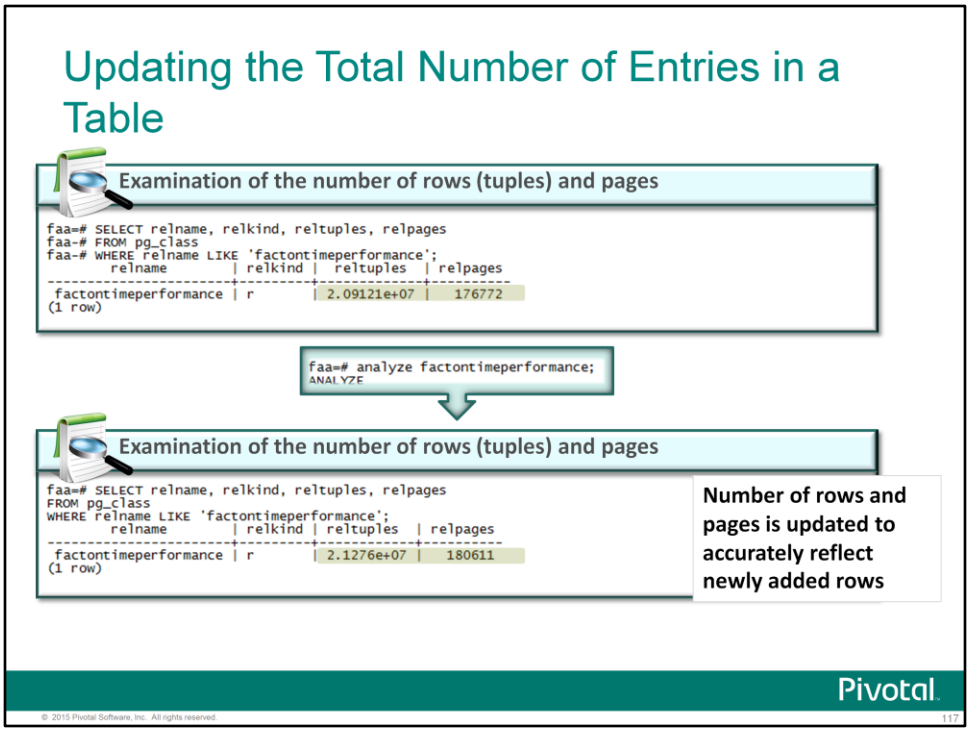
Keep in mind that the `default_statistics_target` server configuration parameter applies to all columns by default. The default target value is 25.

A larger target value will increase the time needed to do `ANALYZE`, but may improve the quality of the query planner's estimates. especially for columns with irregular data patterns, may allow for more accurate query plans.



In this example, the statistics for the `originairportid` column of the `factontimeperformance` table is displayed. The sampling is based on the statistics value of 25, so the database sampled enough rows to obtain 25 distinct values in the `most_common_vals` column.

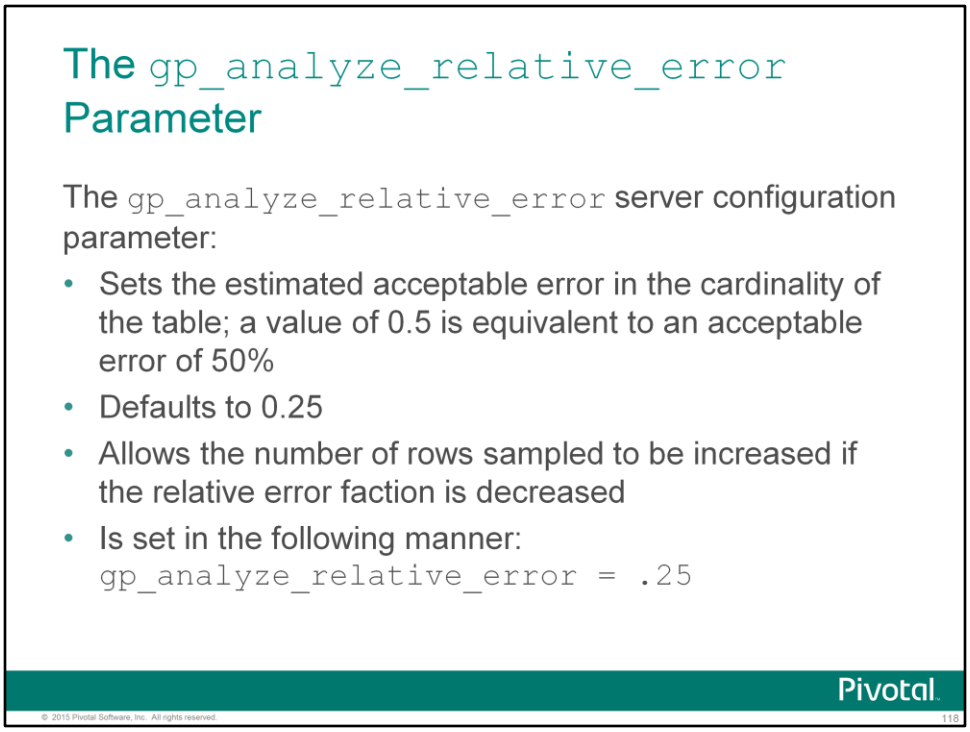
Next, the statistics on the `originairportid` table is increased to 50 and the table analyzed. This requests that the analyzer increase the sampling size so that there are now up to 50 distinct values in the `most_common_vals` column. By increasing the sampling size, the query planner will have a better insight into the makeup of the values within the column. This type of action can be performed on columns that are used more often in `WHERE` clauses. While it increases the amount of time required for analyzing the table, it gives a more accurate description of the makeup of the column, potentially improving the performance for the query.



The number of rows and pages, or disk blocks, used by a table are stored in the `pg_class` table. This information is a component of statistics, used by the planner to build the query plan. The `reltuples` and `relpages` are not constantly updated and relies on the analysis performed by commands such as `VACUUM`, `ANALYZE`, and `CREATE INDEX` to update the values. As these values may be out of date, it could impact the final query plan generated for the table.

In this example, the number of rows is displayed as of the last `ANALYZE`. Separately, several thousand rows were added to the table. If the `ANALYZE` command is not executed, the pg_class table continues to display the now outdated number of rows in the table.

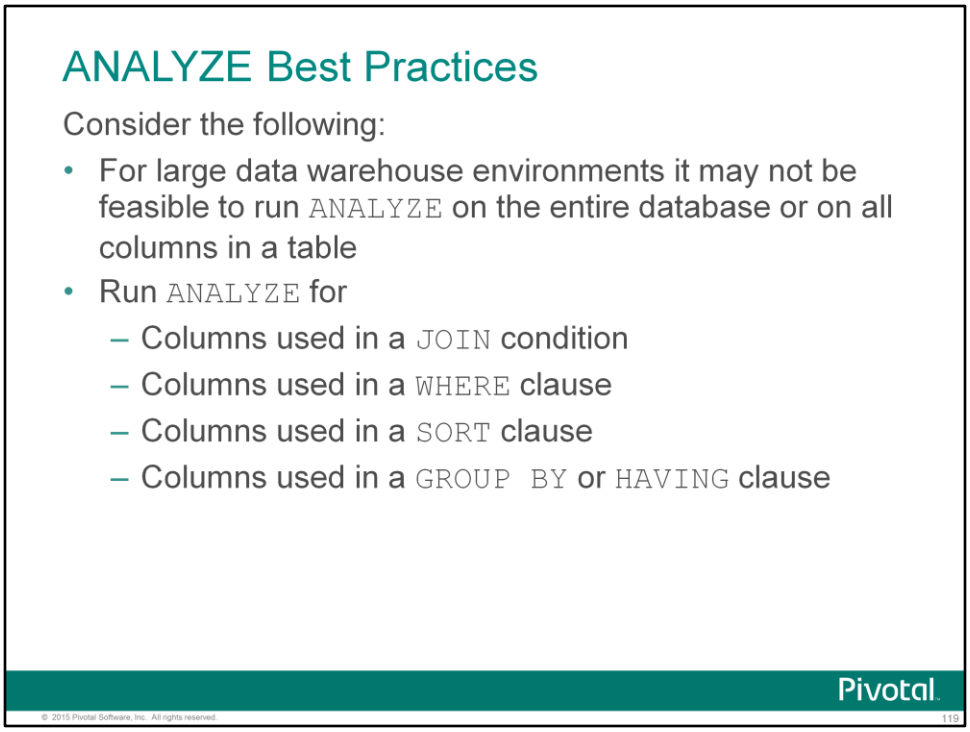
Executing the `ANALYZE` command updates the number of rows as well as the number of disk blocks occupied by those rows.



The `gp_analyze_relative_error` server configuration parameter sets the estimated acceptable error in the cardinality of the table. A value of 0.5 is equivalent to an acceptable error of 50%. The default value is 0.25.

If the statistics collected are not producing good estimates of cardinality for a particular table attribute, decreasing the relative error fraction, or accepting less error, increases the number of rows sampled to determine the number of distinct non-null data values in a column. The number of distinct non-null data values in a column are stored in the `stadistinct` column of the `pg_statistic` table.

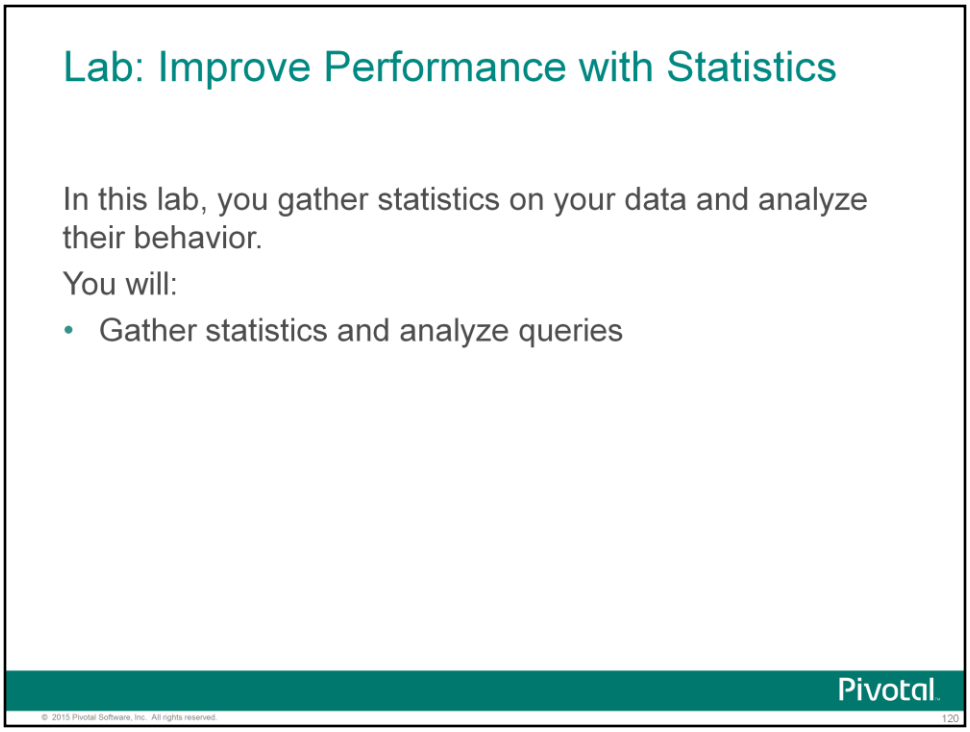
A larger target value will increase the time needed to do `ANALYZE`, but may improve the quality of the query planner's estimates.



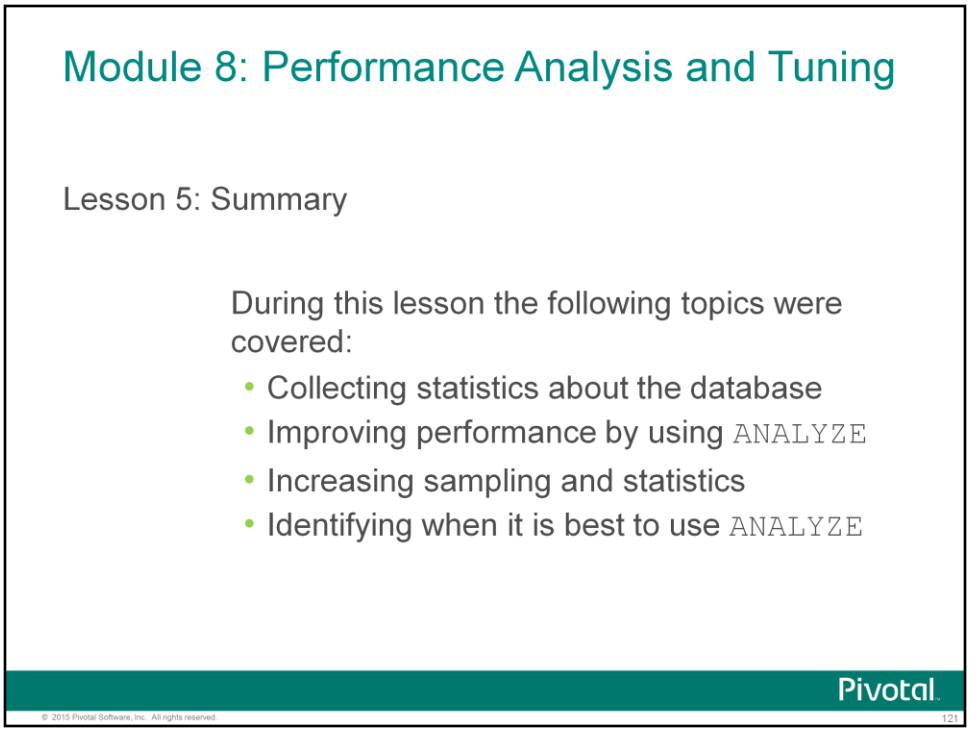
It is important to maintain accurate statistics for optimal query performance. In large data warehouse environments it may not be feasible to run `ANALYZE` on an entire database or table due to time constraints. In this scenario it is important to generate statistics on columns used in a:

- `JOIN` condition
- `WHERE` clause
- `SORT` clause
- `GROUP BY` or `HAVING` clause

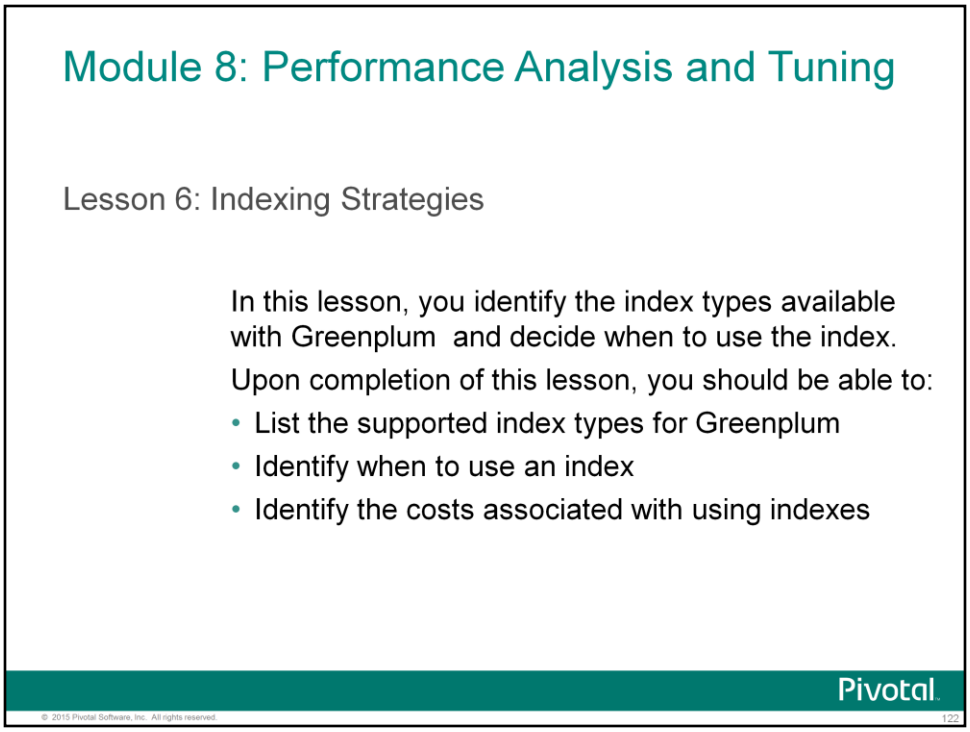
`ANALYZE` requires only a read lock on the table, so may be run in parallel with other database activity though not recommended when performing loads, `INSERT`, `UPDATE`, `DELETE`, and `CREATE INDEX` operations. Run `ANALYZE` after load, `INSERT`, `UPDATE`, `DELETE`, and `CREATE INDEX` operations.



In this lab, you gather statistics on your data and analyze their behavior.



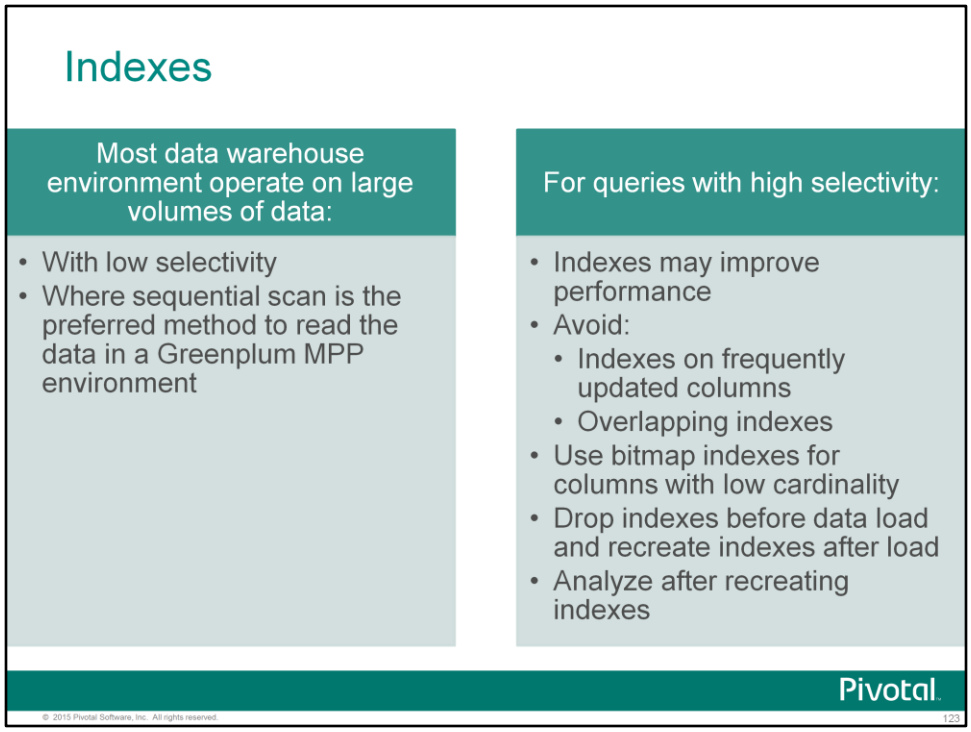
This lesson discussed statistics and the need to provide updated statistics to the optimizer to more accurately predict the costs and time required to complete a query. The EXPLAIN ANALYZE and ANALYZE commands allow you to update metadata about tables, rows, and columns to improve the query plan that is generated by the optimizer. In addition to updating information on relations, increasing sampling size and statistics helps to fine tune the statistics collected on relations, which in turn can improve the query plan the optimizer generates.



If and when to index are key considerations when tuning your queries. You should be aware of when to use indexes and what types should be used, if you intend to implement them in the Greenplum Database.

In this lesson, you will:

- List the supported index types for Greenplum
- Identify when to use an index
- Identify the costs associated with using indexes

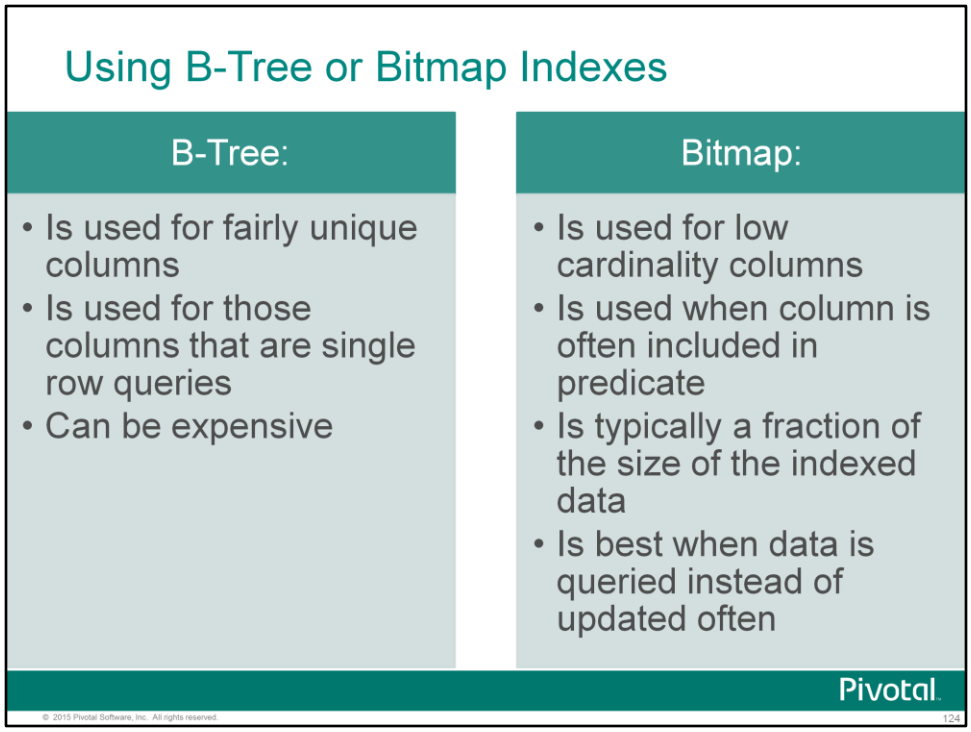


In most data warehouse environment, queries operate on large volumes of data. In the Greenplum MPP environment a sequential is an efficient method to read data as each segment instance contains a portion of the data and all segment instances work in parallel. In an MPP shared nothing environment, segments are not competing for resources like in a traditional SMP environment.

For queries with high selectivity, indexes may improve query performance. First, run the query without indexes to determine if the query performance is acceptable. If it is determined that indexes are needed, consider best practices when creating and managing indexes.

Avoid indexes on columns that are frequently update and avoid overlapping indexes. Use bitmap indexes instead of a B-tree index for columns with very low cardinality. Low cardinality is a column that contains few values, such as `active` and `inactive`.

Drop indexes before loading data into a table. After the load re-create the indexes for the table. This will run an order of magnitude faster than loading data into a table with indexes.



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

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

A bitmap index offers the best choice when less than 1% of the row count is distinct. Low cardinality with a high row count favors bitmap indexes.

Data with low cardinality and low row count should consider using a B-tree index.

In the end, the strategy you use depends on the nature of your data.

Create Index Syntax

The following is the syntax to create an index:

```
CREATE [UNIQUE] INDEX [CONCURRENTLY]
    name ON table
   [USING method]        ( {column | (expression)}
  [opclass] [, ...] )
 [ WITH ( FILLFACTOR = value )
[WHERE predicate]
```

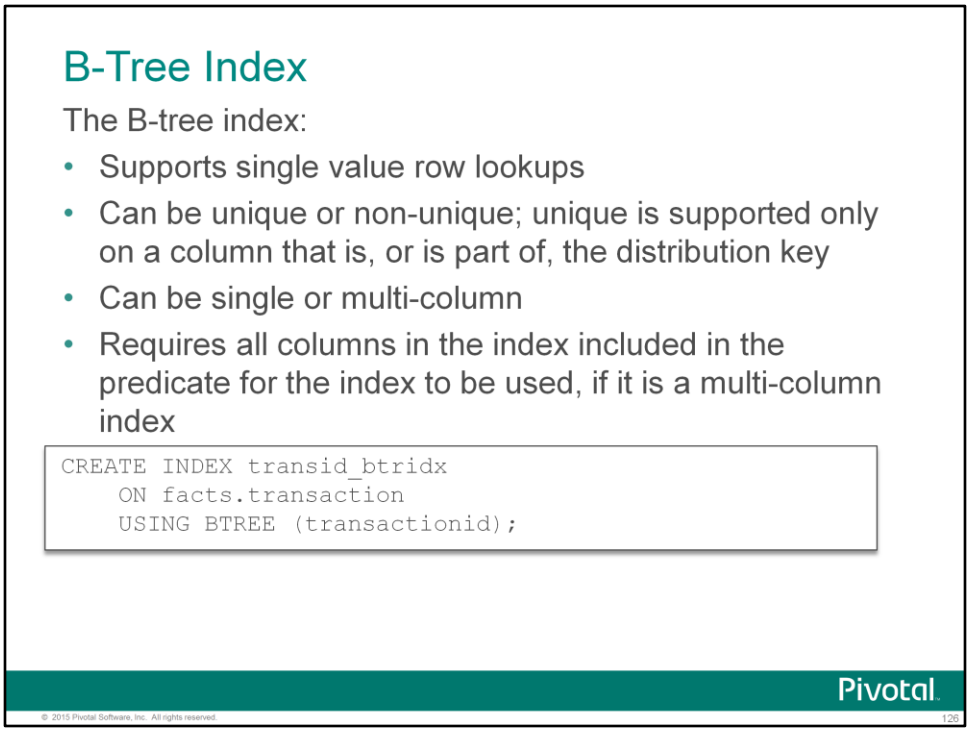
The following is an example of how to create a bitmap index:

```
CREATE INDEX gender_bmp_idx ON employee USING bitmap
(gender);
```

Pivotal

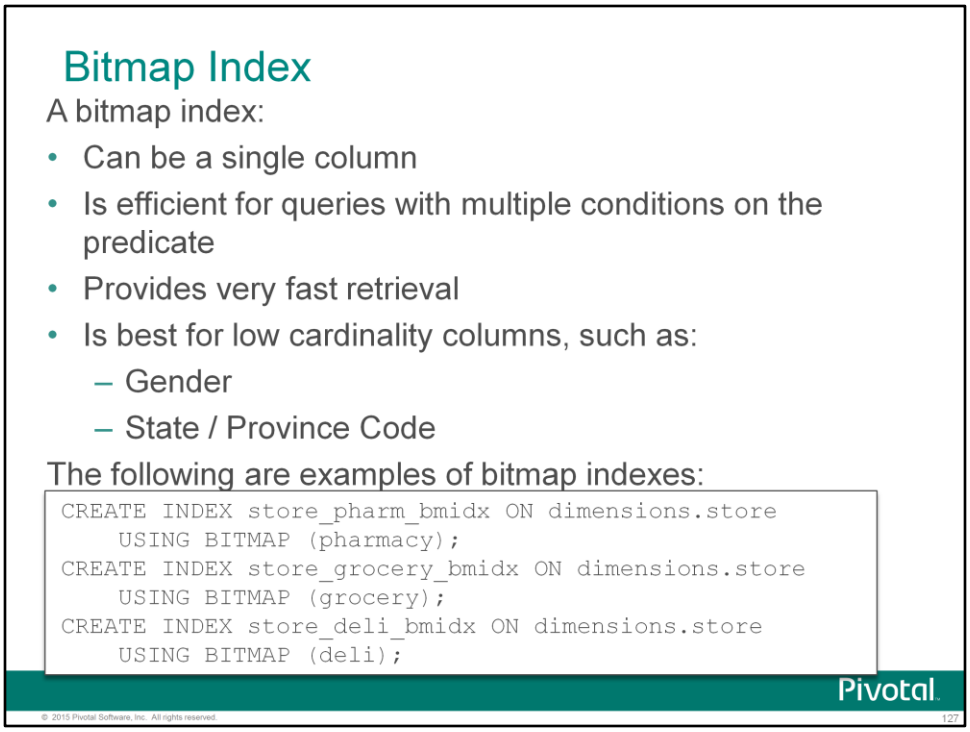
Partial indexes are used as filters. These are considered indexes with a `WHERE` clause.

You can create an index on a function name or other expression. The expression has to be in the predicate so that the expression can use the index.

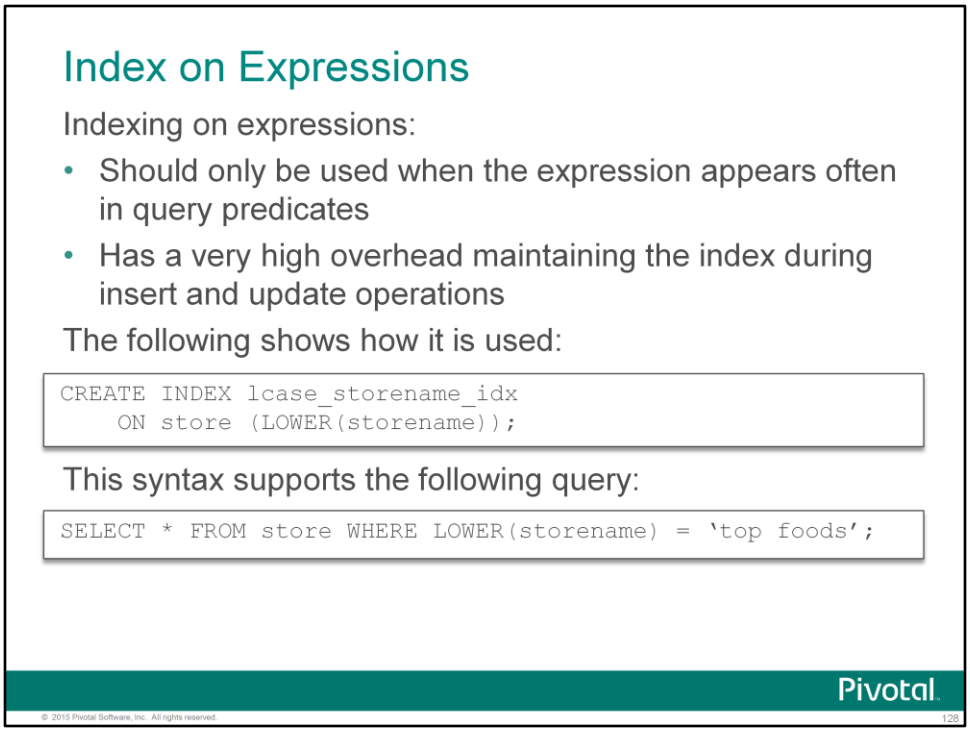


A B-tree index supports single value row lookups. It can be used on unique and non-unique values and with single or multi-column data.

You do not have to specify USING BTREE as it is the default type of index created when a terminology is not specified.

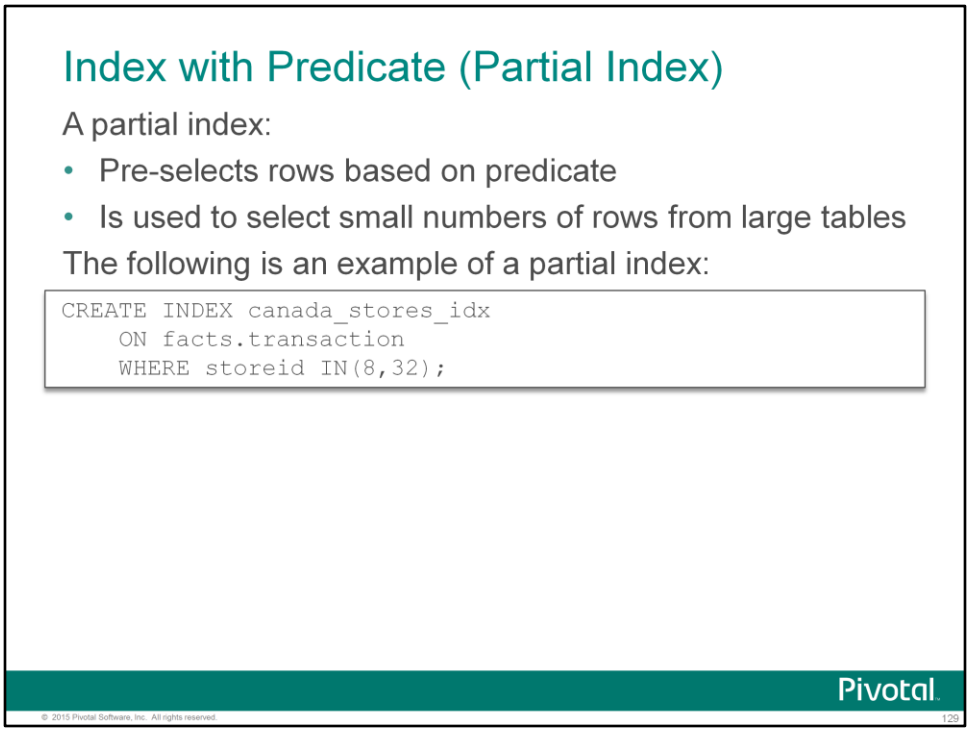


It is not recommended to have multi-column indexes with bitmap indexes. However, a bitmap index is most effective for queries that contain multiple conditions in the WHERE clause. Rows that satisfy some, but not all, conditions are filtered out before the table itself is accessed. This improves response time, often dramatically.

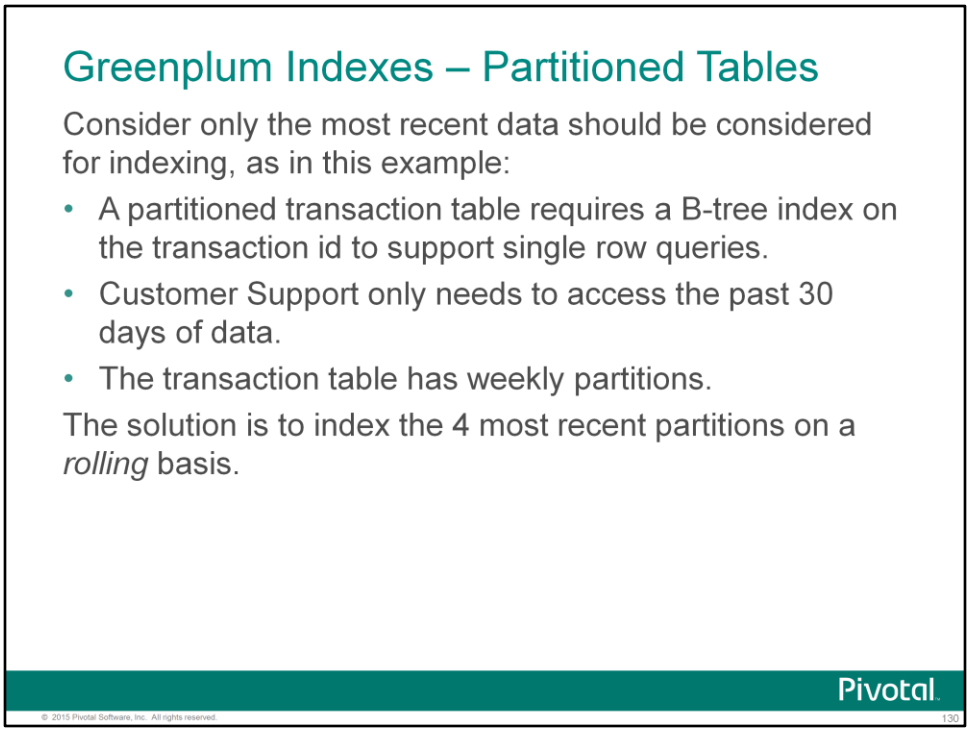


While an index can be used as part of an expression, care should be taken as it has a very high overhead when maintaining the index during an insert or update operation. This method should only be used when the expression appears often in query predicates, the `WHERE` clause.

In this example, the `LOWER (storename)` expression may be used often in queries, so you could create an index on that expression.



A partial index is created when the WHERE clause is presented as part of the CREATE INDEX statement. A partial index is an index that contains entries for only a portion of a table, usually a portion that is more useful for indexing than the rest of the table. For example, if you have a table that contains both billed and unbilled orders where the unbilled orders take up a small fraction of the total table and yet is most often selected, you can improve performance by creating an index on just that portion.

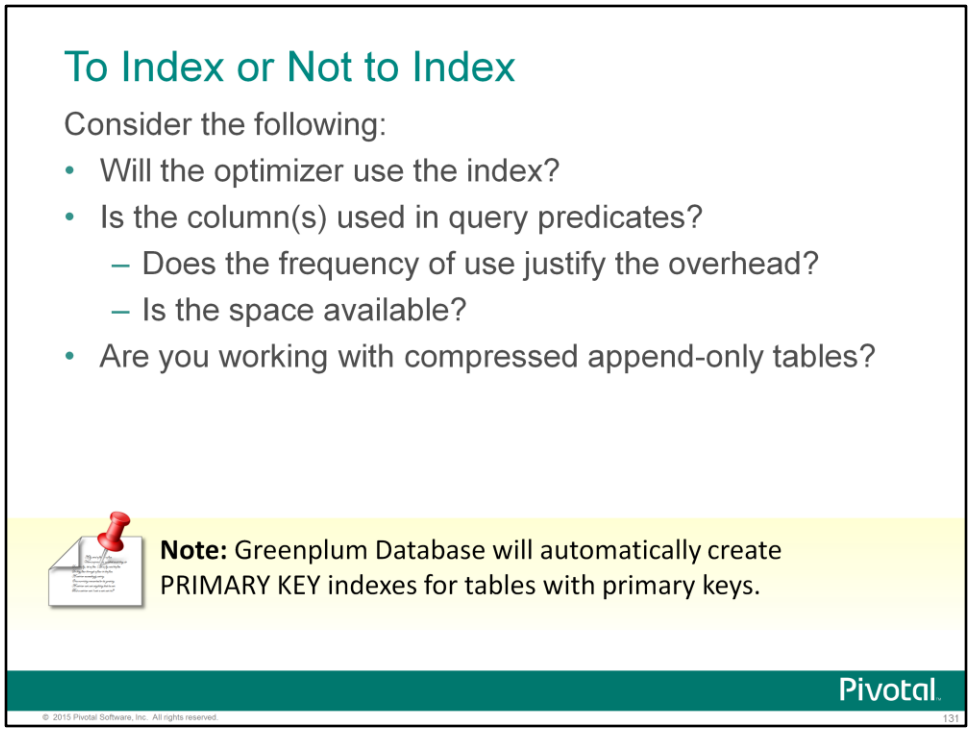


Often, only the most recent data should be considered for indexing based on the performance requirements of the user community.

For Example:

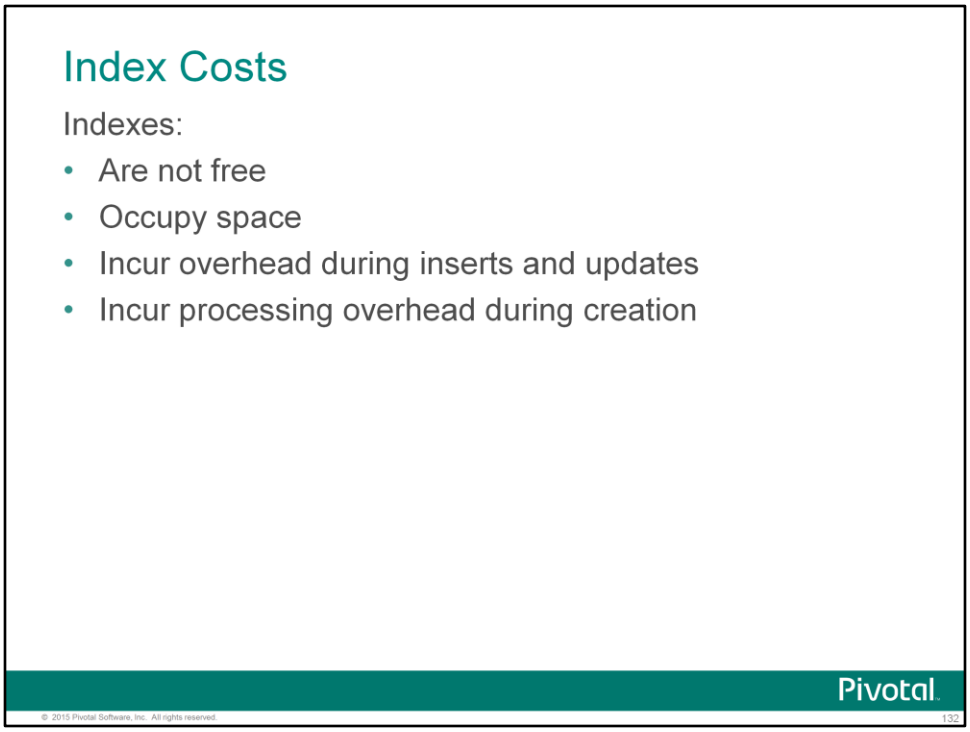
- A partitioned transaction table requires a B-tree index on the transaction id to support single row queries.
- Customer Support only needs to access the past 30 days of data.
- The transaction table has weekly partitions.

The solution for this problem is to Index the 4 most recent partitions on a rolling basis.

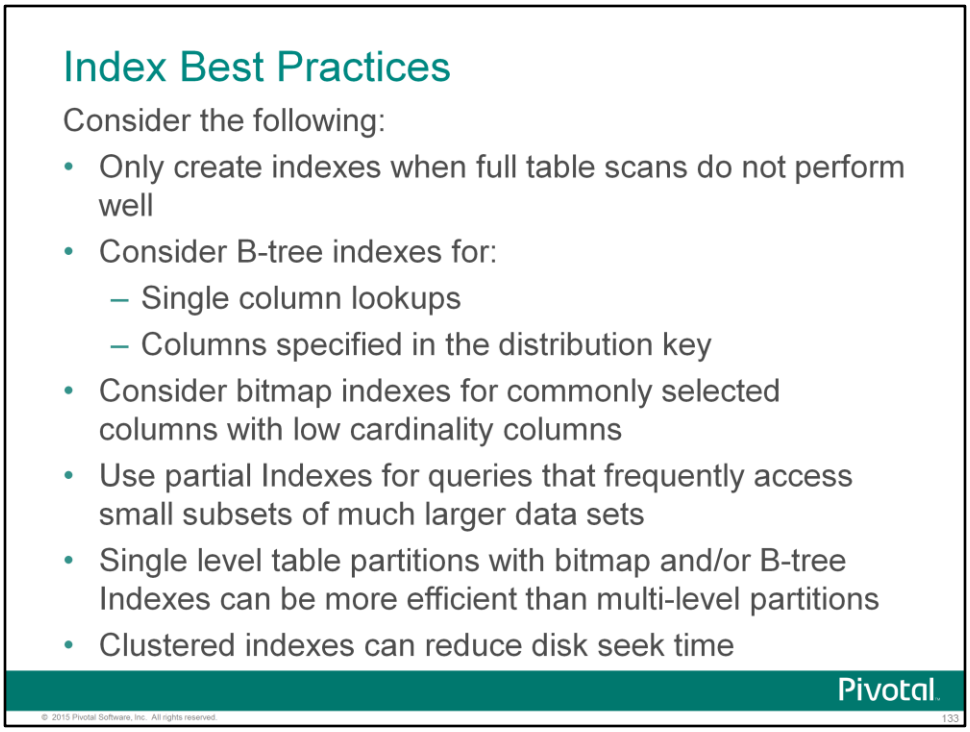


There are several things to consider when deciding whether or not to use indexes:

- Will the optimizer use the index? If the optimizer does not use the index, you may be adversely affecting overall performance by maintaining an index that is not in use. Verify that the optimizer uses the index by issuing queries on production volumes. Additionally, consider that indexes can potentially lead to performance improvements for OLTP type workloads as opposed to BI query workloads, where OLTP transactions return a much smaller data set.
- Is the column(s) used in query predicates? If the column is constantly used in a query predicate, then there may be benefits to creating the index as it may decrease the number of rows selected from a large table.
  However, consider how much it costs to store the index, especially if it is a B-tree index. Sometimes, the index can be several times larger than the data stored in the table. Consider also how often the index is used. If it is used rarely, the costs may outweigh the benefits.
- Are you working with compressed append-only tables? If so, you can see some benefits with implementing queries where an index access method means only necessary rows are uncompressed.

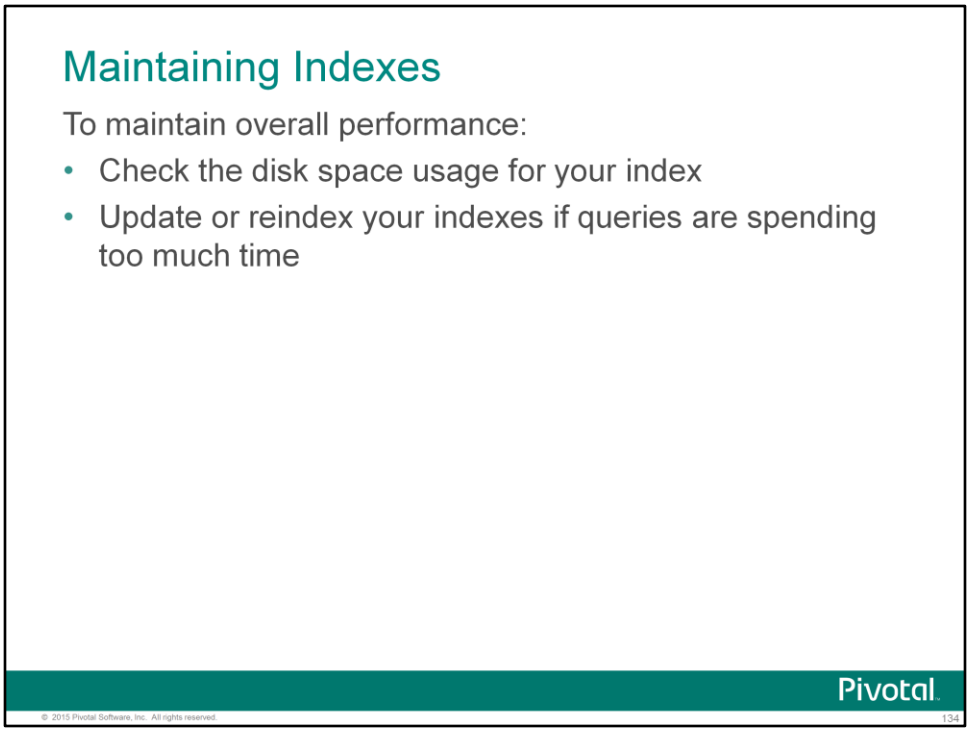


Indexes are not free. They do require space. The worst single performance killer for queries are indexes. If needed, drop them first to reduce overhead. They require maintenance whenever the table is updated. Make sure that the indexes you create are actually being used by your query workload to justify creation and maintenance costs.



Consider the following best practices when working with indexes:

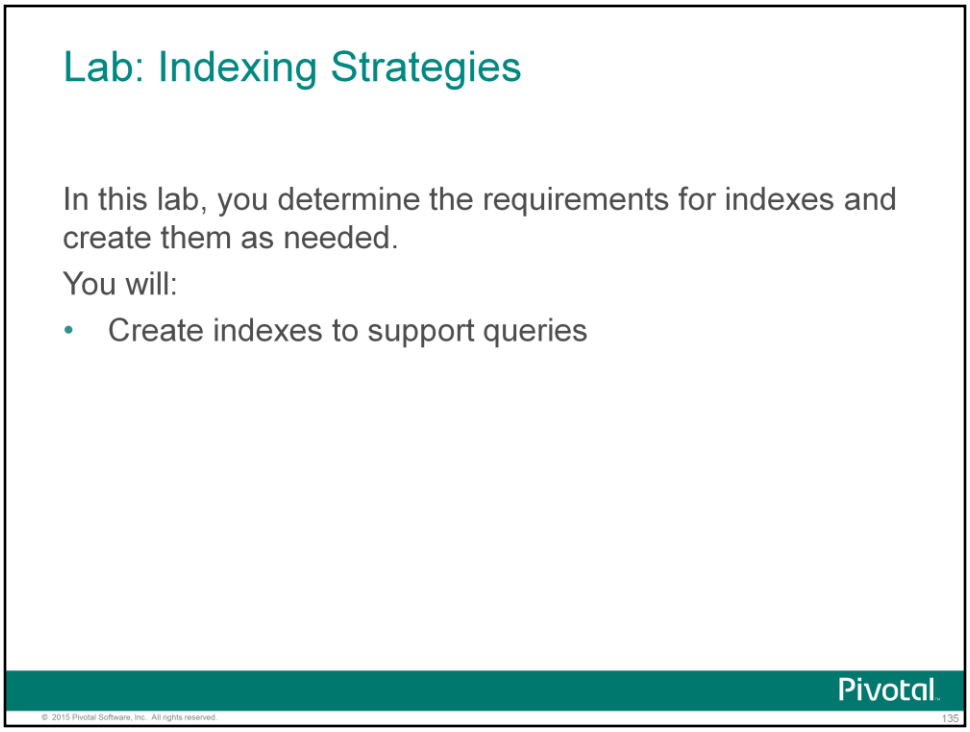
- Verify that you do need an index. As a general rule, do not create indexes if you do not absolutely need them. If you find that table scans are slow and the data can support, try inserting an index. If you do not see any performance gains however, remove the index. Verify that for any index you create, it is used by the optimizer.
- If you find you have a need for a single column lookup, use a B-tree index for a single row lookup.
- Consider using bitmap indexes for commonly selected columns with low cardinality.
- Partial indexes can reduce the number of rows selected. This is particularly useful when working with large tables.
- Using an index for your data may be more efficient than using multi-level partitioning. Consider partitions created on gender or other low-cardinality values. An index can be easier to implement and may provide performance gains.
- Clustered indexes potentially reduce seek times by physically ordering records together according to the index. Randomly distributed records requires that Greenplum seeks across the disk for the records. However, moving them closer together can reduce the seek time on disk making the fetches more sequential. A good example of a clustered index is on a date range. Use the `CLUSTER` command to create a clustered index.



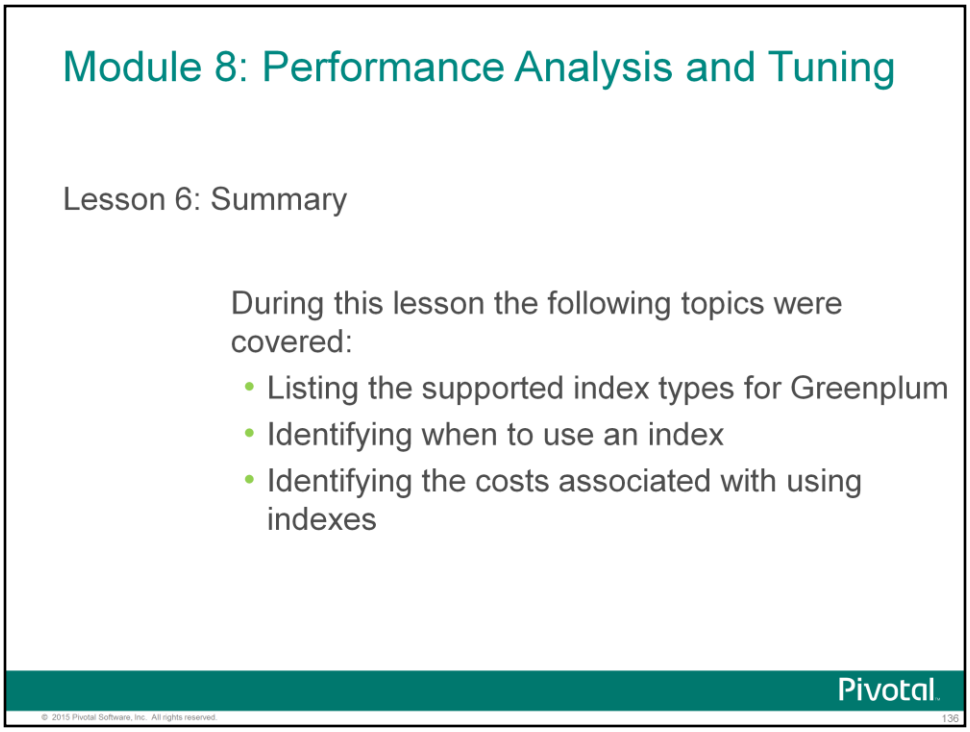
The `gp_toolkit` administrative schema contains a number of views for checking index sizes. To see the total size of all indexes on a table, use the `gp_size_of_all_table_indexes` view. To see the size of a particular index, use the `gp_size_of_index` view. The index sizing views list tables and indexes by object ID and not by name. To check the size of an index by name, you must look up the table name in the `pg_class` by cross-referencing the `relname` column.

For B-tree indexes, a freshly-constructed index is somewhat faster to access than one that has been updated many times. This is because logically adjacent pages are usually also physically adjacent in a newly built index. It might be worthwhile to reindex periodically to improve access speed. Also, if all but a few index keys on a page have been deleted, there will be wasted space on the index page. A reindex will reclaim that wasted space. In Greenplum Database it is often faster to drop an index, using the `DROP INDEX` command and then recreate it with the `CREATE INDEX` command than it is to use the `REINDEX` command.

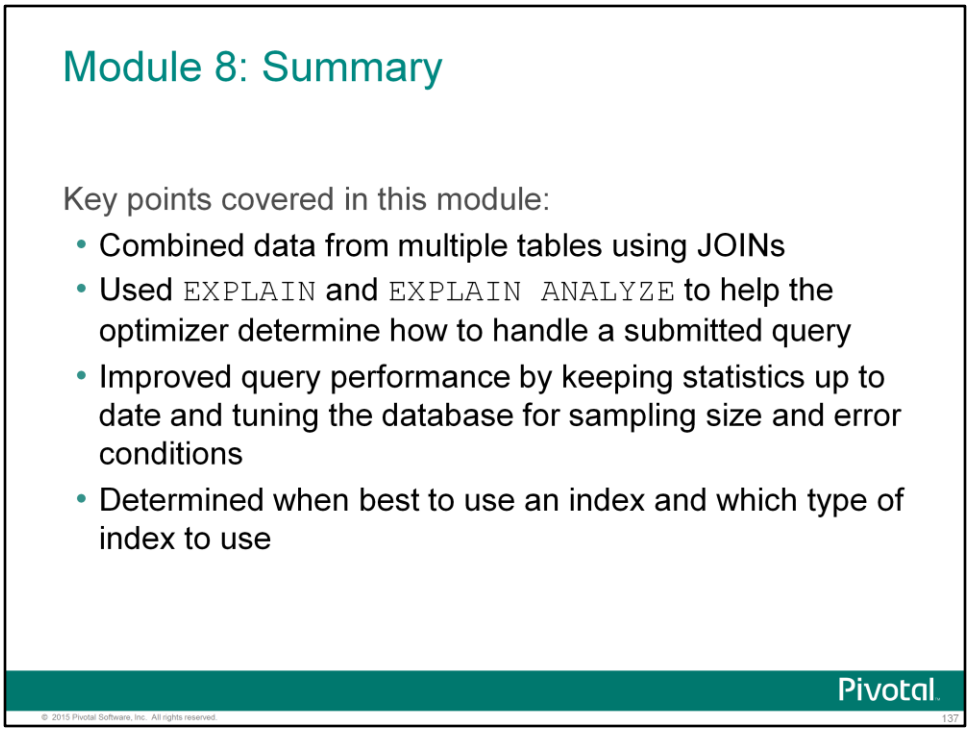
Bitmap indexes are not updated when changes are made to the indexed column(s). If you have updated a table that has a bitmap index, you must drop and recreate the index for it to remain current.



In this lab, you determine the requirements for indexes and create them as needed.



This lesson focused more closely on indexes, the negative and positive impacts of using supported index types in Greenplum, when to use indexes, and how to determine if the use of the indexes is having a positive or negative impact.



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

- Combine data from multiple tables using JOINs.
- 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.

This slide is intentionally left blank.

Pivotal