|  |
| --- |
| EPAM Systems, RD Dep. |
| MTN.BI.07 Oracle Data Access and Optimizer |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| REVISION HISTORY | | | | | |
| Ver. | Description of Change | Author | Date | Approved | |
| Name | Effective Date |
| 1.0 | Initial status | [Kiryl Bucha](mailto:Kiryl_Bucha@epam.com) | 12-JAN-2012 |  |  |
| 2.0 | Updated in accordance with renewed content | [Elias Nema](mailto:Elias_Nema@epam.com) | 20-JAN-2014 |  |  |

Contents

[1. Query Optimization 3](#_Toc383288457)

[1.1. Old school: Rule-Based Optimization 3](#_Toc383288458)

[1.2. Nowadays: Cost-Based Optimization 4](#_Toc383288459)

[1.2.1. How statistics are used 4](#_Toc383288460)

[1.3. Optimizer Components 4](#_Toc383288461)

[1.3.1. Query Transformer 5](#_Toc383288462)

[1.3.2. Estimator 5](#_Toc383288463)

[1.3.3. Plan Generator 5](#_Toc383288464)

[2. High-water Mark 6](#_Toc383288465)

[2.1. Influencing the cost-based optimizer 7](#_Toc383288466)

[3. Overview of SQL Processing 8](#_Toc383288467)

[3.1. Stages of SQL Processing 8](#_Toc383288468)

[3.1.1. SQL Parsing 8](#_Toc383288469)

[3.1.2. SQL Optimization 9](#_Toc383288470)

[3.1.3. SQL Row Source Generation 9](#_Toc383288471)

[3.1.4. SQL Execution 9](#_Toc383288472)

[3.1.5. Differences Between DML and DDL Processing 9](#_Toc383288473)

[4. Access Paths 9](#_Toc383288474)

[4.1. Full Scan Access Methods 10](#_Toc383288475)

[4.2. Index Scan Access Methods 11](#_Toc383288476)

[4.2.1. Index Unique Scan 12](#_Toc383288477)

[4.2.2. Index range scan 12](#_Toc383288478)

[4.2.3. Index Full Scan 12](#_Toc383288479)

[4.2.4. Index Skip Scan 13](#_Toc383288480)

[4.2.5. Index Fast Full Scan 13](#_Toc383288481)

[5. Source Books and Articles 14](#_Toc383288482)

# Query Optimization

All of the data structures discussed so far are database entities. Users request data from an Oracle server through database queries. Oracle’s query optimizer must then determine the best way to access the data requested by each query.

To understand how Oracle Database processes SQL statements, it is necessary to understand the part of the database called the optimizer (also known as the query optimizer or cost-based optimizer). All SQL statements use the optimizer to determine the most efficient means of accessing the specified data.

One of the great virtues of a relational database is its ability to access data without predefining the access paths to the data. When a SQL query is submitted to an Oracle Database, Oracle must decide how to access the data. The process of making this decision is called query optimization, because Oracle looks for the optimal way to retrieve the data. This retrieval is known as the execution path. The trick behind query optimization is to choose the most efficient way to get the data, since there may be many different options available.

For instance, even with a query that involves only a single table, Oracle can take either of these approaches:

* Use an index to find the ROWIDs of the requested rows and then retrieve those rows from the table.
* Scan the table to find and retrieve the rows; this is referred to as a full table scan.

Although it’s usually much faster to retrieve data using an index, the process of getting the values from the index involves an additional I/O step in processing the query. This additional step could mean that there were more total I/Os involved to satisfy the query —if, for instance, all the rows in a table were being selected. Query optimization may be as simple as determining whether the query involves selection conditions that can be imposed on values in the index. Using the index values to select the desired rows involves less I/O and is therefore more efficient than retrieving all the data from the table and then imposing the selection conditions. But when you start to consider something as simple as what percent of the rows in a table will be eliminated by using an index, you can see that the complexity in selecting the right execution path can grow very complex very rapidly in a production scenario.

Another factor in determining the optimal query execution plan is whether there is an ORDER BY condition in the query that can be automatically implemented by the presorted index. Alternatively, if the table is small enough, the optimizer may decide to simply read all the blocks of the table and bypass the index since it estimates the cost of the index I/O plus the table I/O to be higher than just the table I/O.

The query optimizer has to make some key decisions even with a query on a single table. When a more involved query is submitted, such as one involving many tables that must be joined together efficiently, or one that has complex selection criteria and multiple levels of sorting, the query optimizer has a much more complex task.

Prior to Oracle Database 10g, you could choose between two different Oracle query optimizers, a rule-based optimizer and a cost-based optimizer; these are described in the following sections. Since Oracle Database 10g, the rule-based optimizer is desupported. The references to syntax and operations for the rule-based optimizer in the following sections are provided for reference and are applicable only if you are running a very old release of Oracle.

## Old school: Rule-Based Optimization

Oracle has always had a query optimizer, but until Oracle7 the optimizer was only rule based. The rule-based optimizer, as the name implies, uses a set of predefined rules as the main determinant of query optimization decisions. Since the rule-based optimizer has been desupported as of Oracle Database 10g, your interest in this topic is likely be limited to supporting old Oracle Databases where this choice may have been made.

Rule-based optimization sometimes provided better performance than the early versions of Oracle’s cost-based optimizer for specific situations. The rule-based optimizer had several weaknesses, including offering only a simplistic set of rules—and, at the time of this writing, has not been enhanced for several releases. The Oracle rule-based optimizer had about 20 rules and assigned a weight to each one of them. In a complex database, a query can easily involve several tables, each with several indexes and complex selection conditions and ordering. This complexity means that there were a lot of options, and the simple set of rules used by the rule-based optimizer might not differentiate the choices well enough to make the best choice.

The rule-based optimizer assigned an optimization score to each potential execution path and then took the path with the best optimization score. Another weakness in the rule-based optimizer was resolution of optimization choices made in the event of a “tie” score. When two paths presented the same optimization score, the rule-based optimizer looked to the syntax of the SQL statement to resolve the tie. The winning execution path was based on the order in which the tables occurred in the SQL statement. You can understand the potential impact of this type of tiebreaker by looking at a simple situation in which a small table with 10 rows, SMALLTAB, is joined to a large table with 10,000 rows, LARGETAB. If the optimizer chose to read SMALLTAB first, the Oracle Database would read the 10 rows and then read LARGETAB to find the matching rows for each of the 10 rows. If the optimizer chose to read LARGETAB first, the database would read 10,000 rows from LARGETAB and then read SMALLTAB 10,000 times to find the matching rows. Of course, the rows in SMALLTAB would probably be cached, reducing the impact of each probe, but you could still see a dramatic difference in performance (20 vs. 20000 logical I/Os).

Differences like this could occur with the rule-based optimizer as a result of the ordering of the table names in the query. In the previous situation the rule-based optimizer returned the same results for the query, but it used widely varying amounts of resources to retrieve those results.

## Nowadays: Cost-Based Optimization

To improve the optimization of SQL statements, Oracle introduced the cost-based optimizer in Oracle7. As the name implies, the cost-based optimizer does more than simply look at a set of optimization rules; instead, it selects the execution path that requires the least number of logical I/O operations. This approach avoids the problems discussed in the previous section. The cost-based optimizer would know which table was bigger and would select the right table to begin the query, regardless of the syntax of the SQL statement.

Oracle8 and later versions, by default, use the cost-based optimizer to identify the optimal execution plan. And, since Oracle Database 10g, the cost-based optimizer is the only supported optimizer. To properly evaluate the cost of any particular execution plan, the cost-based optimizer uses statistics about the composition of the relevant data structures. These statistics are automatically gathered by default since the Oracle Database 10g release. Among the statistics gathered in the AWR are database segment access and usage statistics, time model statistics, system and session statistics, statistics about which SQL statements that produce the greatest loads, and Active Session History (ASH) statistics.

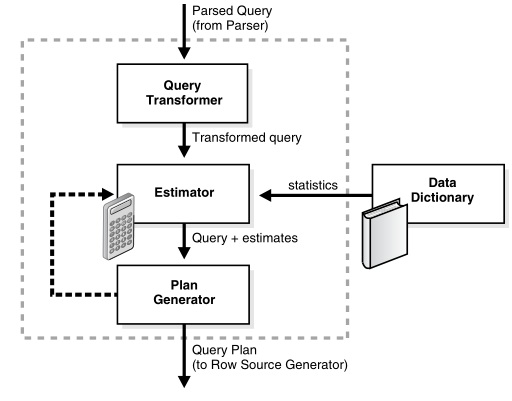
### How statistics are used

The cost-based optimizer finds the optimal execution plan by assigning an optimization score for each of the potential execution plans using its own internal rules and logic along with statistics that reflect the state of the data structures in the database. These statistics relate to the tables, columns, and indexes involved in the execution plan (such kind of statistics as: number of rows/blocks in table, number of distinct values per column and density factor, depth of index B\*-tree structure, number of leaf blocks and distinct values of index, clustering factor). Oracle Database 10g and more current database releases also collect overall system statistics, including I/O and CPU performance and utilization. These statistics are stored in the data dictionary.

The use of statistics makes it possible for the cost-based optimizer to make a much more well-informed choice of the optimal execution plan. For instance, the optimizer could be trying to decide between two indexes to use in an execution plan that involves a selection based on a value in either index. The rule-based optimizer might very well rate both indexes equally and resort to the order in which they appear in the WHERE clause to choose an execution plan. The cost-based optimizer, however, knows that one index contains 1,000 entries while the other contains 10,000 entries. It even knows that the index that contains 1,000 values contains only 20 unique values, while the index that contains 10,000 values has 5,000 unique values. The selectivity offered by the larger index is much greater, so that index will be assigned a better optimization score and used for the query.

## Optimizer Components

The optimizer contains three main components, which are shown in Figure 1.



**Figure 1 Optimizer Components**

The input to the optimizer is a parsed query. The optimizer performs the following operations:

1. The optimizer receives the parsed query and generates a set of potential plans for the SQL statement based on available access paths and hints.
2. The optimizer estimates the cost of each plan based on statistics in the data dictionary. The cost is an estimated value proportional to the expected resource use needed to execute the statement with a particular plan.
3. The optimizer compares the costs of plans and chooses the lowest-cost plan, known as the query plan, to pass to the row source generator.

### Query Transformer

The query transformer determines whether it is helpful to change the form of the query so that the optimizer can generate a better execution plan. The input to the query transformer is a parsed query, which the optimizer represents as a set of query blocks.

### Estimator

The estimator determines the overall cost of a given execution plan. The estimator generates three different types of measures to achieve this goal:

* Selectivity. This measure represents a fraction of rows from a row set. The selectivity is tied to a query predicate, such as last\_name=‘Smith’, or a combination of predicates.
* Cardinality. This measure represents the number of rows in a row set.
* Cost. This measure represents units of work or resource used. The query optimizer uses disk I/O, CPU usage, and memory usage as units of work.

If statistics are available, then the estimator uses them to compute the measures. The statistics improve the degree of accuracy of the measures.

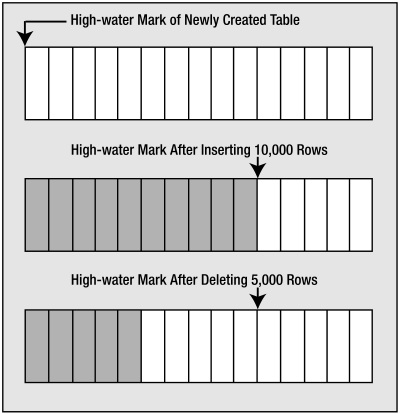
### Plan Generator

The plan generator tries out different plans for a submitted query, and then picks the plan with the lowest cost. The optimizer generates subplans for each of the nested subqueries and unmerged views. The optimizer represents each subplan as a separate query block. The plan generator explores various plans for a query block by trying out different access paths, join methods, and join orders.

The adaptive query optimization capability changes plans based on statistics collected during statement execution. All adaptive mechanisms can execute a final plan for a statement that differs from the default plan. Adaptive optimization uses either dynamic plans, which choose among subplans during statement execution, or reoptimization, which changes a plan on executions after the current execution.

# High-water Mark

This is a term used with table segments stored in the database. If you envision a table, for example, as a flat structure or as a series of blocks laid one after the other in a line from left to right, the high-water mark (HWM) would be the rightmost block that ever contained data, as illustrated in Figure 2.

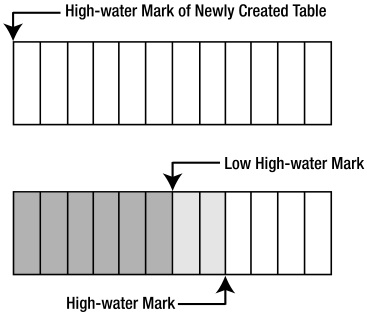


**Figure 2 Oracle HWM**

Figure 2 shows that the HWM starts at the first block of a newly created table. As data is placed into the table over time and more blocks get used, the HWM rises. If we delete some (or even all) of the rows in the table, we might have many blocks that no longer contain data, but they are still under the HWM, and they will remain under the HWM until the object is rebuilt, truncated, or shrunk (shrinking of a segment is a new Oracle 10g feature that is supported only if the segment is in an ASSM tablespace).

The HWM is relevant since Oracle will scan all blocks under the HWM, even when they contain no data, during a full scan. This will impact the performance of a full scan—especially if most of the blocks under the HWM are empty. To see this, just create a table with 1,000,000 rows (or create any table with a large number of rows), and then execute a SELECT COUNT(\*) from this table. Now, DELETE every row in it and you will find that the SELECT COUNT(\*) takes just as long (or longer, if you need to clean out the block!). This is because Oracle is busy reading all of the blocks below the HWM to see if they contain data. You should compare this to what happens if you used TRUNCATE on the table instead of deleting each individual row. TRUNCATE will reset the HWM of a table back to zero and will truncate the associated indexes on the table as well. If you plan on deleting every row in a table, TRUNCATE—if it can be used—would be the method of choice for this reason.

In an MSSM tablespace, segments have a definite HWM. In an ASSM tablespace, however, there is an HWM and a low HWM. In MSSM, when the HWM is advanced (e.g., as rows are inserted), all of the blocks are formatted and valid, and Oracle can read them safely. With ASSM, however, when the HWM is advanced Oracle doesn’t format all of the blocks immediately—they are only formatted and made safe to read upon their first actual use. The first actual use will be when the database decides to insert a record into a given block. Under ASSM, the data is inserted in any of the blocks between the low high water mark and the high water mark, so many of the blocks in this area might not be formatted. So, when full scanning a segment, we have to know if the blocks to be read are safe or unformatted (meaning they contain nothing of interest and we do not process them). To make it so that not every block in the table needs go through this safe/not safe check, Oracle maintains a low HWM and a HWM. Oracle will full scan the table up to the HWM—and for all of the blocks below the low HWM, it will just read and process them. For blocks between the low HWM and the HWM (see Figure 3), it must be more careful and refer to the ASSM bitmap information used to manage these blocks in order to see which of them it should read and which it should just ignore.



**Figure 3 Low High Water Mark**

## Influencing the cost-based optimizer

There are two ways you can influence the way the cost-based optimizer selects an execution plan. The first way is by setting the OPTIMIZER\_MODE initialization parameter. ALL\_ROWS is the default setting for OPTIMIZER\_MODE, enabling optimization with the goal of best throughput. FIRST\_ROWS optimizes plans for returning the first set of rows from a SQL statement. You can specify the number of rows using this parameter. The optimizer mode tilts the evaluation of optimization scores slightly and, in some cases, may result in a different execution plan.

Oracle also gives you a way to influence the decisions of the optimizer with a technique called hints. A hint is nothing more than a comment with a specific format inside a SQL statement. Hints can be categorized as follows:

* Optimizer SQL hints for changing the query optimizer goal
* Full table scan hints
* Index unique scan hints
* Index range scan descending hints
* Fast full index scan hints
* Join hints, including index joins, nested loop joins, hash joins, sort merge joins,
* Cartesian joins, and join order
* Other optimizer hints, including access paths, query transformations, and parallel execution

Hints come with their own set of problems. A hint looks just like a comment, as shown in this extremely simple SQL statement. Here, the hint forces the optimizer to use the EMP\_IDX index for the EMP table:

SELECT /\*+ INDEX(EMP\_IDX) \*/ LASTNAME, FIRSTNAME, PHONE FROM EMP

If a hint isn’t in the right place in the SQL statement, if the hint keyword is misspelled, or if you change the name of a data structure so that the hint no longer refers to an existing structure, the hint will simply be ignored, just as a comment would be. Because hints are embedded into SQL statements, repairing them can be quite frustrating and time-consuming if they aren’t working properly. In addition, if you add a hint to a SQL statement to address a problem caused by a bug in the cost-based optimizer and the cost-based optimizer is subsequently fixed, the SQL statement will still not use the corrected (and potentially improved) optimizer.

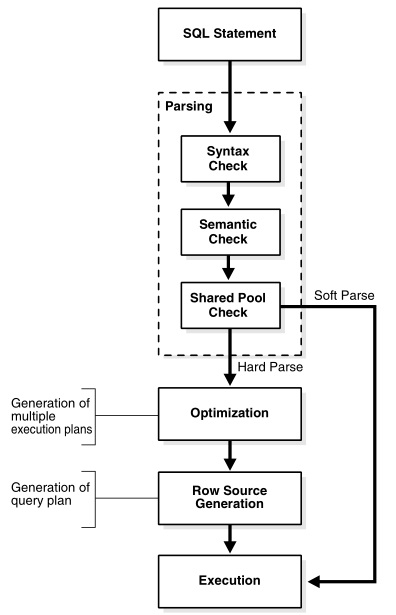
However, hints do have a place—for example, when a developer has a user-defined datatype that suggests a particular type of access. The optimizer cannot anticipate the effect of user-defined datatypes, but a hint can properly enable the appropriate retrieval path.

# Overview of SQL Processing

This section explains how Oracle Database processes SQL statements. Specifically, the section explains the way in which the database processes DDL statements to create objects, DML to modify data, and queries to retrieve data.

## Stages of SQL Processing

Figure 4 depicts the general stages of SQL processing: parsing, optimization, row source generation, and execution. Depending on the statement, the database may omit some of these steps.



**Figure 4 Stages of SQL Processing**

### SQL Parsing

As shown in Figure 4, the first stage of SQL processing is SQL parsing. This stage involves separating the pieces of a SQL statement into a data structure that can be processed by other routines.

When an application issues a SQL statement, the application makes a parse call to the database to prepare the statement for execution. The parse call opens or creates a cursor, which is a handle for the session-specific private SQL area that holds a parsed SQL statement and other processing information. The cursor and private SQL area are in the PGA.

During the parse call, the database performs the following checks:

* Syntax check
* Semantic check
* Shared pool check

The preceding checks identify the errors that can be found before statement execution. Some errors cannot be caught by parsing. For example, the database can encounter deadlocks or errors in data conversion only during statement execution.

### SQL Optimization

Query optimization is the process of choosing the most efficient means of executing a SQL statement. The database optimizes queries based on statistics collected about the actual data being accessed. The optimizer uses the number of rows, the size of the data set, and other factors to generate possible execution plans, assigning a numeric cost to each plan. The database uses the plan with the lowest cost.

The database must perform a hard parse at least once for every unique DML statement and performs optimization during this parse. DDL is never optimized unless it includes a DML component such as a subquery that requires optimization.

### SQL Row Source Generation

The row source generator is software that receives the optimal execution plan from the optimizer and produces an iterative plan, called the query plan that is usable by the rest of the database. The query plan takes the form of a combination of steps. Each step returns a row set. The rows in this set are either used by the next step or, in the last step, are returned to the application issuing the SQL statement. A row source is a row set returned by a step in the execution plan along with a control structure that can iteratively process the rows. The row source can be a table, view, or result of a join or grouping operation.

### SQL Execution

During execution, the SQL engine executes each row source in the tree produced by the row source generator. This step is the only mandatory step in DML processing. During execution, the database reads the data from disk into memory if the data is not in memory. The database also takes out any locks and latches necessary to ensure data integrity and logs any changes made during the SQL execution. The final stage of processing a SQL statement is closing the cursor.

### Differences between DML and DDL Processing

Most DML statements have a query component. In a query, execution of a cursor places the results of the query into the result set.

The database can fetch result set rows either one row at a time or in groups. In the fetch stage, the database selects rows and, if requested by the query, sorts the rows. Each successive fetch retrieves another row of the result until the last row has been fetched.

Oracle Database processes DDL differently from DML. For example, when you create a table, the database does not optimize the CREATE TABLE statement. Instead, Oracle Database parses the DDL statement and carries out the command.

# Access Paths

An access path is the technique that a query uses to retrieve rows. For example, a query that uses an index has a different access path from a query that does not. In general, index access paths are best for statements that retrieve a small subset of table rows. Full scans are more efficient for accessing a large portion of a table.

The database can use several different access paths to retrieve data from a table. The following is a representative list:

1. **Full table scans.** This type of scan reads all rows from a table and filters out those that do not meet the selection criteria. The database sequentially scans all data blocks in the segment, including those under the high water mark (HWM) that separates used from unused space.
2. **Rowid scans.** The rowid of a row specifies the data file and data block containing the row and the location of the row in that block. The database first obtains the rowids of the selected rows, either from the statement WHERE clause or through an index scan, and then locates each selected row based on its rowid.
3. **Index scans.** This scan searches an index for the indexed column values accessed by the SQL statement. If the statement accesses only columns of the index, then Oracle Database reads the indexed column values directly from the index.
4. **Cluster scans.** A cluster scan retrieves data from a table stored in an indexed table cluster, where all rows with the same cluster key value are stored in the same data block. The database first obtains the rowid of a selected row by scanning the cluster index. Oracle Database locates the rows based on this rowid.
5. **Hash scans.** A hash scan locates rows in a hash cluster, where all rows with the same hash value are stored in the same data block. The database first obtains the hash value by applying a hash function to a cluster key value specified by the statement. Oracle Database then scans the data blocks containing rows with this hash value.

The optimizer chooses an access path based on the available access paths for the statement and the estimated cost of using each access path or combination of paths.

## Full Scan Access Methods

When full scanning an object, all the blocks associated with that object must be retrieved and processed to determine if rows in a block match your query’s needs. Remember that Oracle must read an entire block into memory in order to get to the row data stored in that block. So, when a full scan occurs, there are actually two things the optimizer (and you) needs to consider: how many blocks must be read and how much data in each block will be thrown away. The idea I want you to grab on to at this point is that the decision as to whether a full scan is the right choice isn’t just based on how many rows your query will return. There have been many “rules of thumb” published that state things like “if your query will retrieve more than x% of rows from the table, then a full scan should be chosen.” There’s more to the decision than just that ROT (Rule Of Thumb = ROT) and I don’t want you to get stuck on a rule that limits the consideration that should be given to the choice.

There are 4 main factors regarding choosing full table scans over other access methods:

* **Percentage of retrieved rows.** It’s not just about rows, it's also about how data is distributed between the data blocks. The combination of all of these pieces of information may lead to a conclusion that it makes sense to do a full scan even when the percentage of rows is quite small (e.g. you need to choose 100 rows or 1% of table that consists of 10000 rows that are stored in 100 blocks; if needed rows are contained in each of 100 blocks - there is no sense to use index to get rows and full table scan might be chosen). On the other hand, a full scan may not be chosen even when a large percentage of the rows are returned.
* **Throwaway rows (CPU overhead).** Always remember that whether or not a full scan will be an effective choice depends on the number of blocks that will need to be read as much as on how many rows will end up in the final result set. How the data is stored plays an important role in the decision, as demonstrated in last example. However, the other key factor in whether or not a full scan is an effective choice is throwaway. Throwaway rows are those rows that are checked against a filter predicate and don’t match the filter and are thus rejected from the final result set. In the previous example, the full table scan operation would have to check all 10,000 rows in the table and throw away 9,900 of them to end up with the final result set of 100 rows. In order to execute this filter, the CPU will be utilized for each check. That means that while the number of blocks accessed will be limited, there will be quite a bit of CPU resources used to complete the filter checks for each row. The use of the CPU will be factored into the cost of the full scan. As the number of blocks accessed and the amount of throwaway increases, the more costly the full scan will become. Over time, as rows are added to the table and the table grows larger, the cost of throwing away so many rows would increase enough to cause the optimizer to switch to an index scan operation instead.
* **Multiblock reads.** Another thing you need to know about full scans is how blocks are read. A full scan operation makes multiblock reads. This means that a single IO call will request several blocks instead of just one. The number of blocks requested will vary and can actually range anywhere from one to the number of blocks specified in the db\_file\_multiblock\_read\_count parameter. For example, if the parameter is set to 16 and there are 160 blocks in the table, there could be only 10 calls made to get all the blocks. I say that only 10 calls *could* be made because there are limitations on multiblock read calls (crossing extents boundary, block is already cached and multiblock read includes this block, exceed an operating system limit for multiblock read sizes).
* **High Water Mark.** A final point of note regarding full table scans is that as the multiblock read calls for the scan are made, Oracle will read blocks up to the highwater mark in the table. The highwater mark marks the last block in the table that has ever had data written to it. To be technically correct, this is actually called the low highwater mark. When a full scan operation occurs, all blocks up to the highwater mark will be read in and scanned, even if they are empty. This means that many blocks that don’t need to be read because they are empty will still be read. The overhead of reading additional empty blocks can mean performance takes a significant hit. For tables that are frequently loaded and unloaded (using DELETE instead of TRUNCATE), you may discover that response time suffers. This occurs often with tables that are used for ETL or any form of load/process/unload activity.

## Index Scan Access Methods

Rowid points to the exact location of a particular row. Therefore, when an index is used to access a row, all that happens is that a match is made on the access criteria provided in the predicate, then the rowid is used to access the specific file/block/row of data. Block accesses made via an index scan are made using single-block reads. That makes sense when you consider how the rowid is used. Once the index entry is read, only the single block of data identified by that rowid is retrieved; once it is retrieved, only the row specified by the rowid is accessed.

What this means is that for each row that will be retrieved via an index scan, at least two block accesses will be required: at least one index block and one data block. If your final result set contains 100 rows and those 100 rows are retrieved using an index scan, there would be at least 200 block accesses required. I keep saying “at least” because depending on the size of the index, Oracle may have to access several index blocks initially in order to get to the first matching column value needed.

There are several different types of index scans but each share some common ground in how they must traverse the index structure to access the leaf block entries that match the values being searched. First, the root block of the index is accessed with a single block read. The next step is to read a branch block. Depending on the height of the index, one or more branch blocks may need to be read. Each read is for a separate single block. Finally, the first index leaf block that contains the start of the index entries needed is read. If the height of an index is 4, to get to the leaf block needed, 4 single block reads will be performed. At this point, the rowid for the first matching index value in the leaf block is read and used to make a single block read call to retrieve the table block where the entire row resides. Therefore, in this example, to retrieve a single row from a table using an index, Oracle would have to read 5 blocks: 4 index blocks and 1 table block.

The various index scan types you will review are index range scan, index unique scan, index full scan, index skip scan, and index fast full scan. An index fast full scan is actually more like a full table scan.

The clustering factor statistic of an index helps the optimizer generate the cost of using the index and is a measure of how well ordered the table data is as related to the indexed values. Recall that index entries are stored in sorted order while table data is stored in random order. Unless an effort has been made to specifically load data into a table in a specific order, you are not guaranteed where individual rows of data will end up. This statistic is used by the optimizer during query optimization to determine the relative efficiency of an index. In short, the index clustering factor is a measure of how many I/Os the database would perform if it were to read every row in that table via the index in index order. If the rows of a table on disk are sorted in about the same order as the index keys, the database will perform a minimum number of I/Os on the table to read the entire table via the index. That is because the next row needed from an index key would likely be the next row in the table block as well. The query would not be skipping all over the table to find row after row—they are naturally next to each other on the block. Conversely, if the rows in the table are not in the same order on disk as the index keys—if the data is scattered—the query will tend to perform the maximum number of I/Os on the table, as many as one I/O for every row in the table. That is because as the database scans through the index, the next row needed will probably not be on the same block as the last row. The database will have to discard that block and get another block from the buffer cache. The query will end up reading every block from the buffer as many times as it has rows on it.

Therefore, if a table and an index key are in about the same order, the clustering factor will be near the number of blocks in the table and the index will be useful for very large index range scans and for retrieving numerous rows from the table. On the other hand, if the data is randomly scattered, the clustering factor will be near the number of rows in the table, and given that the number of rows in a table is usually at least an order of magnitude more than the number of blocks, the index will be less efficient for returning numerous rows. For example, if a table is 100 blocks in size and has 100 rows per block, an index with a clustering factor of 100 (near the number of blocks) will perform about 2 I/Os against the table to retrieve 200 rows. That is because when the database reads the first table block to get row #1, rows 2–100 are probably on that same block, so the query will be able to get the first 100 rows by reading the table block once. The process will be similar for rows 101–200. If the index has a clustering factor of 10,000—the number of rows in the table—the number of I/Os required against the table will be approximately 200, even though there are only 100 blocks! That is because the first row in the index will be on a different block than the second row, which, in turn, will be on a different block than the third row, and so on—the database will probably never be able to get more than one row from a table block at a time.

### Index Unique Scan

An index unique scan is chosen when a predicate contains a condition using a column defined with a UNIQUE or PRIMARY KEY index. These types of indexes guarantee that only one row will ever be returned for a specified value. In this cases, the index structure will be traversed from root to leaf block to a single entry, retrieve the rowid, and use it to access the table data block containing the one row. The TABLE ACCESS BY INDEX ROWID step in the plan indicates the table data block access. The number of block accesses required will always be equal to the height of the index plus one unless there are special circumstances like the row is chained or contains a LOB that is stored elsewhere.

### Index range scan

An index range scan is chosen when a predicate contains a condition that will return a range of data. The index can be unique or non-unique as it is the condition that determines whether or not multiple rows will be returned or not. The conditions specified can use operators such as <, >, LIKE, BETWEEN and even =. In order for a range scan to be selected, the range will need to be fairly selective. The larger the range, the more likely a full scan operation will be chosen instead.

A range scan will traverse the index structure from the root block to the first leaf block containing an entry matching the specified condition. From that starting point, a rowid will be retrieved from the index entry and the table data block will be retrieved (TABLE ACCESS BY INDEX ROWID). After the first row is retrieved, the index leaf block will be accessed again and the next entry will be read to retrieve the next rowid. This back-and-forth between the index leaf blocks and the data blocks will continue until all the matching index entries have been read. Therefore, the number of block accesses required will include the number of branch blocks in the index (this can be found using the blevel statistic for the index) plus the number of index entries that match the condition multiplied by two. You have to multiply by two because each retrieval of a single row in the table will require that the index leaf block be accessed to retrieve the rowid and then the table data block will be accessed using that rowid. Therefore, if the example returned 5 rows and the blevel was 3, the total block accesses required would (5 rows x 2) + 3 = 13.

If the range of entries matching the condition is large enough, it is likely that more than one leaf block will have to be accessed. When that is the case, the next leaf block needed can be read using a pointer stored in the current leaf block that leads to the next leaf block (there’s also a pointer to the previous leaf block). Since these pointers exist, there is no need to go back up to the branch block to determine where to go next.

One final nuance of an index range scan that I’d like to note is the ability of an ascending ordered index (the default) to return rows in descending sorted order. The optimizer may choose to use an index to access rows via an index even if a full scan might be warranted. This may occur when the query includes an ORDER BY clause on a column that is indexed. Since the index is stored in sorted order, reading rows using the index will mean the rows are retrieved in sorted order and the need to do a separate sort step can be avoided. But, what if the ORDER BY clause is requested in descending order? Since the index is stored in ascending order, the index couldn’t be used for a descending order request, could it? In this case, the index entries are actually read in reverse order to avoid the need for a separate sort.

### Index Full Scan

An index full scan is chosen under several conditions including: when there is no predicate but the column list can be satisfied through an index on a column, the predicate contains a condition on a non-leading column in an index, or the data can be retrieved via an index in sorted order and save the need for a separate sort step.

An index full scan operation will scan every leaf block in the index structure, read the rowids for each entry, and retrieve the table rows. Every leaf block is accessed. This is often more efficient than doing a full table scan as the index blocks will contain more entries than the table blocks will, therefore fewer overall blocks may need to be accessed. In cases where the columns needed to satisfy the column list are all present as part of the index entry, the table access step is avoided as well. This means that choosing an index full scan operation will be more efficient than reading all the table blocks.

You may have noticed in the last example that the index full scan operation also has the ability to read in descending order to avoid the need for a separate descending ordered sort request. There is another optimization for index full scans. This optimization occurs when a query requests the minimum or maximum column value and that column is indexed.

### Index Skip Scan

An index skip scan is chosen when the predicate contains a condition on a non-leading column in an index and the leading columns are fairly distinct. In earlier releases of Oracle, if a predicate used a column that wasn’t the leading column in an index, the index couldn’t be chosen. This behavior changed in Oracle version 9 with the introduction of the index skip scan. A skip scan works by logically splitting a multi-column index into smaller subindexes. The number of logical subindexes is determined by the number of distinct values in the leading columns of the index. Therefore, the more distinct the leading columns are, the more logical subindexes would need to be created. If too many subindexes would be required, the operation won’t be as efficient as simply doing a full scan. However, in the cases where the number of subindexes needed would be smaller, the operation can be many times more efficient than a full scan as scanning smaller index blocks can be more efficient than scanning larger table blocks.

### Index Fast Full Scan

An index fast full scan is more like a full table scan than like other index scan types. When an index fast full scan operation is chosen, all the index blocks are read using multiblock reads. This type of scan is chosen as an alternative to a full table scan when all the columns needed to satisfy the query’s column list are included in the index and at least one column in the index has the NOT NULL constraint (or query predicate condition makes result set NOT NULL).In this case, the data is accessed from the index instead of having to access table blocks. Unlike other index scan types, the index fast full scan cannot be used to avoid a sort since the blocks are read using unordered multiblock reads.

A fast full index scan reads the entire index, unsorted, as it exists on disk. It is basically using the index as a “skinny” version of the table. The query in question would be accessing only attributes in the index. (We are not using the index as a way to get to the table—we are using the index instead of the table.) We use multiblock I/O and read all the leaf, branch, and root blocks. We ignore the branch and root blocks when executing the query and just process the (unordered) data in the leaf blocks.

A full index scan reads the index a block at a time, from start to finish. It reads the root block, navigates down the left-hand side of the index (or the right-hand side for a descending full scan), and then when it hits the leaf block, it reads across the entire bottom of the index—a block at a time—in sorted order. It uses single-block, not multiblock, I/O for this operation.

Now, let’s address the misconceptions the questioner “read elsewhere.” It is true that a fast full index scan is an alternative to a full table scan when the index contains all the columns referenced in the query, but “at least one column in the index key has the NOT NULL constraint” is not a requirement. We can prove this by example. Suppose we have the following table:

SQL> create table t

2 as

3 select \*

4 from all\_objects;

Table created.

SQL> alter table t

2 modify owner null;

Table altered.

SQL> create index t\_idx

2 on t(status,owner);

Index created.

SQL> desc t

Name Null? Type

——————————— ———————— ————————————

OWNER VARCHAR2(30)

OBJECT\_NAME NOT NULL VARCHAR2(30)

...

STATUS VARCHAR2(7)

As you can see, neither STATUS nor OWNER is defined as NOT NULL. However, if we execute the next query we will observe a fast full index scan in action. That’s because we can use the index in place of the table (the index is acting as a skinny version of the table), and the predicate, where owner = ‘SCOTT’, makes it such that the OWNER column must be NOT NULL to be in the result set. And because any NOT NULL value would appear in the index, we can use the index. If the query were SELECT COUNT(\*) FROM T, it is true that we would not be able to use the index with a fast full scan instead of the table, because the table could have rows that are not in the index. Any row in which STATUS was null and OWNER was null would not appear in the index, and we’d miss counting it. But because the WHERE clause explicitly says that we are looking for a not-null value of OWNER, we know that every row I need appears in the index.

# Source Books and Articles

1. Kyte T. Expert Oracle Database Architecture: Oracle Database 9i, 10g, and 11g Programming Techniques and Solutions, Second Edition. Apress, 2010.
2. Morton K., & Osborne K., & Sands R., & Shamsudeen R., & Still J. Pro Oracle SQL. Apress, 2013.
3. Greenwald R., & Stackowiak R., & Stern J. Oracle Essentials, Fifth Edition. Sebastopol: O’Reilly Media, Inc., 2013.