In Defense of Full-Table Scans

By Jeff Maresh

Often times in life, we make decisions based upon the Law of Primacy. This law implies that what you learned first is what you recall first. This occurs even after new concepts appear that contradict what was originally learned. The adage, "old habits die hard" is the popular version of this law. For DBAs and developers, we often see the Law of Primacy in action when it comes to full-table scans. If you've been around Oracle for any period of time, you are well familiar with the contention that, "full-table scans are bad" *and that they must be eliminated to improve database performance. This thought prevails despite the fact that significant incremental improvements have sneaked into the Oracle server over successive versions. Yet these versions improve the utility and performance of full-table scans.*

So just what is a full-table scan? It's simply the process of a query reading all of the data blocks in a table in sequence. When a table scan occurs, no indexes are used. If the table is large, a table scan can take a considerable amount of time to complete, and may consume a lot of bandwidth on the disk subsystem.

History

For many years, full-table scans have been blamed for poor query and database performance. And rightly so. Queries often performed poorly because of a missing index or because the query was constructed in such a way that an available index could not be used. While the overall performance impact to the database caused by a few full-table scans may not have been significant, many full-table scans performed concurrently could bring a database to its knees. DBAs and developers could achieve hero status simply by optimizing queries to rid the database of full-table scans.

Beginning with version 7 of Oracle Enterprise Server, Oracle has employed a two-pronged approach to dealing with full-table scans. First, there have been considerable improvements in the buffer cache architecture and algorithms to handle data blocks more effectively and efficiently. The amount of buffer cache space occupied by scanned blocks was significantly limited by modifying the Least Recently Used (LRU) list algorithm. The LRU is a data structure that is used to determine when blocks in the buffer cache that have not been recently accessed should be replaced with new ones. Blocks were also loaded onto the LRU list on the *least recently used* side assuring that they would be the first to be flushed out of the cache. While this did not fix the performance problems of particular queries caused by poor indexing, it largely rectified poor overall database performance caused by many full-table scans.

More recent improvements in the way that blocks are aged out of the buffer cache using *touch count* algorithms have further improved overall database performance. With the *touch count* algorithm, each LRU is split in half and blocks start out in the middle. Blocks that have a lot of interest migrate towards the hot side of the LRU list. Blocks that have little interest migrate towards the cold side and are eventually removed from the LRU list. Conventional LRU list processing as it existed in earlier versions of Oracle is a thing of the past.

In addition to the improvements to the buffer cache architecture and algorithms, Oracle added features that could take advantage of full-table scans to achieve optimal performance. Many of these features were introduced as the cost-based optimizer began to mature. In Oracle 7.3, hash joins and the Parallel Query Option (PQO) were introduced. Both of these new features could take advantage of full-table scans. Specifically, hash joins are much more efficient than other join methods when joining a small row set to a large one. This type of join occurs in OLTP environments when joining a small code or reference table to a much larger table. Parallel Query was useful for processing large amounts of nonselective rows from large tables.

The impact of full-table scans on the buffer cache was further reduced by the introduction of Direct Path operations. Blocks processed using Direct Path operations bypass the buffer cache altogether. In addition to eliminating the space required to hold the scanned blocks, contention for the various latches associated with the buffer cache is eliminated. Direct Path operations are useful when there is little chance of blocks being reused by other processes. Direct Path reads most often occur during parallel full-table scans on larger tables. SQL*Loader and INSERT statements can also utilize Direct Path writes. Direct Path operations offer significantly better performance than their *conventional path* counterparts and reduce the overall negative database impact of full-table scans. Obviously, the behavior of full-table scans has undergone a significant evolution during the past few years and in many cases, should not be treated as the outlaws they once used to be.

A Practical Example

To gain a better understanding of when full-table scans perform better than index access methods, a practical example is in order. A table and two indexes were carefully constructed for the illustration. The table has three

columns. The first column is a numeric surrogate key, which will serve as a unique identifier. The second column contains a numeric code value that could represent a customer type code. It is populated uniformly with values between 0 and 99. The third column is a 200-character VARCHAR2 field that could represent a description. To achieve a nearly uniform row length, the description column is always fully populated with a 108-character string. The value of PCTFREE is set to the default value of 10, which leaves 10% of the block free for updates after rows are inserted.

The number of logical reads required to retrieve the data is used as the primary performance metric. A logical read occurs when a block of data is read from the buffer cache. The number of logical reads is invariant for a particular query regardless of the database load, number of physical reads, and the execution time. Since between 30 and 50% of total CPU time is spent performing logical I/O, it also takes into account a component of CPU time spent satisfying the query. Logical I/O is a good performance metric because internal work on the database is measured in blocks processed rather than rows. Minimizing the number of logical reads usually results in the fastest execution time and smallest amount of CPU time used. After spending a considerable amount of time optimizing queries, one is bound to find exceptions to this rule. So although there is no perfect performance metric, logical reads is the least flawed metric.

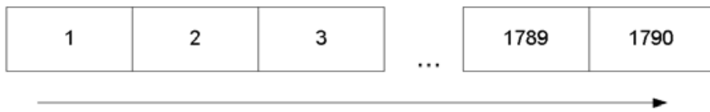

Figure 1 – A Full-Table Scan

The empty table is now populated with 100,000 rows and we find that the data are contained within 1,790 blocks. In the absence of any indexes, all queries against the table must perform a full-table scan. This will require reading all 1,790 blocks (Figure 1).

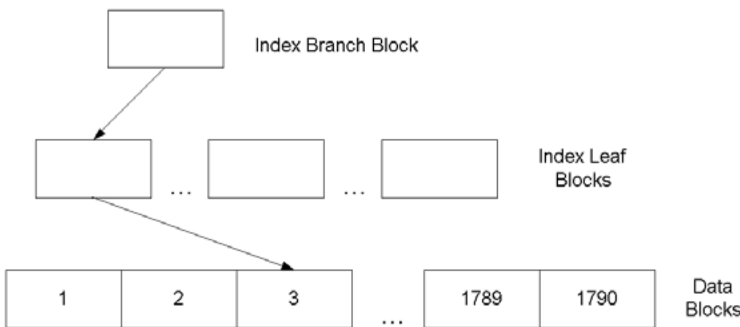

Figure 2 – A Unique Index Scan

Next, a unique B*Tree index is created on the surrogate key column. Since there are 100,000 rows in the table, the index has 100,000 unique values. The index has a total of two branch blocks and 222 leaf blocks. To retrieve any row from the table, a total of three logical reads are required; two reads to retrieve the ROWID from the index and another to read the block from the table (Figure 2).

Last of all, a nonunique B*Tree index is created on the code column. Each of the 100 code values is found in 1,000 table rows. The index has one branch block and 307 leaf blocks. The number of logical reads required to retrieve all of the rows associated with a particular code value is not obvious. The reason is that one doesn't know how the code values are distributed among the data blocks in the table. For this case, the code value was incremented after each row was inserted. When the code reached a value of 99, it was reset to zero and the process was repeated. This procedure was done 1,000 times as shown in the following illustration.

```
0,1,2,3,4,5,6,7,8,9,10…93,94,95,96,97,98,99
…
0,1,2,3,4,5,6,7,8,9,10…93,94,95,96,97,98,99
```

The result is that each table block contains few rows with the same code value. The rows containing each value are evenly distributed among all of the blocks in the table. When the index is accessed for a particular code value, the ROWIDs correspond to many table blocks.

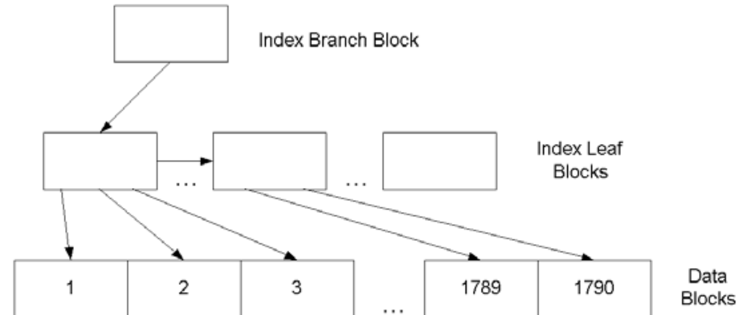

Figure 3 – A Non-unique Index Scan

It requires 503 logical reads to retrieve all of the table rows for any one of the 100 code values (Figure 3). This represents one extreme of the many data loading possibilities for the table. At the other extreme, the code values could have been loaded in the following order.

```
0,0,0,0,0,0,0,0,0,0…0,0,0,0,0
1,1,1,1,1,1,1,1,1,1…1,1,1,1,1
…
98,98,98,98,98,98,98,98,98,98…98,98,98,98,98
99,99,99,99,99,99,99,99,99,99…99,99,99,99,99
```

For this case, all 1,000 rows were inserted for a value of 0. The value was then incremented by 1 and an additional 1,000 rows were inserted. This process was repeated until all 100 code values were loaded. In the table, the result is that all rows with the same code value will occupy a few adjacent data blocks.

The non-unique index is created on the code column. The physical characteristics are the same as the first index with one branch block and 307 leaf blocks. But now the query on a single code value can be satisfied in a fraction of the logical reads compared with the first case.

The query now requires only 53 logical reads to return 1,000 rows (Figure 4). Here, most of the rows for a particular code value are concentrated within a few data blocks. The unique index for this data load case still returns a single row in 3 logical reads.

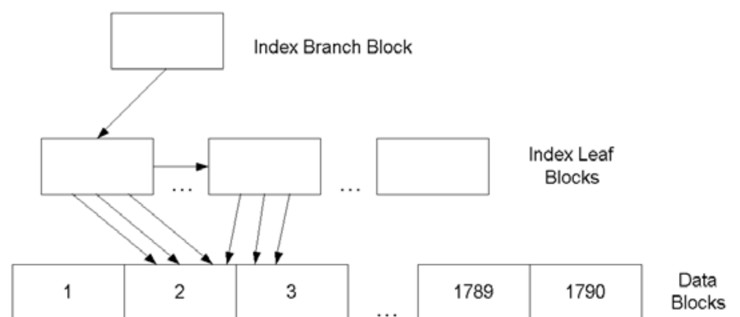

Figure 4 – Another Non-unique Index Scan

continued on page 16

Obviously, there are major differences between the data distribution between the two load methods, hence the performance of the non-unique index. Interestingly enough, the selectivity of the index on both versions of the table are identical at 1%, so it is not a good metric for determining how well the index will perform. This is the source of a many misunderstood performance problems encountered with non-unique indexes.

Clustering factor is a more useful metric for determining the efficiency of a non-unique index. Clustering factor quantifies how rows for a particular index key are dispersed across table data blocks. For two indexes comprised of the same columns, lower clustering factors produce better performance than higher ones. The cost-based optimizer uses clustering factor to determine the efficiency of indexes when determining the table access path. This metric is available in the clustering_factor column in the dba_indexes and dba_ind_partitions table after statistics have been gathered on the index. The range of values for the clustering factor is between the number of data blocks and the number of rows in the table.

In the above example, the clustering factor for the non-unique index in the first case has a value of 100,000, the same value as the number of rows in the table. This represents the least efficient case for the index. In the second case, the clustering factor has a value of 1,786, a value nearly equal to the number of data blocks in the table. This represents the most efficient non-unique index.

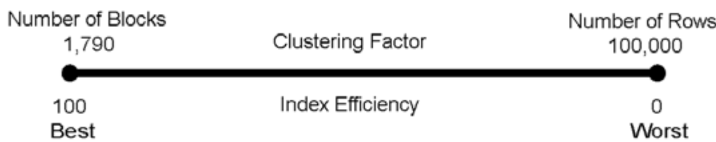

Figure 5 – Continuum of Clustering Factor and Index Efficiency

Since it's difficult to quickly determine the significance of where in the range of two big numbers the clustering factor resides, a measure more simple to comprehend called index efficiency can be computed as follows.

$$IndexEfficiency = 100x\left[1 - \frac{ClusteringFactor - Blocks}{Rows - Blocks}\right]$$

Figure 6 – Computing Index Efficiency

This formula yields numbers between zero and 100, where zero represents the least efficient index and 100 represents the most efficient one (Figure 5). The following table shows the physical and performance characteristics of the unique index and the two non-unique indexes (Table 1).

	Unique Index Index on key column	Non-unique index Case 1 Index on code column	Non-unique index Case 2 Index on code column
Height	2	2	2
Branch Blocks	1	1	1
Leaf Blocks	222	307	307
Distinct Keys	100,000	100	100
Logical Reads/Access	3	503	53
Rows/Access	1	1,000	1,000
Clustering Factor	1,786	100,000	1,786
Index Efficiency	100	0	100

Table 1 – Physical and Performance Characteristics for the Indexes

While the two non-unique indexes in the illustration represent the best and worst possible efficiencies, most non-unique indexes will have index efficiency values across the entire continuum of values. However there are certain cases where the index will have efficiency near one extreme or the other.

Consider a column that records the date of a financial transaction in a table that holds five years of data. If no rows are being deleted from the table so that rows are always being loaded into empty data blocks then the data for a particular day will all be clustered within adjacent data blocks. An index created on this column would be highly selective and have an index efficiency close to a value of 100. This represents a very efficient index.

Now consider a column that records information that is more uniformly distributed as rows are inserted, such as a zip code or a customer type code in a table that records shipping transactions for an internet book seller. Here, the index efficiency will have a value closer to zero. This represents a very inefficient index.

Oracle has recommended using full-table scans instead of indexes when the total number of rows retrieved from the table is above a certain threshold. This threshold value has varied between 1% and 15% depending upon the version of Oracle Enterprise Server being used. That's quite a big range! But based upon the above metrics, we can judge more precisely what access method will produce the best performance for any particular case. The following table shows the number of logical reads for each of the various access methods for retrieving specific numbers of rows from the table. (Table 2)

Percent of Table	Rows Retrieved	Full-Table Scan	Unique Index	Non-unique index Case 1	Non-unique index Case 2
0.001	1	1,790	3	N/A	N/A
1	1,000	1,790	3,000	503	53
5	5,000	1,790	15,000	2,515	265
10	10,000	1,790	30,000	5,030	530
25	25,000	1,790	75,000	12,575	1,325
50	50,000	1,790	150,000	25,150	2,650

Table 2 – Logical Reads for All Table Access Methods

A single table row represents 0.001% of the table. A single row can only be retrieved using the unique index. The unique index is shown in the table for more than just a reference. Consider an application that produces a report, or processes a batch of transactions written in a procedural language (e.g. Java, Visual Basic, etc.) by a developer that is naïve to database performance. The application author may reason that because a query that returns a single row performs much faster than a single query that returns 1,000 rows, the fastest way to generate the entire report is to code it such that one row is retrieved at a time on the primary key. Or the application author may decide that it's easier to incorporate logic procedurally instead of embedding it within a SQL statement. The result is that the application author essentially performs a *nested loops* join inside the report application. Besides the obvious network overhead of all the round trips to the database, one can see from the table that a full-table scan will outperform the unique index when approximately 0.6% of the table is retrieved. The full-table scan requires 1,790 logical reads regardless of how many rows are returned. The unique index can only retrieve less than 600 rows for the same number of logical reads.

When either of the non-unique indexes is used, 1,000 rows are returned for each key accessed. For the first non-unique index, three accesses will consume 1,509 logical reads and return 3,000 rows or 3% of the table. This represents the breakeven point for the full-table scan compared with the worst possible non-unique index since a fourth index access would require more logical reads than the scan. For the second non-unique index, 33 accesses will consume 1,749 logical reads and return 33,000 row, or 33% of the table. The difference between the logical reads required for the two non-unique indexes is more than a factor of 10, simply because of the differences in the way that rows of interest are clustered in the table. This performance difference can be accurately predicted by comparing the index efficiency of 0 for the first case index and 100 for the second case index.

Discussion

Based upon the above case, it becomes obvious that the greatest performance variability occurs with non-unique indexes. For an index created on the same column, the breakeven point for the number of logical reads compared with the full-table scan ranges between 3% and 33% of the table. These figures represent the best and worst possible scenarios for this particular index based upon how the data were loaded. Depending upon numerous factors including the data block size, the amount of free space reserved in the table for updates, and row lengths, the breakeven point for each index will vary from this example. The key point of the illustration is that there is a very high degree of variability in the efficiency of non-unique indexes so one should use the index efficiency metric to help decide if it is suitable for the query at hand. Replacing an index range scan with a full-table scan may improve performance.

Iterative Index Access

Anytime any type of index is accessed repetitively, a full-table scan should be considered as a viable alternative. While poor performance resulting from the incorrect use of indexes in procedural programs was discussed, poor performance can also result from less than optimal access paths in SQL statements. This may occur with a single table access but is more prevalent when one or more tables are joined.

Table access using a unique index will be the most efficient means of retrieving a few rows of data. This type of access occurs frequently in on-line transaction processing applications. An example of this is a Product Ordering Application that is used at a mail order house. When a repeat customer calls to place an order, information about the customer is located using a unique identifier such as a telephone number or customer identification number.

But there are many other times when queries have the opportunity to access table rows by looping over an index. Anytime this behavior occurs, it is worth considering whether or not a full-table scan will produce better results. While it is easy to find examples of cases involving non-unique indexes, let's consider the following query that performs repetitive accesses on a unique index.

```
SELECT o.order_num, o.order_date, o.tax_amt, s.state_nm, s.county_nm,
s.tax_rate
 FROM orders o, tax_info s
 WHERE o.jurisdiction_id = s. jurisdiction_id
 AND o.order_date = '01-FEB-2003';
```

The query joins an order table with a tax information table to produce a daily tax liability report. The resulting execution plan indicates that a *nested loops* join will be performed between the two tables using indexes. Here, rows from the order table for the specified date are retrieved using a non-unique index. This is likely to be the most efficient method because the table holds 5 years of data, so one day is likely to represent far less than 1% of the data. The tax_information table is accessed using a unique index for each row produced from the orders table. The unique index has a depth of 2, so each row retrieved from the tax table requires 3 logical reads. If the report produces a total of 4,000 rows than it requires a total of 12,000 (4,000 x 3) logical reads to retrieve information from the tax_info table. Incidentally, this is the execution plan produced by the rule-based optimizer.

```
NESTED LOOPS JOIN
 TABLE ACCESS BY INDEX ROW ID "ORDERS"
 INDEX RANGE SCAN "ORDERS_IDX1"
 TABLE ACCESS BY INDEX ROW ID "TAX_INFO"
 INDEX UNIQUE SCAN "TAX_INFO_IDX1"
```

The execution plan produced by the cost-based optimizer uses a full-table scan in place of the unique index to access the tax_info table, and a hash join instead of the *nested loops* join. To perform the hash join, the 21 blocks in the tax_info table are scanned and an internal bitmap of the keys is created. As rows are read from the orders table, the value corresponding to the key in the tax_info table is hashed. Rows that match produce a joined row. This process is repeated for each row retrieved from the orders table.

```
HASH JOIN
 TABLE ACCESS FULL "TAX_INFO"
 TABLE ACCESS BY INDEX ROW ID "ORDERS"
 INDEX RANGE SCAN "ORDERS_IDX1"
```

The difference in performance between the two queries is significant. The first execution plan required a total of 12,000 logical reads to retrieve rows from the tax_info table using the unique index. The second plan required only 21 logical reads, or 0.2% of the logical reads of the first plan. While this is an example of a small report query that may run infrequently, there are many opportunities in OLTP environments for this type of optimization. Smaller gains in queries that are executed several times an hour by several thousand customer service representatives in a call center can produce significant overall database performance gains. In most cases, the cost-based optimizer will properly decide when a full-table scan should be performed as long as accurate optimizer statistics have been gathered on all involved objects.

continued on page 18

In Defense of Full-Table Scans continued from page 17

Data Skew

Skewed data presents a query tuning challenge. A skew occurs when data is not uniformly distributed on the key value of interest within a table. Consider a table that holds reporting information about sales shipments and one of the indexed columns is the state where the order was shipped. Orders from populous states such as California and New York each represent more than 20% of the table contents. Data from less inhabited states such as Montana and North Dakota each represent less than 2% of the table. When querying for all orders in the more populous states, it has been determined that a full-table scan is the most efficient access method. But when querying the less populous states, a non-unique index scan produces the best performance. Even in the presence of global statistics, the cost-based optimizer is likely to consistently choose one access method or the other because it is unaware of the skewed data.

Index histograms are helpful to solve this problem by providing the optimizer with the additional information required to choose the correct access path based upon the column value requested in the query. There are a variety of options available for creating index histograms. They can be created with either the ANALYZE command, or the DBMS_STATS package.

Parallel Query

In reporting and data warehouse environments, Parallel Query can be an effective means of processing large amounts of nonselective data using full-table scans. The short synopsis of Parallel Query is that a large table can be broken up into multiple ranges of ROWIDs that are then processed by multiple Parallel Execution (PX) slaves. Each slave essentially performs a mini full-table scan over the assigned range of ROWIDs. In most cases, parallel queries read data using the *direct path read* mechanism, which achieves very high throughput. While this activity may consume a considerable amount of I/O bandwidth, there is little impact on the buffer cache.

Caveats

There are several issues that should be considered when trying to determine which access path is best. In our example table, data were always loaded into an empty table. In the second case where all of the rows were loaded at one time for a particular code value, the data were clustered into a few adjacent data blocks because the inserts occurred on empty data blocks. This resulted in a highly efficient non-unique index. In a real application, this will occur if little or no data is ever deleted from the table. When enough data is deleted from a data block to cause the amount of free space to drop below the value of PCT_USED, the block will be placed on one of the table's free lists, indicating that is once again eligible for new rows to be inserted. When new rows are inserted, any blocks eligible for inserts are retrieved from the free lists and filled up to the value of PCT_FREE before a new table extent is allocated. Because these blocks may reside anywhere in the table, the new data is likely to be distributed to many data blocks rather than concentrated into a few blocks at the leading edge of the table. The result is that although the data loading strategy is optimized to concentrate rows into a few adjacent data blocks, the presence of numerous free blocks on the free lists prevents this from occurring. This effect can be easily detected by inspecting the index efficiency value for the index. Here, a full-table scan may perform better than any index on the table if a significant percentage of rows will be retrieved.

One instance where full-table scans perform particularly poorly is when a large table has a highly dynamic number of rows such as those that occur on equity and commodity order staging tables. Consider a stock-trading database where a table contains open orders. As orders are taken, rows are added to the table. At the peak of the day, the table may contain a million or more rows. Processes periodically query the table to determine when the market prices meet the order conditions. Orders are deleted from the table when conditions are met and the orders are filled. When the market closes, there are likely to be few orders, perhaps less than 10,000, remaining in the table.

Recall that database I/O occurs in units of blocks rather than rows. While there are few rows in the open order table at the end of the day, they are likely to be scattered throughout the table. In our case, there could be 10,000 blocks in the table because that was the capacity required to hold the one million rows at the peak of the day. At the end of the day when there are only 10,000 rows remaining in the table, the table still has 10,000 blocks. Notice that the row density will have dropped from 100 rows per block when it was full, to one row per block at the end of the day. When a full-table scan occurs, all of the data blocks are read up to the table high-water mark. The high-water mark represents the last block in the table that ever contained data. Therefore in this case, 10,000 logical reads would be required to read the entire table. This is a case where index performance is likely to be superior to a full-table scan.

Do You Really Need that Index?

At this point, the case has been made for improving performance with full-table scans when certain conditions exist. Now the question becomes, "When does a table really need a particular index?" For any table size, an index is useful if it produces better performance than a full-table scan. Indexes also support primary and unique key constraints. When considering whether or not an index will produce better performance, the most interesting case is the *small table*. It is pretty obvious that a table contained in a single block doesn't need any index other than to support primary and unique key constraints. One could also easily argue that if a table is housed in 3 or fewer blocks, it too doesn't need additional indexes because a minimum of three logical reads are required to access a table row using an index. One could even argue further that additional indexes may be unnecessary on tables that have fewer blocks than the value of the *db_file_multiblock_read_count* parameter.

This parameter controls the number of blocks that are read from disk in a single I/O operation when a full-table scan occurs. This parameter is dependent on a number of host and I/O subsystem configuration parameters but a parameter value of 8 is common on OLTP systems and may be as high as 256 on data warehouses. If the value of the parameter were 8 then any table that has fewer than 8 blocks will be retrieved in a single physical I/O operation. It's been the author's experience that additional indexes may not be useful when the table is contained in between 3 blocks and the number of blocks corresponding to the value of *db_file_multiblock_read_count*. If the index is used to always retrieve a single row of a table, then an index would be useful if the table were contained in 4 or more blocks since the table row could be retrieved in 3 logical reads. However, if an index is usually used repetitively in a query to retrieve multiple values, then an index on a particular column will probably not be useful. If such an index is present, the cost-based optimizer is likely to correctly choose a full-table scan over the index. Recall this behavior in above hash join example. The definitive answer comes through testing performance against the table and index with the queries that will actually run against them. The point to be made here is that there have been significant enough changes to the database architecture to warrant a fresh look at the usefulness of certain indexes.

Summary

Full-table scans may perform better than non-unique indexes when even smaller percentages of the table are being retrieved. They may also perform better than both unique and non-unique indexes when the index is accessed repetitively in either a query, or an application that performs multiple accesses to the same table. The *index efficiency* computed on the *clustering factor* is a good metric to help determine how well a non-unique index will perform. Ultimately, performance will be optimized by testing the various access methods available for the query of interest. The size of the table should be taken into account when considering the creation of a new index. For small tables, indexes may be unnecessary, other than those required to support primary and unique constraints. While the most efficient access path for most queries is likely to use an index, there are quite a few cases where a full-table scan will produce superior performance. With all of the improvements that have been made to improve the utility and performance of full-table scans, it should now be considered as a viable table access method to improve performance.

References

Harrison, Guy. 2000. *Oracle SQL High-Performance tuning, Second Edition*. Prentice-Hall, Inc.

Oracle 9i Database Reference Release 2 (9.2). 1996, 2002. Oracle Corporation.

Oracle 9i Database Performance Tuning Guide and Reference Release 2 (9.2). 1996, 2002. Oracle Corporation.

Shallahamer, Craig. 2001. *All about Oracle's Touch-Count Data Block Buffer Algorithm.* OraPub, Inc.

About the Author

Jeff Maresh is an Oracle Database Architect, DBA, and application developer with more than19 years of consulting experience in the IT industry. His areas of specialty include data warehousing, process automation, and database performance tuning. During the past several years, he has written and taught a number of training courses on PL/SQL, database tuning, data warehousing, and PERL, and has provided technical mentoring. Jeff has provided consulting services to large corporations primarily in the telecommunications and oil and gas industries. He has been a speaker at a number of Oracle user group meetings, written articles for a number of Oracle related publications, and is a Contributing *SELECT* Editor. Aside from work, Jeff enjoys camping, hiking, mountain biking, and cross country skiing in Colorado. Jeff can be reached at jeffery.maresh@verizon.net. Other papers and presentations are available at www.evdbt.com.