Optimizing the indexes for an Oracle schema

Dave Ensor, BMC Software, Inc.

Introduction

The performance of an Oracle-based application depends, amongst other things, on appropriate indexing of the tables used by the application. Most applications go into production with indexing based on the implementers' predictions of the requirements, sometimes adjusted in the light of experience during testing. Even if this index set is initially effective, it will require revision over time in the light of code changes and enhancements, changes in usage patterns, and changes in data distribution.

Code changes are often simply not permitted to mission critical applications, especially when running a packaged solution to a business problem. Once the application is live, and basic instance tuning has been performed, changes to index strategy represent the only opportunity for significant improvements to performance without the need to change the application code.

Before discussing how we can assess the effectiveness of an index strategy, and reform it if required, the paper reviews key properties of Oracle's index support as the effectiveness of an index strategy depends in large part on the validity of the assumptions on which it is based.

Index Benefits and Costs

Index Benefits

The main benefit of having an index on a table is that it allows table rows with specific values to be located more quickly than would be possible if the entire table had to be read searching for the rows.

Unique indexes, including primary key indexes, also have the valuable effect of maintaining "guaranteed addressibility" by ensuring that each possible value of the key on which they are based will appear either zero or one times in the underlying table. It is a common feature of applications that many of the unique keys are *concatenated* as in the following fragment from a table creation script:

```
, constraint ORDER_LINES_PK primary key (CUST_ID, ORDER_NO, LINE#)
, constraint ORDER_LINES_FK foreign key (CUST_ID, ORDER_NO)
                            references   ORDERS
```

An index like the implicitly created primary key index in the above example may serve (at least) three purposes:

- to ensure that duplicate order lines are not stored within the database,
- to allow efficient retrieval of all the lines for a given order if the full key of the order is known, and
- to allow deletes and primary key updates against the table ORDERS to efficiently check for the existence of ORDER_LINES for that order. This usage is worth bearing in mind because it has severe implications for table locking, but cannot be discovered from inspection of Oracle's execution plans.

This index also allows the efficient retrieval of a specific order line if its full key is known, but as most order processing functions deal with complete orders rather than single lines, this may well

not be a significant advantage. When dealing with indexes it is not uncommon to discover that the most obvious use of the index is actually the least important during normal application running.

Index Costs

Indexes must be created and maintained, which both uses CPU time and causes additional physical I/O. Except in the special case of operations conducted with NOLOGGING in effect, the additional physical I/O extends beyond the index blocks themselves to rollback segments and the redo log.

As a very general rule of thumb, if it is one unit of work to insert a row into a simple table then it will be three units of work to insert an index entry for the row into a B*tree index. So having 10 indexes on the most maintained table in your most DML intensive web application is not a good idea. Indexes also consume space on disk, and they must be included within backups unless their re-creation is part of the restore process.

Finally, for any given index, there may be certain queries for which the index might be used that will perform better if it is not used. This is much more of a problem for applications still running under Oracle's rule based optimizer than it is for applications using Oracle's cost based optimizer (CBO).

Index Types

Oracle has two basic index types with quite different properties. They are discussed briefly below, and fortunately it is rare for there to be any real doubt as to which type of index should be employed in a given circumstance. A third mechanism, hash clustering, is briefly mentioned for completeness but it is little used and not generally recommended.

Oracle Release 8.1.5 and up allows expressions to be used as components of index keys, whereas all previous Oracle releases supported only the use of column names. This is an important new feature that removes a number of design and performance difficulties, but it makes little or no difference to the issues discussed in this paper. The reader is simply asked to remember that wherever there is a reference to an index key, it might well be an expression e.g.

```
create index CUSTOMER_NAMES on CUSTOMERS(upper(NAME));
```

B*Tree Indexes

This is Oracle's traditional index type. Each index comprises a *sequence set* listing all the index keys and their target rowids in row order within key order, and a *branch set* providing efficient lookup of the sequence set block in which a particular key will be present if it exists within the index. Completely null keys are never stored but partially null keys can be stored, and are only retrievable through the index in specific circumstances. For practical purposes the statement that B*tree indexes do not contain null keys can be considered to be true.

B*Tree indexes are often the subject of index range scans to support queries such as

```
select <column list>
  from ORDERS
 where ORDER_DATE between to_date('01-Jan-2000', 'dd-Mon-yyyy')
                      and to_date('01-Feb-2000', 'dd-Mon-yyyy');
```

Oracle's Cost Based Optimizer (CBO) may also elect to perform an INDEX FAST FULL SCAN on an appropriate index in order to avoid a full table scan. Such operations are typically I/O limited and the total size of the sequence set becomes a significant determinant of the query execution time. In earlier versions of Oracle the complete key was stored for every target row, but Oracle Releases

8.1.5 and up allow key compression to be specified for an index. This option can greatly reduce the size of the index with little negative performance impact on DML but significant positive impact on index range scans.

B*tree indexes are used to support the primary and unique key constraints enabled for a table. They should also be used for retrieval based on keys with at least 20 distinct values which are the subject of predicates that will either retrieve less than 5% of the table or will be fully satisfied from the index key.

The author does not normally endorse the practice of arbitrarily extending a concatenated key so that a specific query can be resolved directly from the index.

IOT's – A Special Case

At its simplest, an IOT (Index Organized Table) is a B*tree primary key index that also contains the table columns that are not part of the primary key. This can have both performance and space advantages where the primary key is a substantial percentage of the total row size. On the other hand secondary indexes on IOT's require significantly more space than secondary indexes on conventional tables and can incur a performance overhead.

Bit Mapped Indexes

The proposition is simple. For each distinct key value that actually occurs within the table, a compressed bit map is held to identify, one bit for each row, whether the row does or does not contain that key value. When the number of distinct key values is low, a bit mapped index can use an order of magnitude less space than its B*tree equivalent.

It is worth noting that NULL values are supported by bit mapped indexes.

These indexes are at their most efficient when used to retrieve data based on the intersection of a set of key values each of which is relatively unselective e.g.

```
select <column list>
  from ORDER_HISTORY
 where SALES_REGION   = 'SE'
   and DEAL_TYPE      = 4
   and YEAR           = 99
   and CUSTOMER_TYPE  = 6
   and DISCOUNT_LEVEL = 3;
```

The downside is that row updating is very expensive on a large table with bit mapped indexes. Firstly the (compressed) bit map has to be rebuilt, and secondly Oracle's row level locking algorithm treats each bit map as a row in its own right. Since bit map indexes tend to be relatively unselective, the result is often that a significant proportion of the table is subject to an exclusive lock for the duration of the transaction.

Bit mapped indexes are best employed for non-selective keys used in combinations against tables that are not updated in real time.

Oracle Table Clusters

Most of the ideas expressed in this paper assume that the application schema does not contain Oracle table clusters.

In the author's experience application schemas hardly ever specify clusters even though they are extensively used by Oracle's data dictionary. If table clusters are used then joins between rows that share the same cluster key should be highly efficient. However table clusters are sensitive to the number of rows per cluster key, and the more this varies the more difficult it is to tune the cluster

structure to conserve space. Once the data to be stored per cluster key becomes a multiple of the database block size then performance difficulties are usually encountered.

Table clusters support an additional access method, hashing. The author has never seen this feature used to beneficial effect and it is not discussed further in this paper.

Effects of Normalization

The author firmly believes that all Oracle schemas other than data warehouses should be in at least third normal form (3NF). One of the standard effects of normalization is that it increases the number of tables in the schema, and this is often perceived as negatively impacting performance. However this increase in tables is usually accompanied by a corresponding decrease in both the number of indexes per table and the total number of indexes required (or at least perceived to be required). As has already been discussed, indexes usually cost more to maintain than the underlying table, so the net performance impact is positive for DML operations.

Oracle's rule based optimizer often makes spectacularly bad index choices when there is more than one index on a table that could be used to satisfy a particular query. Reducing the number of indexes has the valuable side effect of reducing the opportunities for the optimizer to make this type of error.

Optimizer Choice

This paper is not optimizer specific, but it is almost certainly easier to arrive at an optimal index strategy using Oracle's Cost Based Optimizer (CBO) than it is under the rule based optimizer because of the latter's habit of making apparently random choices.

Over the years the author has roundly criticized earlier versions of CBO, but it is getting more and more difficult to find valid arguments for not using it. Amongst many other advantages, it now seems reliable at finding the better of two non-unique indexes, avoiding the problem discussed above under normalization. Recently it has come to light that a side effect of reducing `optimizer_max_permutations` can markedly improve CBO's selection of the driving table for a query.

Why go Schema-wide?

Although there are a number of Oracle-based applications that perform all of their joins within application code, the underlying thesis behind this paper is that when adjusting index strategy no table or query should be considered in complete isolation. Every change is almost certain to have a series of effects elsewhere in the application, and the goal is to arrive at an efficient compromise. Because of the space and maintenance overheads incurred by unnecessary indexes, particular emphasis is placed on reducing the index set to the minimum necessary for efficient operation.

SQL Execution History

As discussed in a later section, there are some indexes (such as those enforcing primary key constrains) that are necessary whatever the load or the distribution of key values. Many indexes, however, exist solely to improve the performance of specific queries or query types. Especially in packaged applications, these need to be reviewed against the production load to see whether they are effective.

No two sites are identical in either their key distributions or their usage of application functions, and even within a single schema both the key distribution and the usage of application functions will vary over time.

To cite a trivial example, an Order Processing package may have the ability to handle payments in any convertible currency but be installed at a site whose entire customer base is in the continental

United States and which deals exclusively in US dollars. Non-unique indexes on currency code against the various transaction tables will simply waste maintenance time, back-up time and disk space, as well as confusing the rule based optimizer (if enabled).

Basing index strategy on the SQL execution history often gives rise to the objection that some jobs that are only run occasionally, such as year-end tasks, are unlikely to appear in the history. The author perceives this to be rather more of an advantage than a disadvantage. In general we should be tuning databases to support their on-going load in preference to distorting our solution to support essentially one-off activities.

Sources

There are a number of sources available from which SQL execution history can be captured. Two of these are reviewed below.

V$ Views

The information in V$SQLAREA is sufficient to build a good history of SQL execution at the instance level. This source is too expensive in CPU terms to be used for short interval polling, but if the shared pool has been tuned at all well then SQL statements will persist in the cache for several tens of minutes. Thus even relatively infrequent polling of the cache will find them. If the tuner is determined to sustain short polling intervals then the SQL statements active at any point in time can be found from V$SESSION and V$SQLAREA e.g.

```
select SQL_TEXT
  from V$SESSION S
     , V$SQLAREA A
 where S.STATUS          = 'ACTIVE'
   and S.SQL_HASH_VALUE = A.HASH_VALUE
   and S.SQL_ADDRESS    = A.ADDRESS;
```

One difficulty here is that the column SQL_TEXT is limited to 1,000 characters so it is better to go to V$SQL_TEXT where the entire statement can be retrieved in 64 character chunks. Sadly there are bugs in this part of the Oracle server in current releases, and in Release 8.1.5 data can only be reliably retrieved from V$SQL_TEXT by joining to it from V$SQL. The author has been unable to retrieve data from V$SQLTEXT_WITH_NEWLINES, and so it has not been possible to use EXPLAIN PLAN to process statements containing comments starting "--" because the end of the comment cannot be found.

The views V$SQLAREA and V$SQL are identical except that the former contains a GROUP BY clause which reduces the number of rows that must be processed when retrieving instance-wide execution history.

Oracle's sql_trace facility

This facility, which can be enabled at either system or session level, presents a simple and relatively complete history of SQL statement execution. Although Oracle does not officially support user interrogation of the trace files, they are in character format and almost trivial to interpret. A major benefit is that in addition to CPU consumption data, these trace files also contain the execution plans used which removes all of the difficulties associated with the use of Explain Plan against a history.

Unfortunately enabling this facility at instance level causes an excessive amount of operating system serial output to the trace file directory, and this alone is sufficient to dissuade most DBA's from enabling it. The serial I/O volume can still be a moderate concern when the feature is enabled only for selected sessions, and it may be difficult to find a valid sampling technique. A few years ago the author became aware of a site that ran one percent of production database connections with

`sql_trace` set to `true`. They reckoned that over a few days this was enough to capture a completely representative sample of SQL execution.

It should be noted that although the execution plans used are contained within the trace file, the objects touched are identified by object_id rather than by name. These references must be resolved before any drop and re-create operations take place against the schema.

Issues with EXPLAIN PLAN

Clearly in any review of index strategy it is essential to know which indexes are being used, and whether they are being used to good effect. V$SQLAREA will tell us about the use of a given SQL statement and summary information about its resource consumption, and EXPLAIN PLAN should be able to tell us which indexes are used by the statement. Unfortunately life is never quite that simple for the reasons outlined below. Use of the `sql_trace` facility bypasses these issues because the execution plan is available within the trace file.

Parsing User ID

One of the major internal optimizations in Oracle Versions 7 and 8 is the way in which parsed SQL statements can be shared between users. The problem is that where a statement has been shared by many Oracle users, the rows describing it in V$SQLAREA show the user ID of whichever user happened to be the first one to parse the statement at a time when it was not present in the cache. This is entirely likely to be a different ID for the same statement encountered at different points in time, and it is desirable to be able to group the resulting statistics according to the underlying objects being referenced. The author has found that this can be done with reasonable efficiency when the statement is first seen in the cache by retrieving the owner(s) of the underlying tables from V$DEPENDENCY.

Optimizer Settings

V$SQLAREA contains the `optimizer_mode` that was used to optimize the query, but gives no clue as to the settings of the other optimizer parameters such as `optimizer_index_caching` and `optimizer_index_cost_adj`. To date the author has been lucky, and has not encountered cases where these have been set at session level in transaction processing applications. There is always the concern that they may have been altered at instance level since the statistics were gathered.

Search Columns

The column SEARCH_COLUMNS in the Explain Plan output is never populated. This is sad, because the attribute was introduced to report how many of the columns of a concatenated index were being used in an index scan. It is possible to imply the likely value from inspection of both the SQL query and the execution plan, but it would make life much easier if Oracle were to report the data as originally planned.

Missing Index References

Execution plans do not show index lookups for the purpose of inserting or deleting a key, though these can be implied from trace files by inspecting the number of database blocks visited in *current mode*. It may appear obvious that index visits will have to occur for each DML operation against a table, but this is not the case for two reasons. Firstly completely NULL keys do not appear in B*tree indexes, and secondly no index operation is performed on UPDATE if the key value is unchanged. The net effect is that the DML load imposed by an index may be overestimated using the INSERT, UPDATE and DELETE counts for the underlying table.

Execution plans also omit the index references incurred by the enforcement of foreign key constraints, and it is important to realize that these look-ups can occur at either end of the constraint.

There is no choice about having a unique index at the master end of the relationship, but a discretionary non-unique index at the detail end may well appear unused from inspection of the query plans.

Finally I am indebted to Jonathan Lewis for pointing out during a recent presentation that execution plans do not show operations incurred in support of Oracle's "Virtual Private Database" functionality.

INSERT statements

It is very useful to know how often tables are subject to DML and to get an idea of the cost. Unless SQL*Loader direct path is being used this information is contained within a SQL statement execution history. Oracle Release 8.1.5 and up allow `MONITORING` to be set on a table to track the number of INSERTS, UPDATES and DELETES performed against the table. However the manuals are less than clear about this option, and no resource utilization information is recorded. If a SQL statement execution history is available then it is likely to be a better source of data about when DML operations have occurred, the number of rows affected, and the resources consumed.

Most DBA's and many application developers are familiar with the SQL command Explain Plan and its results table. A number of scripts exist to display the execution plan, including two scripts developed by Oracle. These are `utlxplp.sql` and `utlxpls.sql` and can normally be found in the `rdbms/admin` directory.

For UPDATE and DELETE statements the second line of the plan output identifies the object that is the subject of the DML. There is no need to worry about whether the name in the SQL statement might be a synonym or an updateable view, you just need to check the second line of the plan and you will immediately know the underlying table that is the subject of the command. Unfortunately this second row does not appear for INSERT statements, so a little more ingenuity is required to rigorously identify the target. If the INSERT does not contain a subquery then during data capture its entry in V$SQLAREA can be joined to V$DEPENDENCY to identify the target table.

If there is a subquery then the processing is more involved. One approach is to extract the target name from the INSERT statement and to use it to fabricate a query of the form

```
SELECT * FROM <target>;
```

If this statement is parsed then by navigating from its cursor to V$DEPENDENCY the schema and name of the target table can be found. Processing the synthetic query through EXPLAIN PLAN is not as effective. It takes longer, and if the table is an IOT (or it has a primary key that includes all of its columns) then the execution plan will be an `Index Fast Full Scan` and the object referenced will be an index rather than the underlying table.

Defining Query Efficiency

In order to have a view as to whether an index is "good" or "bad" it was essential to have a rule for deciding whether a query that used the index was "efficient" or "inefficient". To date the author has been using block visits per row retrieved, deriving this from the columns BUFFER_GETS and ROWS_PROCESSED in V$SQLAREA. Block visits were picked in preference to physical blocks read on the grounds that whether or not a block visit causes a physical read is an instance tuning issue rather than a statement tuning issue.

Current prototype code is using a threshold of 15 for this value, but it is planned to convert this to fuzzy logic using a default boundary range of 12 to 50. It has been observed that although these values can be justified for OLTP queries, they are punitive for queries that contain GROUP BY clauses as it may be necessary to visit very large amounts of data to perform the aggregation. This issue has been deferred for further study.

Classifying Existing Indexes

Once we have overcome the problems of gathering and analyzing a history of SQL statement execution then we can use the data gathered plus information in Oracle's data dictionary to divide the existing indexes into five groups. Originally I referred to these imaginatively as Groups 1 to 5 but more recently I have started to refer to them by the names "No Brainers", "No Worries", "No Hopers", "No Thank You", and "No Use". Until very recently I optimistically referred to the groups as being mutually exclusive, but recent work has forced me to drop this claim.

No Brainers

These are indexes that you know you need whether or not they appear in any execution plans. This is not to say that you should be preserving unused indexes because you are convinced that someday some SQL statement will come along and use them but rather that you may have indexes all of whose uses are hidden. This can happen with a unique index, though it is relatively unusual because most unique keys are used from time to time to retrieve records. Even if the full key is not quoted, the leading edge of the primary key of a detail record is often used to join the detail records to their master.

The "No Brainers" are the unique indexes required to implement primary and unique key constraints, plus any index whose leading edge supports the detail end of a foreign key constraint where the master end is a table routinely subject to DML.

"No Brainers" must be retained, though the order of any concatenated keys could be changed provided that any required foreign keys were still at the leading edge.

No Worries

These are the indexes that appear in the execution plans of efficient queries, and that (ideally) do not appear in the execution plans of inefficient queries. In the current prototype any index that appears in a greater number of efficient executions than inefficient executions is placed in this category.

"No Worries", or other indexes with the same set of columns (or expressions) at their leading edge, should be retained.

No Hopers

These are:

- all non-unique indexes on small tables, currently defined as tables with less than 101 rows and less than 5 data blocks, and
- all non-unique B*tree indexes with less than 20 distinct key values that are not "No Brainers". It is possible for a foreign key to be non-selective but to have a heavily updated table at its master end.
- "No Hopers" should be dropped.

No Thank You

These are indexes that appear solely or predominantly in the execution plans of inefficient queries.

Not only should "No Thank You" indexes be dropped, but also no new or revised index should be introduced that has the same columns at its leading edge. Doing so would incur the risk that the optimizer will generate a similar query plan to the ones that we are implicitly rejecting.

No Use

These are indexes that do not appear in any query plan, and which are not "No Brainers".

If the analysis is correct then a high percentage of "No Hopers" should also appear as "No Use" though in some cases the optimizer will use "No Hopers" for Index Fast Full Scan operations that are of only marginal benefit.

"No Use" indexes should be dropped.

Proposing Additional Indexes

Eliminating unnecessary indexes is only part of the problem. We will still be left with at least some inefficient queries that must be optimized. Taking these queries in order of either their frequency of use or their overall resource consumption (whichever is felt to be the highest priority) new index paths should be proposed as follows:

pick a table order based on the twin goals of minimizing the row set at every stage of the query, and avoiding visiting any individual row more than once with the exception of reference tables, then

for each table from which rows must be retrieved, propose an index path to support that operation based on the predicate clauses and the data available from the tables already visited (if any).

Case Study, Part 1

To give a simple example, consider the following query:

```
select C.CUST_NAME
     , P.PRODUCT_NAME
     , sum(L.VALUE)
  from CUSTOMERS C
     , ORDERS    O
     , LINES     L
     , PRODUCTS  P
 where C.CUST_TYPE   = :ctype
   and O.ORDER_TYPE  = :otype
   and L.LINE_TYPE   = :ltype
   and C.CUST_ID     = O.CUST_ID
   and O.CUST_ID     = L.CUST_ID
   and L.ORDER#      = O.ORDER#
   and L.PRODUCT#    = P.PRODUCT#
 group by C.CUST_NAME
        , P.PRODUCT_NAME
```

If we knew that the Order Type was highly selective, and that the other values supplied as bind variables were not, then we would want to make ORDERS the driving table, and join from there to CUSTOMERS and from there to LINES. This visits a Customer once per Order, which may not be ideal but is certainly better than doing so once per Line and is the only way that we can use the restrictive nature of the Order Type. Finally we join to the Products table.

The proposed index paths that we record are:

```
ORDERS       (ORDER_TYPE)                   to pick up the driving rows
CUSTOMERS    (CUST_ID, CUST_TYPE)           to find and filter the customer
LINES        (CUST_ID, ORDER#, LINE_TYPE)   to find and filter the order lines
PRODUCTS     (PRODUCT#)                     to pick up the product name
```

Of course in the real world it will often more difficult than this with a need to find driving tables for sub-queries as well as the main query. It may also be necessary to take decisions on when to use multiple driving tables and perform hash joins of partial result sets rather than trying to progress step-wise with indexed joins as in the above example. However the typical untuned application makes upwards of 80% of its block visits in the course of 5 or less queries, so not too many of these exercises need to be performed in order to achieve real benefit.

Putting the results together

Once we know which indexes we should keep, and which additional index paths we would like to see introduced, the final stage to define our new index set is to rationalize the existing and proposed indexes into a minimal set per table.

The algorithm for doing this was originally proposed to the author by Chris Ellis who in the 15 years before his retirement built a worldwide reputation for his ability to tune Oracle under Versions 1 thru 7. This algorithm has been refined during the work that led to the writing of this paper.

Once the currently effective index definitions and desired index paths are known then they can be combined and reduced using three basic rules:

No two indexes on the same table shall contain exactly the same columns unless all of the indexes will generate query plans that avoid visits to the underlying table

Each leading edge permutation shall be preserved, but no two indexes of the same type on the same table shall share the same leading edge in the same order

No index may be over unique i.e. no additional column may be concatenated with a unique key

One difficulty with the second rule is that because EXPLAIN PLAN fails to populate SEARCH_COLUMNS, the length of the effective leading edge of a concatenated index is not known, strictly speaking.

The third rule is required because the original unique index cannot be removed, as doing so would permit duplicates. The over-unique index saves just one block visit on each key access to go to the table block and perform an equality test on the referenced column.

Case Study, Part 2

Let us assume that the following “No Brainer” and “No Worries” indexes have been found on the tables in the sample query in the previous section.

```
CUSTOMERS  (CUST_ID)              primary key
           (upper(CUST_NAME))     used for enquiry
           (upper(POSTAL_CODE))   used for enquiry

ORDERS     (CUST_ID, ORDER#)      primary key and used to support FK constraint
           (CUST_ID, ORDER_TYPE)  used for enquiry

LINES      (CUST_ID, ORDER#, LINE#) primary key and used to support FK constraint
           (PRODUCT#)             used for enquiry and to support FK constraint

PRODUCTS   (PRODUCT#)             primary key
           (upper(PRODUCT_NAME))  used for enquiry
```

We do not need to make any changes to the indexes on CUSTOMERS because (CUST_ID, CUST_TYPE) is over-unique and therefore not allowed. The query can used the primary key index instead.

On the ORDERS table we have two indexes sharing the same leading edge, and inverting one of them will give us the access path that we need, so we replace (CUST_ID, ORDER_TYPE) with (ORDER_TYPE, CUST_ID).

The LINES table requires a new index, but the path proposed would share its leading edge with the primary key, so instead we add the index (LINE_TYPE, CUST_ID, ORDER#). This works equally well for the current query and may also serve some other query that we have not yet examined that requires retrieval by LINE_TYPE alone.

The PRODUCTS table does not require any index changes as a result of this analysis.

Real World Issues

Normally there will be more than one problem query to analyze, and each analysis will generate a number of proposed index paths. Note that some care has been taken to refer to them as index paths rather than indexes because in many cases the desired index look-up can be accommodated within the index strategy without having to provide an index dedicated to that specific path. The DBA or application tuner may also wish to give some key orders priority over others based on their knowledge of the application or of the data.

When different queries use differing numbers of columns of the leading edge of an index key there is a specific inefficiency that may affect the queries that use a shorter key. To take an example from the case study, consider the case where a query retrieves from the index on (LINE_TYPE, CUST_ID, ORDER#) looking only for a specific LINE_TYPE. Not only do the additional indexed columns mean that the resulting index range scans must visit more data blocks, but also multiple index entries for the same key are always stored in ascending ROWID order which can result in a slight improvement in the cache hit ratio. The rule that no two indexes may share the same leading key may need to be relaxed in some circumstances.

Conclusion

Indexes are required when running an application against Oracle to prevent the introduction of duplicate data, to avoid locking conflicts caused by foreign key constraints, and to speed up query execution. Only the first of these three categories indicates an absolute requirement for an index; in every other case the requirement will depend on the combination of the data volume and key distribution present in the database and the usage being made of the various application components.

Every index involves at least some overhead in its creation, on-going maintenance, back-up and storage and an index that is beneficial to one specific application function may have a negative effect on some other component of the service. In particular, the over-provision of indexes can have a major impact on DML performance.

By interrogating the data dictionary and building a history of SQL statement execution it is possible to determine which indexes should be retained and which should be discarded, and also to identify the SQL queries that are not currently adequately served by any index. A methodology has been presented to derive index paths to support the non-performant queries, and to derive a new index strategy from these proposed index paths and the existing index set.

This paper is based on work currently in progress, and the author looks forward to providing updated findings at the conference.