Query Optimization in Oracle Database10g Release 2

An Oracle White Paper
June 2005

ORACLE

Query Optimization in Oracle Database 10g Release 2

Query Optimization in Oracle Database10g Release 2

EXECUTIVE OVERVIEW

This paper describes Oracle's query optimizer, a key database component that enables Oracle's customers to achieve superior performance. Oracle's query optimizer technology is unmatched in the breadth of its functionality, and this paper provides a detailed discussion of all major areas of query optimization.

INTRODUCTION

What is a query optimizer?

Query optimization is of great importance for the performance of a relational database, especially for the execution of complex SQL statements. A query optimizer determines the best strategy for performing each query. The query optimizer chooses, for example, whether or not to use indexes for a given query, and which join techniques to use when joining multiple tables. These decisions have a tremendous effect on SQL performance, and query optimization is a key technology for every application, from operational systems to data warehouse and analysis systems to content-management systems.

The query optimizer is entirely transparent to the application and the end-user. Because applications may generate very complex SQL, query optimizers must be extremely sophisticated and robust to ensure good performance. For example, query optimizers transform SQL statements, so that these complex statements can be transformed into equivalent, but better performing, SQL statements.

Query optimizers are typically 'cost-based'. In a cost-based optimization strategy, multiple execution plans are generated for a given query, and then an estimated cost is computed for each plan. The query optimizer chooses the plan with the lowest estimated cost.

What does Oracle provide for query optimization?

Oracle's optimizer is perhaps the most proven optimizer in the industry. Introduced in 1992 with Oracle7, the cost-based optimizer has been continually enhanced and improved through almost a decade's worth of real-world customer experiences. A good query optimizer is not developed in a laboratory based on purely theoretical conjectures and assumptions; instead, it is developed and honed by adapting to actual customer requirements. Oracle's query optimizer has

been used in more database applications than any other query optimizer, and Oracle's optimizer has continually benefited from real-world input.

Oracle's optimizer consists of four major components (each of which is discussed in more details in subsequent sections of this paper):

SQL transformations: Oracle transforms SQL statements using a variety of sophisticated techniques during query optimization. The purpose of this phase of query optimization is to transform the original SQL statement into a semantically equivalent SQL statement that can be processed more efficiently.

Execution plan selection: For each SQL statements, the optimizer chooses an execution plan (which can be viewed using Oracle's EXPLAIN PLAN facility or via Oracle's "v$sql_plan" views). The execution plan describes all of the steps when the SQL is processed, such as the order in which tables are accessed, how the tables are joined together and whether tables are accessed via indexes. The optimizer considers many possible execution plans for each SQL statement, and chooses the best one.

Cost model and statistics: Oracle's optimizer relies upon cost estimates for the individual operations that make up the execution of a SQL statement. In order for the optimizer to choose the best execution plans, the optimizer needs the best possible cost estimates. The cost estimates are based upon in-depth knowledge about the I/O, CPU, and memory resources required by each query operation, statistical information about the database objects (tables, indexes, and materialized views), and performance information regarding the hardware server platform. The process for gathering these statistics and performance information needs to be both highly efficient and highly automated.

Dynamic runtime optimization: Not every aspect of SQL execution can be optimally planned ahead of time. Oracle thus makes dynamic adjustments to its query-processing strategies based on the current database workload. The goal of dynamic optimizations is to achieve optimal performance even when each query may not be able to obtain the ideal amount of CPU or memory resources.

Oracle additionally has a legacy optimizer, the rule-based optimizer (RBO). This optimizer exists in Oracle Database 10g Release 2 solely for backwards compatibility. Beginning with Oracle Database 10g Release 1, the RBO is no longer supported. The vast majority of Oracle's customers today use the cost-based optimizer. All major applications vendors (Oracle Applications, SAP, and Peoplesoft, to name a few) and the vast majority of recently built custom applications utilize the cost-based optimizer for enhanced performance, and this paper describes only the cost-based optimizer.

SQL TRANSFORMATIONS

There are many possible ways to express a complex query using SQL. The style of SQL submitted to the database is typically that which is simplest for the end-user to write or for the application to generate. However, these hand-written or machine-generated formulations of queries are not necessarily the most efficient SQL for executing the queries. For example, queries generated by applications often have conditions that are extraneous and can be removed. Or, there may be additional conditions that can be inferred from a query and should be added to the SQL statement. The purpose of SQL transformations is to transform a given SQL statement into a semantically-equivalent SQL statement (that is, a SQL statement which returns the same results) which can provide better performance.

All of these transformations are entirely transparent to the application and end-users; SQL transformations occur automatically during query optimization.

Oracle has implemented a wide range of SQL transformations. These broadly fall into two categories:

heuristic query transformations: These transformations are applied to incoming SQL statements whenever possible. These transformations always provide equivalent or better query performance, so that Oracle knows that applying these transformations will not degrade performance.

cost-based query transformations: Oracle uses a cost-based approach for several classes of query transformations. Using this approach, the transformed query is compared to the original query, and Oracle's optimizer then selects the best execution strategy.

The following sections discuss several examples of Oracle's transformation technologies. This is by no means a definitive list, but instead is intended to provide the reader with an understanding of the key transformation technologies and their benefits.

Heuristic query transformations

Simple view merging

Perhaps the simplest form of query transformation is view merging. For queries containing views, the reference to the view can often be removed entirely from the query by 'merging' the view definition with the query. For example, consider a very simple view and query:

```
CREATE VIEW TEST_VIEW AS
SELECT ENAME, DNAME, SAL FROM EMP E, DEPT D
WHERE E.DEPTNO = D.DEPTNO;

SELECT ENAME, DNAME FROM TEST_VIEW WHERE SAL > 10000;
```

Without any query transformations, the only way to process this query is to join all of the rows of EMP to all of the rows of the DEPT table, and then filter the rows with the appropriate values for SAL.

With view merging, the above query can be transformed into:

```
SELECT ENAME, DNAME FROM EMP E, DEPT D
WHERE E.DEPTNO = D.DEPTNO
AND E.SAL > 10000;
```

When processing the transformed query, the predicate 'SAL>10000' can be applied before the join of the EMP and the DEPT tables. This transformation can vastly improve query performance by reducing the amount of data to be joined. Even in this very simple example, the benefits and importance of query transformations is apparent.

Complex view merging

Many view-merging operations are very straightforward, such as the previous example. However, more complex views, such as views containing GROUP BY or DISTINCT operators, cannot be as easily merged. Oracle provides several sophisticated techniques for merging even complex views.

Consider a view with a GROUP BY clause. In this example, the view computes the average salary for each department:

```
CREATE VIEW AVG_SAL_VIEW AS
SELECT DEPTNO, AVG(SAL) AVG_SAL_DEPT FROM EMP
GROUP BY DEPTNO
```

A query to find the average salary for each department in Oakland:

```
SELECT DEPT.NAME, AVG_SAL_DEPT
FROM DEPT, AVG_SAL_VIEW
WHERE DEPT.DEPTNO = AVG_SAL_VIEW.DEPTNO
AND DEPT.LOC = 'OAKLAND'
```

can be tranformed into:

```
SELECT DEPT.NAME, AVG(SAL)
FROM DEPT, EMP
WHERE DEPT.DEPTNO = EMP.DEPTNO
AND DEPT.LOC = 'OAKLAND'
GROUP BY DEPT.ROWID, DEPT.NAME
```

The performance benefits of this particular transformation are immediately apparent: instead of having to group all of the data in the EMP table before doing the join, this transformation allows for the EMP data to be joined and filtered before being grouped.

Subquery "flattening"

Oracle has a variety of transformations that convert various types of subqueries into joins, semi-joins, or anti-joins. As an example of the techniques in this area, consider the following query, which selects those departments that have employees that make more than 10000:

```
SELECT D.DNAME FROM DEPT D WHERE D.DEPTNO IN
(SELECT E.DEPTNO FROM EMP E WHERE E.SAL > 10000)
```

There are a variety of possible execution plans that could be optimal for this query. Oracle will consider the different possible transformations, and select the best plan based on cost.

Without any transformations, the execution plan for this query would be similar to:

```
OPERATION           OBJECT_NAME       OPTIONS
SELECT STATEMENT
 FILTER
  TABLE ACCESS      DEPT                FULL
  TABLE ACCESS      EMP         FULL
```

With this execution plan, the all of the EMP records satisfying the subquery's conditions will be scanned for every single row in the DEPT table. In general, this is not an efficient execution strategy. However, query transformations can enable much more efficient plans.

One possible plan for this query is to execute the query as a 'semi-join'. A 'semi-join' is a special type of join which eliminates duplicate values from the inner table of the join (which is the proper semantics for this subquery). In this example, the optimizer has chosen a hash semi-join, although Oracle also supports sort-merge and nested-loop semi-joins:

```
OPERATION           OBJECT_NAME       OPTIONS
SELECT STATEMENT
 HASH JOIN                             SEMI
  TABLE ACCESS      DEPT                FULL
  TABLE ACCESS      EMP         FULL
```

Since SQL does not have a direct syntax for semi-joins, this transformed query cannot be expressed using standard SQL. However, the transformed pseudo-SQL would be:

```
SELECT DNAME FROM EMP E, DEPT D
WHERE D.DEPTNO <SEMIJOIN> E.DEPTNO
AND E.SAL > 10000;
```

Another possible plan is that the optimizer could determine that the DEPT table should be the inner table of the join. In that case, it will execute the query as a regular join, but perform a unique sort of the EMP table in order to eliminate duplicate department numbers:

```
OPERATION           OBJECT_NAME       OPTIONS
SELECT STATEMENT
 HASH JOIN
  SORT                                 UNIQUE
   TABLE ACCESS     EMP                 FULL
  TABLE ACCESS      DEPT                FULL
```

The transformed SQL for this statement would be:

```
SELECT D.DNAME FROM (SELECT DISTINCT DEPTNO FROM EMP) E, DEPT D
WHERE E.DEPTNO = D.DEPTNO
AND E.SAL > 10000;
```

Subquery flattening, like view merging, is a fundamental optimization for good query performance.

Transitive predicate generation

In some queries, a predicate on one table can be translated into a predicate on another table due to the tables' join relationship. Oracle will deduce new predicates in this way; such predicates are called transitive predicates. For example, consider a query that seeks to find all of the line-items that were shipped on the same day as the order data:

```
SELECT COUNT(DISTINCT O_ORDERKEY) FROM ORDER, LINEITEM
WHERE O_ORDERKEY = L_ORDERKEY
AND O_ORDERDATE = L_SHIPDATE
AND O_ORDERDATE BETWEEN '1-JAN-2002' AND '31-JAN-2002'
```

Using transitivity, the predicate on the ORDER table can also be applied to the LINEITEM table:

```
SELECT COUNT(DISTINCT O_ORDERKEY) FROM ORDER, LINEITEM
WHERE O_ORDERKEY = L_ORDERKEY
AND O_ORDERDATE = L_SHIPDATE
AND O_ORDERDATE BETWEEN '1-JAN-2002' AND '31-JAN-2002'
AND L_SHIPDATE BETWEEN '1-JAN-2002' AND '31-JAN-2002'
```

The existence of new predicates may reduce the amount of data to be joined, or enable the use of additional indexes.

Common subexpression elimination

When the same subexpression or calculation is used multiple times in a query, Oracle will only evaluate the expression a single time for each row.

Consider a query to find all employees in Dallas that are either Vice Presidents or with a salary greater than 100000.

```
SELECT * FROM EMP, DEPT
WHERE
(EMP.DEPTNO = DEPT.DEPTNO AND LOC = 'DALLAS' AND SAL > 100000)
OR
(EMP.DEPTNO = DEPT.DEPTNO AND LOC = 'DALLAS' AND JOB_TITLE = 'VICE
PRESIDENT')
```

The optimizer recognizes that the query can be evaluated more efficiently when transformed into:

```
SELECT * FROM EMP, DEPT
WHERE EMP.DEPTNO = DEPT.DEPTNO AND
LOC = ‘DALLAS’ AND
(SAL > 100000 OR JOB_TITLE = 'VICE PRESIDENT');
```

With this transformed query, the join predicate and the predicate on LOC only need to be evaluated once for each row of DEPT, instead of twice for each row. .

Predicate pushdown and pullup

A complex query may contain multiple views and subqueries, with many predicates that are applied to these views and subqueries. Oracle can move predicates into and out of views in order to generate new, better performing queries.

A single-table view can be used to illustrate predicate push-down:

```
CREATE VIEW EMP_AGG AS
  SELECT
    DEPTNO,
    AVG(SAL) AVG_SAL,
  FROM EMP
  GROUP BY DEPTNO;
```

Now suppose the following query is executed:

```
SELECT DEPTNO, AVG_SAL FROM EMP_AGG WHERE DEPTNO = 10;
```

Oracle will 'push' the predicate DEPTNO=10 into the view, and transform the query into the following SQL:

```
SELECT DEPTNO, AVG(SAL)5 FROM EMP WHERE DEPTNO = 10
GROUP BY DEPTNO;
```

The advantage of this transformed query is that the DEPTNO=10 predicate is applied before the GROUP-BY operation, and this could vastly reduce the amount of data to be aggregated.

Oracle has many other sophisticated techniques for pushing WHERE-clause conditions into a query block from outside, pulling conditions out of a query block, and moving conditions sideways between query blocks that are joined. Anytime a WHERE-clause condition can be propagated, it may open opportunities to filter out rows and reduce the size of data set to be processed at an earlier stage. Hence, subsequent operations, like joins or GROUP-BYs, can be applied to much smaller data sets and be performed more efficiently. Additionally, predicate pushdown and pullup may also improve performance by enabling new access paths that may not have been possible without the addition of new predicates.

Group pruning for "CUBE" queries

The SQL CUBE expression is an extension to the SQL group-by operator, which allows multiple aggregations to be retrieved in a single SQL statement. For queries containing views with CUBE expressions, it is sometimes possible to reduce the amount of data which is needed to evaluate the query. For example, consider the following query:

```
SELECT MONTH, REGION, DEPARTMENT FROM
(SELECT MONTH, REGION, DEPARTMENT,
  SUM(SALES_AMOUNT) AS REVENUE FROM SALES
  GROUP BY CUBE  (MONTH, REGION, DEPT))
WHERE MONTH = 'JAN-2001';
```

This query can be transformed into the following SQL:

```
SELECT MONTH, REGION, DEPARTMENT FROM
(SELECT MONTH, REGION, DEPARTMENT,
  SUM(SALES_AMOUNT) AS REVENUE FROM SALES
  WHERE MONTH = 'JAN-2001'
  GROUP BY MONTH, CUBE(REGION, DEPT))
WHERE MONTH = 'JAN-2001';
```

This transformed SQL involves much less aggregation, since the amount of data to be aggregated is vastly reduced (only the January, 2001 data needs to be aggregated) and the number of aggregates is additionally reduced. This is an important transformation for SQL-generators for analytic applications, since

these tools may want to query logical ‘cubes’ which have been pre-defined with views containing the CUBE operator.

Outer-join to inner join conversion

In some circumstances, it is possible to determine that an outer join in a query will return the same result as an inner join. In those cases, the optimizer will convert that outer join to an inner join. This transformation may enable Oracle to do further view merging or choose new join orders, which may not be possible if the query is an outer join.

Cost-based query transformations

Materialized view rewrite

Precomputing and storing commonly-used data in the form of a materialized view can greatly speed up query processing. Oracle can transform SQL queries so that one or more tables referenced in a query can be replaced by a reference to a materialized view. If the materialized view is smaller than the original table or tables, or has better available access paths, the transformed SQL statement could be executed much faster than the original one.

For example, consider the following materialized view:

```
CREATE MATERIALIZED VIEW SALES_SUMMARY
AS SELECT SALES.CUST_ID, TIME.MONTH, SUM(SALES_AMOUNT) AMT
FROM SALES, TIME
WHERE SALES.TIME_ID = TIME.TIME_ID
GROUP BY SALES.CUST_ID, TIME.MONTH;
```

This materialized view can be used to optimize the following query:

```
SELECT CUSTOMER.CUST_NAME, TIME.MONTH, SUM(SALES.SALES_AMOUNT)
FROM SALES, CUSTOMER, TIME
WHERE SALES.CUST_ID = CUST.CUST_ID
AND SALES.TIME_ID = TIME.TIME_ID
GROUP BY CUSTOMER.CUST_NAME, TIME.MONTH;
```

The rewritten query would be:

```
SELECT CUSTOMER.CUST_NAME, SALES_SUMMARY.MONTH, SALES_SUMMARY.AMT
FROM CUSTOMER, SALES_SUMMARY
WHERE CUSTOMER.CUST_ID = SALES_SUMMARY.CUST_ID;
```

In this example, the transformed query is likely much faster for several reasons: the sales_summary table is likely much smaller than the sales table and the transformed query requires one less join and no aggregation.

Oracle has a very robust set of rewrite techniques for materialized views, in order to allow each materialized view to be used for as broad a set of queries as possible.

Another notable trait of Oracle’s materialized views is its integration with declarative dimensions within the Oracle database. Oracle allows for the creation of dimension metadata objects, which describe the hierarchies within each dimension. This hierarchical metadata is used to support more sophisticated materialized-view query rewrite. For example, a materialized view containing monthly sales data could be used to support a query requesting quarterly sales

data if there is a time dimension which describes the hierarchical relationship between months and quarters.

Note that it is not always the case that a transformed query, which uses a materialized view, is more efficient than the original version of the query. Even if the materialized view is smaller than the table or tables upon which it is based, the base tables could be indexed more extensively and, hence, provide for faster access. The only way to choose the optimal execution plan is to calculate the best execution plans with and without the materialized views and compare their costs. Oracle does just that, so materialized view rewrite is an example of a cost-based query transformations. (For more information on materialized views, see the white paper “Oracle10g Materialized Views”).

OR-expansion

This technique converts a query with ORs in the WHERE-clause into a UNION ALL of several queries without ORs. It can be highly beneficial when the ORs refer to restrictions of different tables. Consider the following query to find all the shipments that went either from or to Oakland.

```
SELECT * FROM SHIPMENT, PORT P1, PORT P2
WHERE SHIPMENT.SOURCE_PORT_ID = P1.PORT_ID
AND SHIPMENT.DESTINATION_PORT_ID = P2.PORT_ID
AND (P1.PORT_NAME = 'OAKLAND' OR P2.PORT_NAME = 'OAKLAND')
```

The query can be transformed into:

```
SELECT * FROM SHIPMENT, PORT P1, PORT P2
WHERE SHIPMENT.SOURCE_PORT_ID = P1.PORT_ID
AND SHIPMENT.DESTINATION_PORT_ID = P2.PORT_ID
AND P1.PORT_NAME = 'OAKLAND'
UNION ALL
SELECT * FROM SHIPMENT, PORT P1, PORT P2
WHERE SHIPMENT.SOURCE_PORT_ID = P1.PORT_ID
AND SHIPMENT.DESTINATION_PORT_ID = P2.PORT_ID
AND P2.PORT_NAME = 'OAKLAND' AND P1.PORT_NAME <> 'OAKLAND'
```

Note that each UNION ALL branch can have different optimal join orders. In the first branch, Oracle could take advantage of the restriction on P1 and drive the join from that table. In the second branch, Oracle could drive from P2 instead. The resulting plan can be orders of magnitude faster than for the original version of the query, depending upon the indexes and data for these tables. This query transformation is by necessity cost-based because this transformation does not improve the performance for every query.

Star transformation

A star schema is a one modeling strategy commonly used for data marts and data warehouses. A star schema typically contains one or more very large tables, called fact tables, which store transactional data, and a larger number of smaller lookup tables, called dimension tables, which store descriptive data.

Oracle supports a technique for evaluating queries against star schemas known as the “star transformation”. This technique improves the performance of star queries by applying a transformation that adds new subqueries to the original

SQL. These new subqueries will allow the fact tables to be accessed much more efficiently using bitmap indexes.

The star transformation is best understood by examining an example. Consider the following query that returns the sum of the sales of beverages by state in the third quarter of 2001. The fact table is sales. Note that the time dimension is a "snowflake" dimension since it consists of two tables, DAY and QUARTER.

```
SELECT STORE.STATE, SUM(SALES.AMOUNT)
FROM SALES, DAY, QUARTER, PRODUCT, STORE
WHERE SALES.DAY_ID = DAY.DAY_ID AND DAY.QUARTER_ID =
QUARTER.QUARTER_ID
AND SALES.PRODUCT_ID = PRODUCT.PRODUCT_ID
AND SALES.STORE_ID = STORE.STORE_ID
AND PRODUCT.PRODUCT_CATEGORY = 'BEVERAGES'
AND QUARTER.QUARTER_NAME = '2001Q3'
GROUP BY STORE.STATE
```

The transformed query may look like

```
SELECT STORE.STATE, SUM(SALES.AMOUNT) FROM SALES, STORE
WHERE SALES.STORE_ID = STORE.STORE_ID
AND SALES.DAY_ID IN
(SELECT  DAY.DAY_ID FROM DAY, QUARTER
WHERE DAY.QUARTER_ID = QUARTER.QUARTER_ID
AND QUARTER.QUARTER_NAME = '2001Q3')
AND SALES.PRODUCT_ID IN
(SELECT  PRODUCT.PRODUCT_ID FROM PRODUCT
WHERE PRODUCT.PRODUCT_CATEGORY = 'BEVERAGES')
GROUP BY STORE.STATE
```

With the transformed SQL, this query is effectively processed in two main phases. In the first phase, all of the necessary rows are retrieved from the fact table using the bitmap indexes. In this case, the fact table will be accessed using bitmap indexes on day_id and product_id, since those are the two columns which appear in the subquery predicates.

In the second phase of the query (the 'join-back' phase), the dimension tables are joined back to the data set from the first phase. Since, in this query, the only dimension-table column which appears in the select-list is store.state, the store table is the only table which needs to be joined. The existence of the subqueries containing PRODUCT, DAY, and QUARTER in the first phase of the queries obviated the need to join those tables in second phase, and the query optimizer intelligently eliminates those joins.

The star transformation is a cost-based query transformation and both the decision whether the use of a subquery for a particular dimension is cost effective and whether the rewritten query is better than the original are done based on the optimizer's cost estimates.

This star-query execution technique is unique technology patented by Oracle. While other vendors have similar query-transformation capabilities for star queries, no other vendor combines this with static bitmap indexes and intelligent join-back elimination.

Predicate pushdown for outer-joined views

Typically, when a query contains a view that is being joined to other tables, the views can be merged in order to better optimize the query. However, if a view is being joined using an outer join, then the view cannot be merged. In this case, Oracle has specific predicate pushdown operations which will allow the join predicate to pushed into the view; this transformation allows for the possibility of executing the outer join using an index on one of the tables within the view. This transformation is cost-based because the index access may not be the most effective

ACCESS PATH SELECTION

The object of access path selection is to decide on the order in which to join the tables in a query, what join methods to use, and how to access the data in each table. All of this information for a given query can be viewed using Oracle's EXPLAIN PLAN facility or using Oracle's v$sql_plan view.

Oracle's access path selection algorithms are particularly sophisticated because Oracle provides a particularly rich set of database structures and query evaluation techniques. Oracle's access path selection and cost model incorporate a complete understanding of each of these features, so that each feature can be leveraged in the optimal way.

Oracle's database structures include:
Table Structures
Tables (default)
Index-organized tables
Nested tables
Clusters
Hash Clusters

Index Structures
B-tree Indexes
Bitmap Indexes
Bitmap Join Indexes
Reverse-key B-tree indexes
Function-based B-tree indexes
Function-based bitmap indexes
Domain indexes

Partitioning Techniques
Range Partitioning
Hash Partitioning
Composite Range-Hash Partitioning
List Partitioning
Composite Range-List Partitioning

Oracle's access techniques include:

Index Access Techniques

Index unique key look-up
Index max/min look-up
Index range scan
Index descending range scan
Index full scan
Index fast full scan
Index skip scan
Index and-equal processing
Index joins
Index B-tree to bitmap conversion
Bitmap index AND/OR processing
Bitmap index range processing
Bitmap index MINUS (NOT) processing
Bitmap index COUNT processing

Join Methods

Nested-loop inner-joins, outer-joins, semi-joins, and anti-joins
Sort-merge inner-joins, outer-joins, semi-joins, and anti-joins
Hash inner-joins, outer-joins, semi-joins, and anti-joins
Partition-wise joins

While this paper will not discuss all of the processing techniques, several of the key attributes of Oracle's access path selection are discussed below.

Join ordering

When joining a large number of tables, the space of all possible execution plans can be extremely large and it would be prohibitively time consuming for the optimizer to explore this space exhaustively. For example, a query with 5 tables has 5! = 120 possible join orders, and each join order has dozens of possible execution plans based on various combinations of indexes, access methods and join techniques. With a 5-table query, the total number of execution plans is in the thousands and thus the optimizer can consider most possible execution plans. However, with a 10-table join, there are over 3 million join orders and, typically, well over 100 million possible execution plans. Therefore, it is necessary for the optimizer to use intelligence in its exploration of the possible execution plans rather than a brute-force algorithm.

Adaptive search strategy

Oracle's optimizer uses many techniques to intelligently prune the search space. One notable technique is that Oracle uses an adaptive search strategy. If a query can be executed in one second, it would be considered excessive to spend 10

seconds for query optimization. On the other hand, if the query is likely to run for minutes or hours, it may well be worthwhile spending several seconds or even minutes in the optimization phase in the hope of finding a better plan. Oracle utilizes an adaptive optimization algorithm to ensure that the optimization time for a query is always a small percentage of the expected execution time of the query, while devoting extra optimization time for complex queries.

Some database systems allow the DBA to specify an 'optimization level' which controls the amount of time spent on query optimization. However, this adaptive search strategy is a more effective technique than a system-level parameter which control optimization. With a system-level parameter, the DBA determines the optimization for all queries uniformly, while Oracle's adaptive search strategy determines the best level of optimization for each individual query.

Multiple initial orderings heuristic

Another important technique of the Oracle optimizer's search algorithm is an innovative "multiple initial orderings heuristic". If the optimizer finds the optimal plan early in the search process, then the optimizer will finish earlier. So, this heuristic uses sophisticated methods for instantaneously finding particular plans in the search space which are likely to be nearly optimal or, at least, very good execution plans. The optimizer starts its search with these plans, rather than with randomly generated plans. This heuristic is crucial for efficient query optimization, since it vastly decreases the amount of time required for query optimization.

Bitmap indexes

Oracle innovative and patented bitmap indexes are widely used, particularly in data warehouse applications. While other database vendors provide 'dynamic' bitmap indexes, Oracle supports real bitmap indexes (in addition to dynamic bitmap indexes). Real bitmap indexes are index structures in which the compressed bitmap representation of the index is stored in the database, while dynamic bitmap indexes convert b-tree index structures in the database into bitmap structures during query processing. Real bitmap indexes provide very significant advantages, since they can provide large space savings compared to regular B-tree indexes. These space savings also translate to performance benefits in the form of fewer disk I/Os. Real bitmap indexes can process many queries 10 times faster with 10 times less index storage space. For more information on quantifying the benefits of bitmap indexes, see the performance white paper "Key Data Warehousing Features in Oracle10g: A Comparative Analysis" .

Bitmap indexes are extremely efficient for evaluating multiple predicates which are combined with AND and OR operations. In addition, bitmap indexes use Oracle's standard consistency model so that complete data consistency is maintained when performing DML operations (inserts, updates, deletes) concurrently with queries on tables with bitmap indexes.

The richness of Oracle's bitmap index capabilities offers many new execution strategies for the query optimizer. Oracle's query optimizer can generate execution plans that contain complex trees of bitmap operations that combine indexes corresponding to AND, OR, and NOT conditions in the WHERE-clause on both real bitmap indexes and dynamic bitmap indexes. These boolean operations on bitmaps are very fast, and queries that can take extensive advantage of bitmap operations usually perform very well.

In addition, Oracle supports 'index-only' accesses for bitmap index queries. An index-only access uses an exact algorithm for ANDing bitmaps. This exactness is in contrast to those database systems that use dynamic bitmap indexing, and rely upon hashing to intersect rowid lists (see for instance http://as400bks.rochester.ibm.com/cgi-bin/bookmgr/BOOKS/EZ30XB00/2.4.1). Such a system cannot avoid a table access, even when using dynamic bitmap indexes, in order to guarantee correct results. In contrast, Oracle can evaluate a wide variety of queries without accessing the table. For example, suppose a bank wants to know how many married customers it has in California.

```
SELECT COUNT(*) FROM CUSTOMER
WHERE STATE = 'CA' AND MARITAL_STATUS = 'MARRIED'
```

Assuming there are bitmap indexes on state and marital_status, Oracle can simply perform a bitwise AND of the bitmaps for 'CA' and 'married' and count the number of 1s in the resulting bitmap, an operation that is extremely fast and efficient since the entire query can be executed simply be accessing two highly-compressed index structures. A database system that cannot compute the result of this query merely from the indexes may be forced to access millions of rows in the table resulting in much slower performance.

The exactness of Oracle's real bitmap indexes (compared to dynamic bitmap indexes) is also crucial to the join-back elimination feature of Oracle's star transformation and bitmap join indexes. Database systems that lack exact bitmap operations always have to join every dimension table in the 'join-back' phase, whereas Oracle will only need to join the minimal set of dimension tables.

Oracle's bitmap indexes also have key advantages over other types of compressed index structures. For example, when accessing a range of values using Oracle's bitmap indexes, the query optimizer will generate start keys and stop keys so that only a portion of the index structure is accessed. This starkly contrasts with indexes such as "encoded vector indexes" where each index always has to be scanned in full to execute a query -- a horrendously wasteful practice.

Oracle's query optimizer has the ability to combine multiple types of indexes to access the same table. For example, a bitmap index can be combined with a domain index and a B-tree index to access a given table. The domain index and B-tree index are accessed using dynamic bitmap indexing techniques in order to be combined with the bitmap index.

Bitmap join indexes

A 'join index' is an index structure which spans multiple tables, and improves the performance of joins of those tables.

A join index is an index where the indexed column(s) and the rowid/bitmap representation in the index refer to different tables. Part of the definition of a bitmap join index therefore also includes join conditions specifying how the rows of the tables match. Typically, one would create a join index on a fact table, where the indexed column would belong to a dimension table. For example, given a fact table, sales, and a dimension table, product, a bitmap join index could be created on sales, but where the indexed column is product category in the product table subject to a join condition on product_id. With such an index, it is possible to find all the sales rows for products in a given product category directly from the index without performing the join. Bitmap join indexes have two major advantages:

1. Cardinality reduction: There are likely far fewer distinct product categories than product ids. Hence, the bitmap join index would have a much lower cardinality than any index on the join column of the fact table, something that, for bitmap indexes, translates to smaller size.

2. Join-back elimination. In many cases, it is possible to eliminate joins from the query when bitmap join indexes are used.

Bitmap join indexes can be used together with non-join indexes in the same access path using the bitmap technology for combining multiple indexes. Typically, those other indexes would be used due to the star transformation, which works in conjunction with bitmap join indexes.

Domain indexes and extensibility

Oracle supports application domain indexes to provide efficient access to customized complex data types such as documents, spatial data, images, and video clips. Such indexes are different than the built-in Oracle indexes, but their properties can be registered with Oracle. Oracle's optimizer is extensible so that domain indexes and user-defined functions can have associated statistics and cost functions. They are considered in the same cost model and search space as Oracle's built-in indexes and functions and can even be combined with regular Oracle indexes in the same access path using Oracle's bitmap technology.

Fast full index scans

A fast full scan of an index is the ability to scan through the index as a table, not in a tree-based order like an index. This can be used if all the needed table columns are contained in the index so that no table access is necessary. Fast full scans are useful when a large amount of data needs to be retrieved since they can take maximum advantage of multiblock disk-I/Os and also parallelize better than range scans. Since the index is likely to be much smaller than the table, it is usually much cheaper to scan the index than the table itself.

Index joins

Index joins allow a subset of the columns from the table to be reconstructed from multiple indexes in case there is not a single index that contains all the needed columns. If there is a set of indexes that together contains all the needed table columns, the optimizer can use those indexes to avoid a potentially expensive table access, either by rowid or as a full scan. First, the WHERE-clause conditions, if any, are applied to the indexes and the set of index entries are returned. The resulting entries from the different indexes are then joined on their rowid values. This access method is useful when the underlying table has many columns, but only a small number of columns are needed for the query.

Index skip scans

Index skip scan is a feature that allows the optimizer to utilize a multicolumn index even if there is no start/stop key on the leading column. If the trailing columns of an index have selective predicates, but not the leading column(s), the optimizer might still be able to take advantage of the index provided that the number of values of the leading column(s) is relatively limited. For each value for the leading column(s), the index is probed using the selective trailing columns to reach the relevant leaf block efficiently. Hence, the index can be traversed based on a limited number of skips, where each skip probe is very efficient. This technique is very useful for ameliorating situations where no index is a perfect match for the query and the only alternative would be to perform a full table scan or a full index scan.

Partition optimizations

Partitioning is an important feature for manageability. Oracle supports the partitioning on both tables and indexes. While Oracle supports a variety of partitioning methods, range partitioning (or composite partitioning range/hash or range/list) is perhaps the most useful partitioning method, particularly in data warehousing where “rolling windows” of data are common. Range partitioning on date ranges can have enormous benefits for the load/drop cycle in a data warehouse, both in terms of efficiency and manageability.

However, partitioning is useful for query processing as well and Oracle’s optimizer is fully partition aware. Again, date range partitioning is typically by

far the most important technique since decision support queries usually contain conditions that constrain to a specific time period. Consider the case where two years' worth of sales data is stored partitioned by month. Assume that we want to calculate the sum of the sales in the last three months. Oracle's optimizer will know that the most efficient way to access the data is by scanning the partitions for the last three months. That way, exactly the relevant data is scanned. A system lacking range partitioning would either have to scan the entire table -- 8 times more data -- or use an index, which is an extremely inefficient access method when the amount of data that needs to be accessed is large. The ability of the optimizer to avoid scanning irrelevant partitions is known as *partition pruning*.

Partition-wise joins, GROUP-BY's and sorts

Certain operations can be conducted on a 'partition-wise' basis when those operations involve the partitioning key of a table. For example, suppose that a sales table was range-partitioned by date. When a query requests sales records ordered by data, Oracle's optimizer realizes that each partition could be sorted independently (on a partition-wise basis) and then the results could simply be concatenated afterwards. Sorting each partitioning separately is much more efficient than sorting the entire table at one time. Similar optimizations are made for join and GROUP-BY operations.

Sort elimination

Sorting a large amount of data is one of the most resource intensive operations in query processing. Oracle eliminates unneeded sort operations. There are multiple reasons a sort (for DISTINCT, GROUP BY, ORDER BY) can be eliminated. For instance, the use of an index may guarantee that the rows will already have the right order or be unique; a unique constraint may guarantee that only a single row will be returned; it may be known that all the rows will have the same value for an ORDER BY column; an earlier sort operation, say for a sort-merge join may already have generated the right order. Oracle's optimizer knows that a local decision about, say, what index to use for a table can have more far-reaching side-effect of determining whether an ORDER BY sort can be eliminated once that table has been joined to all the other tables in the query. That determination will also be influenced by the join order and join methods used, factors that may not be known when the index on the table was picked. Therefore, in addition to creating an execution plan based on the locally optimal choices, the best index, cheapest join method, etc., the optimizer will try to generate an execution plan that is explicitly geared towards global sort elimination choosing the right indexes, join methods, and join order. When the overall cost is computed, the plan with sort elimination often turns out to be better than the plan based on the locally least expensive operations.

OLAP optimizations

Oracle supports various SQL constructs that are commonly used in OLAP. When processing queries with CUBE, ROLLUP, or grouping set constructs, as well as SQL analytic functions, there may be a large number of sorts required since the number of different groupings might be large. However, if the data is already sorted according to one grouping, it may eliminate the need for (or ameliorate the cost of) the sort for a different grouping depending on how the grouping columns match up. Hence, being clever about how to perform the grouping sorts can result in large performance gains. Oracle's optimizer uses sophisticated algorithms for ordering the sorts and rearranging the grouping columns so that the result of one sort can speed up or eliminate next one.

Parallel execution

Parallel execution is the ability to apply multiple processes (generally involving multiple CPU's and possibly multiple nodes) to the execution of a single SQL statement. Parallelism is a fundamental capability for querying and managing large data sets.

Oracle's parallel execution architecture allows virtually any SQL statement to be executed with any degree of parallelism. Notably, the degree of parallelism is not based upon the partitioning scheme of the underlying database objects. This flexible parallel architecture provides significant. advantages, since Oracle can adjust the degree of parallelism based upon factors such as the size of tables to be queried, the workload, and the priority of the user issuing the query.

The optimizer fully incorporates parallel execution, and takes into account the impact of parallel execution when choosing the best execution plan.

Hints

Hints are directives added to a SQL query to influence the execution plan. Hints are most commonly used to address the rare cases in which the optimizer chooses a suboptimal plan, yet hints are often misunderstood or misused in marketing or sales presentations. While some would claim that the existence of hints is a sign of weakness in the optimizer, in fact most major database products (Oracle, Microsoft SQL Server, IBM Informix, Sybase) support some form of optimizer directives. No optimizer is perfect (see, for instance, "Why does DB2's optimizer generate wrong plans?" a presentation from the IBM Almaden Research Center at the International DB2 Users Group Meeting, 1999), and directives such as Oracle's hints provide the simplest workaround situations in which the optimizer has chosen a suboptimal plan.

Hints are useful tools not just to remedy an occasional suboptimal plan, but also for users who want to experiment with access paths, or simply have full control over the execution of a query. For example, a user can compare the performance of a suboptimal index in a query to the performance of the optimal index. If the optimal index is only used in this particular query, but the suboptimal one is used

by many different queries, the user can consider dropping the optimal index if the performance difference is small enough. Hints make it very simple and quick to conduct these types of performance experiments (since, without hints, a user would have to drop and then rebuild the optimal index in order to do this experiment).

Hints are designed to be used only in unusual cases to address performance issues; hints should be rarely used and in fact the over-use of hints can detrimentally affect performance since these hints can prevent the optimizer from adjusting the execution plans as circumstances change (tables grow, indexes are added, etc) and may mask problems such as stale optimizer statistics. One example of an appropriate use of hints is Oracle's own applications: the highly-tuned ERP modules of Oracle's E-Business Suite 11i utilize hints in .3% of the 270,000 SQL statements in these modules.

COST MODEL AND STATISTICS

A cost-based optimizer works by estimating the cost of various alternative execution plans and choosing the plan with the best (that is, lowest) cost estimate. Thus, the cost model is a crucial component of the optimizer, since the accuracy of the cost model directly impacts the optimizer's ability to recognize and choose the best execution plans.

The 'cost' of an execution plan is based upon careful modeling of each component of the execution plan. The cost model incorporates detailed information about all of Oracle's access methods and database structures, so that it can generate an accurate cost for each type of operation. Additionally, the cost model relies on 'optimizer statistics', which describe the objects in the database and the performance characteristics of the hardware platform. These statistics are described in more detail below. In order for the cost-model to work effectively, the cost model must have accurate statistics. Oracle has many features to help simplify and automate statistics-gathering.

The cost model is a very sophisticated component of the query optimizer. Not only does the cost-model understand each access method provided by the database, but it must also consider the performance effects of caching, I/O optimizations, parallelism, and other performance features. Moreover, there is no single definition for costs. For some applications, the goal of query optimization is to minimize the time to return the first row or first set of N rows, while for other applications, the goal is to return the entire result set in the least amount of time. Oracle's cost-model supports both of these goals by computing different types of costs based upon the DBA's preference.

Optimizer statistics

When optimizing a query, the optimizer relies on a cost model to estimate the cost of the operations involved in the execution plan (joins, index scans, table scans, etc.). This cost model relies on information about the properties of the

database objects involved in the SQL query as well as the underlying hardware platform. In Oracle, this information, the optimizer statistics, comes in two flavors: object-level statistics and system statistics.

Object-level statistics

Object-level statistics describe the objects in the database. These statistics track values such as the number of blocks and the number of rows in each table, and the number of levels in each b-tree index. There are also statistics describing the columns in each table. Column statistics are especially important because they are used to estimate the number of rows that will match the conditions in the WHERE-clauses of each query. For every column, Oracle's column statistics have the minimum and maximum values, and the number of distinct values. Additionally, Oracle supports histograms to better optimize queries on columns which contain skewed data distributions. Oracle supports both height-balanced histograms and frequency histograms, and automatically chooses the appropriate type of histogram depending on the exact properties of the column.

System statistics

System statistics describe the performance characteristics of the hardware platform. The optimizer's cost model distinguishes between the CPU costs and I/O costs. However, the speed of the CPU varies greatly between different systems and moreover the ratio between CPU and I/O performance also varies greatly. Hence, rather than relying upon a fixed formula for combining CPU and I/O costs, Oracle provides a facility for gathering information about the characteristics of an individual system during a typical workload in order to determine the best way to combine these costs for each system. The information collected includes CPU-speed and the performance of the various types of I/O (the optimizer distinguishes between single-block, multi-block, and direct-disk I/Os when gathering I/O statistics). By tailoring the system statistics for each hardware environment, Oracle's cost model can be very accurate on any configuration from any combination of hardware vendors.

User-defined statistics

Oracle also supports user-defined cost functions for user-defined functions and domain indexes. Customers who are extending Oracle's capabilities with their own functions and indexes can fully integrate their own access methods into Oracle's cost model. Oracle's cost model is modular, so that these user-defined statistics are considered within the same cost model and search space as Oracle's own built-in indexes and functions.

Statistics management

The properties of the database tend to change over time as the data changes, either due to transactional activity or due to new data being loaded into a data warehouse. In order for the object-level optimizer statistics to stay accurate, those

statistics need to be updated when the underlying data has changed. The problem of gathering the statistics poses several challenges for the DBA:

Statistics gathering can be very resource intensive for large databases.

Determining which tables need updated statistics can be difficult. Many of the tables may not have changed very much and recalculating the statistics for those would be a waste of resources. However, in a database with thousands of tables, it is difficult for the DBA to manually track the level of changes to each table and which tables require new statistics.

Determining which columns need histograms can be difficult. Some columns may need histograms, others not. Creating histograms for columns that do not need them is a waste of time and space. However, not creating histograms for columns that need them could lead to bad optimizer decisions.

Oracle's statistics-gathering routines address each of these challenges.

Automatic statistic gathering

In Oracle Database 10g Release 2 the recommended approach to gathering statistics is to allow Oracle to automatically gather the statistics. Oracle will gather statistics on all database objects automatically and maintains those statistics in a regularly-scheduled maintenance job (GATHER STATS job). This job gathers statistics on all objects in the database, which have missing, or stale statistics (more than 10% of the rows have changed since it was last analyzed). The GATHER STATS job is created automatically at database creation time and is managed by the Scheduler. The Scheduler runs this job when the maintenance window is opened. By default, the maintenance window opens every night from 10 P.M. to 6 A.M. and all day on weekends. Automated statistics collection eliminates many of the manual tasks associated with managing the query optimizer, and significantly reduces the chances of getting poor execution plans because of missing or stale statistics. The GATHER STATS job can be stopped completely using the DBMS_SCHEDULER package.

Parallel sampling

The basic feature that allows efficient statistics gathering is sampling. Rather than scanning (and sorting) an entire table to gather statistics, good statistics can often be gathered by examining a small sample of rows. The speed-up due to sampling can be dramatic since sampling not only the amount of time to scan a table, but also subsequently reduces the amount of time to process the data (since there is less data to sorted and analyzed). Further speed-up for gathering statistics on very large databases can be achieved by using sampling in conjunction with parallelism. Oracle's statistics gathering routines automatically determines the appropriate sampling percentage as well as the appropriate degree of parallelism, based upon the data-characteristics of the underlying table.

Monitoring

Another key feature for simplifying statistics management is monitoring. Oracle keeps track of how many changes (inserts, updates, and deletes) have been made to a table since the last time statistics were collected. Those tables that have changed sufficiently to merit new optimizer statistics are marked automatically by the monitoring process. When the DBA gathers statistics, Oracle will only gather statistics on those tables which have been significantly modified.

Automatic histogram determination

Oracle's statistics-gathering routines also implicitly determine which columns require histograms. Oracle makes this determination by examining two characteristics: the data-distribution of each column, and the frequency with which the column appears in the WHERE-clause of SQL statements. For columns which are both highly-skewed and commonly appear in WHERE-clauses, Oracle will create a histogram.

These features together virtually automate the process of gathering optimizer statistics. With a single command, the DBA can gather statistics on an entire schema, and Oracle will implicitly determine which tables require new statistics, which columns require histograms, and the sampling level and degree of parallelism appropriate for each table.

Dynamic sampling

Unfortunately, even accurate statistics are not always sufficient for optimal query execution. The optimizer statistics are by definition only an approximate description of the actual database objects. In some cases, these static statistics are incomplete. Oracle addresses those cases by supplementing the static object-level statistics with additional statistics that are gathered dynamically during query optimization.

There are primarily two scenarios in which the static optimizer statistics are inadequate:

Correlation. Often, queries have complex WHERE-clauses in which there are two or more conditions on a single table. Here is a very simple example:

```
SELECT * FROM EMP
WHERE JOB_TITLE = 'VICE PRESIDENT'
AND SAL < 40000
```

The naïve optimization approach is to assume that these two columns are independent. That is, if 5% of the employees are 'Vice President' and 40% of the employees have a salary less than 40,000, then the simple approach is to assume that .05 * .40 = .02 of employees match both criteria of this query. This simple approach is incorrect in this case. Job_title and salary are correlated, since employees with a job_title of 'Vice President' are much more likely to have higher salaries. Although the simple approach

indicates that this query should return 2% of the rows, this query may in actuality return zero rows.

The static optimizer statistics, which store information about each column separately, do not provide any indication to the optimizer of which columns may be correlated. Moreover, trying to store statistics to capture correlation information is a daunting task: the number of potential column combinations is exponentially large.

Transient data. Some applications will generate some intermediate result set that is temporarily stored in a table. The result set is used as a basis for further operations and then is deleted or simply rolled back. It can be very difficult to capture accurate statistics for the temporary table where the intermediate result is stored since the data only exists for a short time and might not even be committed. There is no opportunity for a DBA to gather static statistics on these transient objects.

Oracle's dynamic sampling feature addresses these problems. While a query is being optimized, the optimizer may notice that a set of columns may be correlated or that a table is missing statistics. In those cases, the optimizer will sample a small set of rows from the appropriate table(s) and gather the appropriate statistics on-the-fly. In the case of correlation, all of the relevant conditions in the WHERE-clause are applied to those rows simultaneously to directly measure the impact of correlation. This dynamic sampling technique is also very effective for transient data; since the sampling occurs in the same transaction as the query, the optimizer can see the user's transient data even if that data is uncommitted.

Optimization cost modes

A cost-based optimizer works by estimating the cost for various alternative execution plans and picking the one with the lowest cost estimate. However, the notion of "lowest cost" can vary based upon the requirements of a given application. In an operational system, which displays only a handful of rows to the end-user at a time, the “lowest cost” execution plan may be the execution plan which returns the first row in the shortest amount of time. On the other hand, in a system in which the end-user is examining the entire data set returned by a query, the “lowest cost” execution plan is the execution plan which returns all of the rows in the least amount of time.

Oracle provides two optimizer modes: one for minimizing the time to return the first N rows of query, and another for minimizing the time to return all of the rows from a query. The database administrator can additionally specify the value for N. In this way, Oracle’s query optimizer can be easily tuned to meet the specific requirements of different types of applications.

DYNAMIC RUNTIME OPTIMIZATIONS

The workload on every database fluctuates, sometimes greatly, from hour to hour, from daytime workloads to evening workloads, from weekday workloads to weekend workloads, and from normal workloads to end-of-quarter and end-of-year workloads. No set of static optimizer statistics and fixed optimizer models can cover all of the dynamic aspects of these ever-changing systems. Dynamic adjustments to the execution strategies are mandatory for achieving good performance.

For this reason, Oracle's query optimization extends beyond just access path selection. Oracle has a very robust set of capabilities which allow for adjustments to the execution strategies for each query based not only on the SQL statement and the database objects, but also the current state of the entire system at the point in time when the query is executing.

The key consideration for dynamic optimization is the appropriate management of the hardware resources, such as CPU and memory. The hallmark of dynamic optimization is the dynamic adjustments of execution strategies for each query so that the hardware resources are utilized to maximize the throughput of all queries. While most other aspects of query optimization focus on optimizing only a single SQL statement, dynamic optimization focuses on optimizing each SQL statement in the context of all of the other SQL statements that are currently executing.

Dynamic degree of parallelism

Parallelism is a great way to improve the response time of a query on a multiprocessor hardware. However, the parallel execution of the query will likely use slightly more resources in total than serial execution. Hence, on a heavily loaded system with resource contention, parallelizing a query or using too high a degree of parallelism can be counterproductive. On other hand, on a lightly loaded system, queries should have a high degree parallelism to leverage the available resources. Therefore, relying on a fixed degree of parallelism is a bad idea since the workload on the system varies over time. Oracle automatically adjusts the degree of parallelism for query, throttling it back as the workload increases in order to avoid resource contention. As the workload decreases, the degree of parallelism is again increased.

Dynamic memory allocation

Some operations (primarily, sorts and hash joins) are faster if they have access to more memory. These operations typically process the data multiple times; the more data that can fit into memory, the less data will need to be stored on disk in temporary tablespaces between each pass. In the best case, sorts and hash joins can occur entirely in memory so that temporary disk space is not used at all.

Unfortunately, there is only a finite amount of physical memory available on the system and it has to be shared by all the operations that are executing

concurrently. If memory is overallocated, then swapping will occur and performance will deteriorate dramatically. On the other hand, if there is memory available that could be used to speed up the execution of a sort or a hash join, it should be allocated or the performance of the operation will be suboptimal. The challenge is to provide the optimal amount of memory for each query: enough memory to process the query efficiently, but not too much memory so that other queries can receive their share of memory as well.

Assigning a fixed amount of memory to each SQL statement (an approach used by other database vendors) is not an effective solution. As the database workload increases, more and more memory will be required to handle the increasing number of queries. Eventually, the physical memory on the system would be exhausted, and the performance of the system would degrade dramatically.

A slightly more clever, but nevertheless inefficient, approach is to give each query an equal-sized portion of memory, so that if there are 100 concurrent queries, then each query gets 1% of the available memory and if there are 1000 concurrent queries, then each query gets .1% of the available memory. This solution is insufficient, because each query operates on different-sized data sets, and each query may have a different number of sort and hash-join operations. If each query was given a fixed amount of memory, then some queries would have far too much memory while other queries would have insufficient memory.

Oracle's dynamic memory management resolves these issues. The DBA specifies the total amount of system memory available to Oracle for SQL operations. Oracle manages the memory so that each query receives an appropriate amount of memory within the boundary that the total amount of memory for all queries does not exceed the DBA-specified limit.

For each query, the optimizer generates a profile with the memory requirements for each operation in the execution plan. These profiles not only contain the ideal memory levels (that is, the amount of memory needed to complete an operation entirely in memory) but also the memory requirements for multiple disk passes. At runtime, Oracle examines the amount of available memory on the system and the query’s profile, and allocates memory to the query to provide optimal performance in light of the current system workload.

Even while queries are running, Oracle will continue to dynamically adjust the memory for each query. If a given query is using less memory than anticipated, that memory will be re-assigned to other queries. If a given query can be accelerated with the addition of more memory, then that query will be given additional memory when it is available. The decision about how each individual operation is affected by altering the memory allocations is based upon its memory profile. When dynamically adjusting memory allocations, Oracle will pick the individual operations that are best suited for the change with respect to the impact on overall performance.

This unique feature greatly improves both the performance and manageability of the database, and relieves the DBA from managing memory allocations -- a problem that is impossible to manually resolve to perfection.

Database resource manager

Oracle's database resource manager provides a framework that allows the DBA to carefully control how resources are allocated among different users. The database resource manager controls many aspects of query execution, including the amount of CPU allocated to each group of users, the number of active sessions for each group of users, and the extent to which each group of users can use parallelism.

The database resource manager provides two main benefits:

- Maximize throughput of the entire system. The database resource manager ensures that the hardware resources are being fully utilized, while at the same time managing the workload so that the system is not over-utilized.
- Provide resources to the right users. Some users of the data warehouse should have higher priority than other users. The database resource manager allows the DBA to dedicate more resources to important users.

The database resource manager is important for query optimization for a couple of reasons. First, the resource manager provides dynamic adjustments during SQL execution. A complex query may require a large, CPU-intensive sort, for example. During runtime, the database resource manager may limit the CPU utilized by this query. In this way, the database resource manager can dynamically adjust the SQL execution so that large queries do not overwhelm the system and deteriorate the response time for other users.

The second way in which the database resource manager is closely tied to the query optimization is that the database resource manager provides a capability for pro-active query governing. For each group of end-users, the database administrator can specify the longest-running SQL statement that those users are allowed to execute. For example, the database administrator could specify that a given group of users is not allowed to submit any queries which will take longer than 20 minutes to execute. If a user tries to submit a long-running SQL statement, that SQL statement will return an error before it even begins to execute. The purpose of query governing is to pro-actively prevent users for issuing unacceptably long-running queries. Behind the scenes, the query governing is implemented in conjunction with the query optimizer. The query optimizer's cost estimates are used to estimate the execution times for each query, so that the database resource manager can determine which queries may need to be pro-actively aborted.

CONCLUSION

Query optimization is a key ingredient for achieving good performance and simplifying administration. Oracle's query optimizer provides a tremendous breadth of capabilities. Not only does it incorporate perhaps the broadest set of access paths of any database vendor, but Oracle's optimizer also provides particularly innovative techniques for dynamic and adaptive adjustments during query execution. By continually enhancing its optimizer, Oracle will continue to provides its customers with leading performance and manageability.

Query Optimization in Oracle Database 10g Release 2
June 2005
Author: George Lumpkin Hakan Jakobsson
Contributing Authors: Maria Colgan

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
Phone: +1.650.506.7000
Fax: +1.650.506.7200
oracle.com