



Your Performance IS Our Business

10713 RR 620 North, Building 400

Austin, Texas 78726-1708

(512) 249-2324 (866) 773-8789

www.DBISoftware.com

DBI Query Rewrite Component (QRC)

This paper discusses some of the background underlying SQL query rewrites and describes their application in the Brother-Owl 7.0.0 Query Rewrite Component.

Introduction

SQL is a powerful language for extracting and manipulating data in a relational database. Over the last thirty years, it has evolved from a set of mathematical concepts to an industry standard.

Much of the power of SQL is derived from its simplicity. An SQL statement describes "what" data should be retrieved or updated; it does not describe "how" this should be achieved. End users need to understand the logical relationships between structures in the database, but the physical storage structures can be completely transparent. Whilst the user must specify the results required from an SQL statement, they are not required to specify any procedural logic as this is generated by the database engine. Consequently, SQL end-users can rapidly become highly productive.

The aforementioned properties lead to SQL often being labelled a fourth generation language or 4GL. On the other hand, languages such as C, C++ or Java are described as 3GLs. In these languages, the user must specify procedural logic although much physical complexity can be abstracted into re-usable objects or components.

Statements written in SQL must be converted into machine instructions by the database engine. In the case of Oracle, SQL statements are parsed to check for syntactical and semantic errors. An Optimizer then generates an execution plan for the statement, which is effectively a set of high-level instructions that is interpreted by a run-time machine when the statement is executed.

One of the reasons that users can achieve results so rapidly is that there are often many ways to write the same statement. Although in theory it should be possible to derive an optimum execution plan for every statement, practical considerations usually limit the amount of time and resource that can be dedicated to statement parsing and Optimization. Whilst the execution plan for an SQL statement will normally generate the correct results, it will not necessarily do so optimally.

Optimization

Rule-Based Optimizer (RBO)

Early versions of Oracle parsed and optimized each statement as it was executed. The Optimizer followed a heuristic algorithm, which applied a set of rules to the statement. The decisions made by the Optimizer were mainly determined by the existence or otherwise of suitable indexes. The order in which tables were joined was determined by the order in which they appear in the FROM clause. The order in which predicates were evaluated was determined by the order of the WHERE clause. Whilst use of indexes could never be forced, inappropriate use of indexes could always be suppressed using widely publicized coding techniques. Thus, the statement could always be modified to achieve the optimum execution plan.

This original Oracle Optimizer became known as the rule-based Optimizer or RBO. It still exists in Oracle 9.2 although Oracle has announced that it will be removed in Oracle 10. Until very recently it was still being used by many packages. Today, it is still used for all statements generated recursively to access the Oracle data dictionary.

However, the rule-based Optimizer has several significant weaknesses. No allowance is made for data volume or growth patterns. Thus, the same execution plan will be generated for a very small table as for a very large table. In addition, an even distribution of data is assumed for each database object. Consequently, for tables with skewed data distributions



Your Performance IS Our Business

10713 RR 620 North, Building 400

Austin, Texas 78726-1708

(512) 249-2324 (866) 773-8789

www.DBISoftware.com

highly inefficient execution plans can be generated. While RBO access paths can be accurately specified by rewriting the SQL statement, the implication of the effects of data growth and changes in data distribution is that statements constantly require review and occasionally require revision in order to maintain optimum execution plans. Over the last decade, Oracle databases have grown in size and complexity. This has led to the introduction of a variety of new physical structures, which are designed to create new, more efficient access paths. It would be extremely difficult to modify the set of 17 rules implemented by the rule-based Optimizer to accommodate each new structure while maintaining the integrity of existing rules.

A final weakness of the RBO in Oracle 6 was that each statement was parsed and then executed. Where the same statement was being executed repeatedly, the parsing and Optimization phases were creating an unacceptable and potentially unnecessary overhead. Bind variables allowed an individual session to reuse an execution plan while the cursor remained open – there was no such capability for multiple sessions.

Cost-Based Optimizer

The weaknesses of the rule-based Optimizer were addressed by Oracle in version 7.0, which was released in 1992. This release saw the introduction of the Cost-Based Optimizer (CBO), which has continued to slowly evolve over the last ten years. The CBO contains significant advances over the RBO. These include the collection and use of object statistics, the ability to support new features and storage mechanisms and the sharing of execution plans in the library cache.

In the CBO, Optimization decisions are based on costs derived from object statistics collected by a background process (ANALYZE) and held statically in the data dictionary. Statistics, which can be collected for both tables and indexes, include the number of physical blocks and number of rows. Additional statistics derived for indexes included the depth of the index and the clustering factor, which measures the correlation between the ordering of data in the table with that in the index. For columns with skewed data distributions, it is possible to generate histograms, which allow the Optimizer to consider individual data values when creating an execution plan. Since the execution plan is derived from the object statistics, it is important that the statistics are regularly updated to reflect the current state of the database.

The Cost-Based Optimizer is aware of all the new physical storage mechanisms that have been introduced in Oracle since version 7.0. These have included bitmap indexes, partitioned tables and indexes, abstract datatypes, domain indexes, index-organized tables (IOTs), global temporary tables, function-based indexes and materialized views. It is also aware of new SQL constructs such as analytic functions. In contrast, whilst the rule-based Optimizer has been maintained through Oracle versions 7, 8 and 9, it has not been enhanced to work with new features.

Finally, Oracle 7.0 also saw the addition of the library cache in the shared memory area. Parsed statements and their execution plans were held in a first-in first-out queue structure, which enabled them to be shared between multiple sessions. This reduced the amount of time and resources that were dedicated to parsing and Optimization. The library cache is not Optimizer specific; thus, the CBO and RBO both utilize this functionality.

Although the CBO promised many benefits in theory, in practice it has taken many years to gain widespread acceptance. This has been due to a number of factors of which the most significant was that early versions frequently generated highly inefficient execution plans. It became possible for a handful of statements to consume the vast majority of the resources in the database, severely impacting overall throughput and response times. Oracle attempted to resolve this problem by introducing hints, which are effectively directives to instruct Oracle how to process a particular statement.

Until Oracle 8i, hints could only be applied by the end-user who had access to the source. Although it was possible to identify inefficient statements in the SGA, it was not necessarily possible to modify the software that was executing them. In Oracle 8i and above, it is possible to create stored outlines to save and modify the execution plans for specific statements. This effectively allows hints to be added to SQL statements at execution time. This approach is appropriate in an OLTP environment where the same statement may be executed multiple times. However, stored outlines are less effective with ad hoc queries often found in DSS environments.



Your Performance IS Our Business

10713 RR 620 North, Building 400

Austin, Texas 78726-1708

(512) 249-2324 (866) 773-8789

www.DBISoftware.com

The added complexity delivered by using statistics, hints and a number of new access paths and objects, plus commercial inertia, led many users including most package suppliers to continue using the rule-based Optimizer until the advent of Oracle 8i.

Unlike the RBO, which follows a simple algorithm to derive an execution plan, the cost-based Optimizer can consider potentially millions of execution plans. Parsing of complex statements can take anything from a few seconds to many minutes. Oracle employs a number of techniques that attempt to reduce the resources required for parsing. The application of these techniques often results in a compromise in which fewer resources are used, but a suboptimal plan is selected

Oracle Transformations

Prior to Optimizing an SQL statement, Oracle applies a series of SQL transformations intended to transform the query into a semantically equivalent SQL statement, which can provide better performance. These transformations fall into two categories

- *Heuristic transformations.* These are applied to SQL statements wherever possible and will always provide equivalent or better performance
- *Cost-based transformations.* These transforms are only applied if they result in a lower Oracle cost.

Heuristic Transformations

These are always applied to SQL statements and include

- Simple view merging
- Complex view merging
- Subquery flattening
- Transitive predicate elimination
- Common subexpression elimination
- Predicate pushdown
- Predicate pullup
- Outer to inner join conversion

In addition, Oracle applies many simple transformations to statements at parse time. For example the predicate

```
WHERE c01 = UPPER ('oracle')
```

would be transformed to the predicate

```
WHERE c01 = 'ORACLE'
```

Cost-Based Transformations

These are only applied if the transformation results in a lower Oracle cost. They include

- Materialized view rewrite
- OR-expansion
- Star transformation
- Predicate pushdown for outer joined views

The transformations automatically applied by Oracle vary with each release as functionality is enhanced.



Your Performance IS Our Business

10713 RR 620 North, Building 400

Austin, Texas 78726-1708

(512) 249-2324 (866) 773-8789

www.DBIssoftware.com

Query Rewrite Component

The remainder of this document describes the functionality of the DBI Query Rewrite Component (QRC).

The QRC generates syntactically different, but semantically identical SQL statements. The rewritten queries may allow the Oracle Optimizer to consider new, possibly more efficient execution plans. This is achieved by giving the Oracle Optimizer a new starting point. This is often sufficient for the Optimizer to generate a different set of plans, ideally with lower costs.

Functionality

The QRC takes as input an SQL statement and a database connection and produces as output zero or more rewritten SQL statements ranked in order of their Oracle costs.

The QRC is designed to apply rewrites recursively. It is fully extensible so that new rewrites can be applied in the future. Further rewrites have already been identified and will be added in future releases.

Where appropriate the QRC mirrors the Oracle SQL transformation functionality. Whilst these transforms would be automatically applied by the Oracle Optimizer, applying them within the QRC opens up more paths for additional QRC transforms and consequently increases the probability of generating a more efficient execution plan.

Within the QRC, rewrites are divided into transforms or hints. Transforms involve the manipulation of the syntactic structure of the SQL statement. Hints are simply added to the statement by the inclusion of a comment in the appropriate subquery.

Transforms can be applied recursively or non-recursively. Recursive transforms are applied many times to the same statement in different orders until no new SQL is produced. Nonrecursive transforms are only applied once and are used to restrict the number of execution plans considered.

Some transforms exist purely for maintenance reasons. As the statement is recursively transformed, new statements are generated. The maintenance transforms are designed to standardize these SQL statements so that they can be compared. This includes rationalizing brackets and standardizing columns returned by EXISTS subqueries. Duplicate statements are removed which eliminates unnecessary parsing.

Hints are only applied after all syntactic transformations. This is because the hints are only applied where they are appropriate within the context of the statement. If any further transformations were to be applied, the hint could be made redundant.

Internally an XML document is used to represent the syntactic structure of the statement in memory. Each transform manipulates the XML document. Methods exist to convert an SQL statement into an XML document and vice versa.

The current version of the QRC ranks statements on Oracle costs. However for complex queries, Oracle costs can be misleading and it may be necessary to execute each query in order to establish which execution plan is the most efficient.

Accuracy

The main criterion for any query rewrite is that the transformed statement should return the same result set as the original. The columns returned must always be in the same order, however the rows returned must only be in the same order if there is an explicit ORDER BY statement.



Your Performance IS Our Business

10713 RR 620 North, Building 400

Austin, Texas 78726-1708

(512) 249-2324 (866) 773-8789

www.DBISoftware.com

Occasionally a statement will return rows in a specific order as a side effect of the execution plan. An example would be an index scan on the leading edge of a composite key always returning the rows in alphanumeric order. If the same result set were selected from the underlying table instead, there is a strong possibility that the rows would be returned in a different order. In this case we assume that if an ORDER BY clause is present, then the order should be preserved by all transformations; if an ORDER BY clause is not present, then the presence of side-effects is ignored and the result set can be returned in any order.

The QRC applies a designation to each transformation. A transform is either safe in which case the correct result set can always be guaranteed, or unsafe in which case the correct result set can usually be guaranteed, but this is data dependent. Unsafe transformations are a consequence of the QRC having insufficient information to guarantee the transformation.

Unsafe transformations can often be made safe by the introduction of additional constraints.

For example the query

```
SELECT COUNT (*) FROM t1,t2
WHERE t1.c1 * t2.c2 = 100;
```

can be transformed into

```
SELECT COUNT (*) FROM t1,t2
WHERE t1.c1 = 100 / t1.c2;
```

which may allow the use of an index.

This transformation is unsafe because it will fail with the error

```
ORA-01476: divisor is equal to zero
```

if any row in c2 contains the value 0.

However, it is possible that column c2 can never contain the value 0, but that this constraint is enforced by the application instead of the database engine.

In this case, depending on the circumstances, the user may choose to add a check constraint to the table e.g.

```
ALTER TABLE t1 ADD CONSTRAINT con1 CHECK (c01 > 0);
```

The user could also choose to ignore the rewrite entirely.

There are two categories of query where the rows returned and/or the number of rows returned is unpredictable. These are statements including the ROWNUM pseudo-column and the SAMPLE clause.

ROWNUM pseudo-column

Certain categories of SQL statements return random result sets. Oracle has long included the ROWNUM pseudo-column. This is normally used with in conjunction with an ORDER BY clause to restrict the number of rows returned. Prior to Oracle 8.1.5, the ROWNUM filter was always applied before the ORDER BY operation. Users often found that such queries did not return the intended results. This is because the rows returned after applying the ROWNUM



Your Performance IS Our Business

10713 RR 620 North, Building 400
Austin, Texas 78726-1708
(512) 249-2324 (866) 773-8789
www.DBISoftware.com

filter would be the first rows returned from the underlying access paths. Changes in the execution plan or the underlying data (for example unloading and reloading) might generate different result sets.

Since Oracle 8.1.5, it has been possible to use an ORDER BY clause in an inline view. The data can be sorted first and then the ROWNUM filter applied afterwards. This technique is often used in top-N queries.

The QRC considers statements using the ROWNUM pseudo-column to be safe if and only if they are executed as part of a top-N query. All other ROWNUM statements are considered unsafe.

SAMPLE keyword

Oracle 8.1.5 also saw the introduction of the SAMPLE keyword. This is applied to a scan to select a sample of the rows. A sample operation may select a different set of rows and also a different number of rows from the base table each time it is executed. The QRC therefore considers all sample queries to be unsafe.

Query Rewrite Component Transformations

The transformations applied by the QRC can be divided into a number of categories

- Simple transforms – These can be applied to any statement
- Subquery transforms – These are applied to statements containing subqueries
- Join transforms – These are applied to statements joining two or more tables.
- Hint transforms – These can be applied to any statement

Simple Transforms

The QRC applies a number of simple transformations, where appropriate, to each SQL statement. Some of these are equivalent to the heuristic transformations applied by Oracle; others are QRC specific.

Many of the transformations in this category are applied to predicates and are designed either to facilitate or to suppress the use of indexes.

Transformation of SUBSTR built-in function to LIKE condition
Transformation of arithmetic expressions
Transformation of IN list to OR conditions
Transformation of NOT IN list to AND conditions
Transformation of BETWEEN expression to >= AND <= conditions
Transformation of NOT BETWEEN expression to < OR > conditions
Elimination of IS NULL expression with a NOT NULL column
Elimination of IS NOT NULL expression with a NOT NULL column
Transformation of ANY/SOME expressions to OR conditions
Transformation of ALL expression to AND conditions
Transformation of OR conditions to UNION ALL
Transformation of OR conditions to UNION
Evaluation of constant expressions
Elimination of redundant expressions
Elimination of redundant UPPER built-in function
Elimination of redundant LOWER built-in function
Transformation of NOT (exp1 OR exp2) to NOT (exp1) AND NOT (exp2)



Your Performance IS Our Business

10713 RR 620 North, Building 400

Austin, Texas 78726-1708

(512) 249-2324 (866) 773-8789

www.DBISoftware.com

Transformation of NOT (exp1 AND exp2) to NOT (exp1) OR NOT (exp2)
Elimination of TO_NUMBER built-in function
Transformation of LIKE condition to equality condition
Transformation of OR conditions to IN LIST
Transformation of double negative expressions
Transformation of negated relational expressions
Transformation of NULL expressions

Example

This is an example of the transformation of arithmetic expressions.

The statement

```
SELECT COUNT(*) FROM t1 WHERE c1 - 1 = 0
```

generates the execution plan

```
SELECT STATEMENT Optimizer=CHOOSE  
SORT (AGGREGATE)  
TABLE ACCESS (FULL) OF 'T1'
```

The statement is transformed by moving the -1 to the right hand side of the expression

The transformed statement

```
ELECT COUNT(*) FROM t1 WHERE c1 = 1
```

can now generate the execution plan

```
SELECT STATEMENT Optimizer=CHOOSE  
SORT (AGGREGATE)  
INDEX (RANGE SCAN) OF 'I1'
```

This execution plan is potentially much more efficient than the full table scan.

Subquery Transforms

The Oracle Optimizer will attempt to flatten some subqueries. The QRC transforms various types of subquery including some not attempted by the Oracle Optimizer.

Subqueries can appear in many parts of an SQL statement; as in-line column views in the SELECT list, as in-line views in the FROM clause and as predicates in the WHERE clause.

Statements can use a number of operators with predicate subqueries including all six relational operators, EXISTS, IN, ANY, SOME, and ALL. The most effective transformations involve the EXISTS, IN, NOT EXISTS and NOT IN.

Subqueries can be correlated or uncorrelated. In a correlated subquery, one of the tables in the parent query is referenced in the subquery; in an uncorrelated subquery, the subquery does not contain any references to the parent query.



Your Performance IS Our Business

10713 RR 620 North, Building 400

Austin, Texas 78726-1708

(512) 249-2324 (866) 773-8789

www.DBISoftware.com

The QRC performs the following subquery transformations

Transformation of correlated NOT EXISTS subquery to NOT IN subquery
Transformation of NOT IN subquery to correlated NOT EXISTS subquery
Transformation of correlated EXISTS subquery to equijoin
Transformation of NOT IN subquery to MINUS subquery
Transformation of NOT IN subquery to outer join query

This is an example of the transformation of a NOT IN subquery to an outer join query, developed on an Oracle 8.1.7 database.

The statement

```
SELECT c1 FROM t1
WHERE (c1,c2) NOT IN
(
    SELECT c1,c2 FROM t2
    WHERE c2 < 5
)
AND c1 < 100;
```

generates the execution plan.

```
SELECT STATEMENT Optimizer=CHOOSE
  FILTER
    TABLE ACCESS (BY INDEX ROWID) OF 'T1'
      INDEX (RANGE SCAN) OF 'I1' (UNIQUE)
  FILTER
    TABLE ACCESS (BY INDEX ROWID) OF 'T2'
      INDEX (UNIQUE SCAN) OF 'I2' (UNIQUE)
```

The statement is transformed by converting the NOT IN (anti) join to an outer join

The transformed statement

```
SELECT t1.c1 FROM t1, t2
WHERE t2.c1(+) = t1.c1
AND t2.c2(+) = t1.c2
AND t2.c2(+) < 5
AND t2.c1 IS NULL
AND t1.c1 < 100;
```

can now generate the execution plan

```
SELECT STATEMENT Optimizer=CHOOSE
  FILTER
    HASH JOIN (OUTER)
      TABLE ACCESS (BY INDEX ROWID) OF 'T1'
        INDEX (RANGE SCAN) OF 'I1' (UNIQUE)
      TABLE ACCESS (FULL) OF 'T2'
```



Your Performance IS Our Business

10713 RR 620 North, Building 400

Austin, Texas 78726-1708

(512) 249-2324 (866) 773-8789

www.DBISoftware.com

This execution plan has the same Oracle cost as the original. However, when the statements are executed, the transformed statement requires significantly fewer logical I/Os than the original.

Join Transforms

As the number of tables being joined together in an SQL statement increases, so does the probability that the execution plan will be sub-optimal. In general, the number of execution plans available to the Optimizer is proportional to the number of permutations of the tables in the query.

The following table shows the number of permutations between tables in an SQL statement from 1 to 10 tables.

Number of Tables	Number of Permutations
1	1
2	2
3	6
4	24
5	120
6	720
7	5040
8	40320
9	362880
10	3628800

For statements involving joins between large numbers, the CBO attempts to reduce the number of permutations in several ways.

An equijoin can be performed in either direction, i.e. either table can be selected as the driving table. However, an outer join can only be performed in one direction i.e. only one table can be selected as the driving table. Each outer join consequently reduces the total number of permutations by 50%.

The CBO can also reduce the number of permutations considered by applying the `OPTIMIZER_SEARCH_LIMIT` parameter. This was a supported parameter until 8.1.5. In Oracle 8.1.6, it is a hidden (unsupported) parameter. The default value is 80000. If required the user can increase this parameter at the session level for specific queries to allow more paths to be considered. However parse times and resource consumption will increase.

Oracle also attempts to intelligently prune the execution plans considered. If when processing a branch, there is no likelihood of improving on the lowest cost already obtained, then that branch is abandoned and processing moves to the next branch.

Since the object of the QRC is to generate different execution plans from those already considered by the CBO, the QRC does not attempt to replicate the behavior of the CBO. Instead, effort is concentrating on identifying a small number of join orders that have the highest probability of success. The QRC identifies sets of tables that can be joined together, and then calculates the selectivity and cardinality for each set. Six join orders are derived for each set by applying different algorithms to the tables in the join set. Finally, the join sets are permuted together and the statements are regenerated. The use of join sets allows the same algorithm to process Cartesian joins.

This method of processing join orders has two strengths. Firstly, It is different from the method that Oracle implements, thereby increasing the probability that different execution plans will be found. Secondly, much of the processing is



Your Performance IS Our Business

10713 RR 620 North, Building 400

Austin, Texas 78726-1708

(512) 249-2324 (866) 773-8789

www.DBISoftware.com

performed and many potential execution plans are eliminated on the client before any parsing is performed, thus minimizing the impact on the server.

Hint Transforms

Hints are directives to the Optimizer that specify how a statement should be executed. Both the RBO and CBO can be hinted, but only a handful hints apply to the RBO. In Oracle 9.2 there are around 130 hints for the CBO, although only about 50 of these are documented.

The QRC selectively applies over 20 hints. Hints applied include the following.

Category	Hint
Optimizer Hints	ALL_ROWS
	FIRST_ROWS
	RULE
Ordering Hints	LEADING
Access Method Hints	FULL
	INDEX
	INDEX_DESC
	INDEX_FFS
	NO_INDEX
	INDEX_COMBINE
	INDEX_SS
	INDEX_SS_DESC
Anti Join Hints	HASH_AJ
	MERGE_AJ
	NL_AJ
Semi Join Hints	HASH_SJ
	MERGE_SJ
	NL_SJ
Join Method Hints	USE_HASH
	USE_MERGE
	USE_NL
Or Expansion Hints	USE_CONCAT
	NO_EXPAND

Hints are only applied where they are syntactically appropriate. Thus, the ALL_ROWS hint is only used in the top level subquery of a SELECT statement, the USE_NL hint is only used in conjunction with a join and the HASH_AJ is only used if the statement contains an anti-join.

Hints are always applied after transformations have been performed. This ensures that the hints are only applied where relevant.

This is an example of the transformation of a NOT IN subquery to an outer join query, developed on an Oracle 8.1.7 database.

Example

This is an example of the FULL transformation.



Your Performance IS Our Business

10713 RR 620 North, Building 400

Austin, Texas 78726-1708

(512) 249-2324 (866) 773-8789

www.DBISoftware.com

The table in this example has a skewed data distribution. In this example 80% of the rows have the value of 1 for column c1 whilst the remaining rows have unique values.

The statement

```
SELECT c02 FROM t1 WHERE c01 = 1;
```

generates the execution plan.

```
SELECT STATEMENT Optimizer=CHOOSE
  TABLE ACCESS (BY INDEX ROWID) OF 'T1'
    INDEX (RANGE SCAN) OF 'I1' (NON-UNIQUE)
```

The statement is transformed by adding the FULL hint.

The transformed statement

```
SELECT /*+ FULL (t1) */ c02 FROM t1 WHERE c01 = 1;
```

generates the execution plan.

```
SELECT STATEMENT Optimizer=CHOOSE
  TABLE ACCESS (FULL) OF 'T1'
```

The Oracle cost of the transformed statement is much higher than the Oracle cost of the original. This is due to the skewed data. In this example, the number of logical I/Os was reduced by over 50% by the new execution plan.

Conclusion

The Oracle Optimizer has become very sophisticated and capable of generating highly efficient execution plans. However, it can still occasionally generate spectacularly poor execution plans, which, if undetected, can have a devastating effect on even a well-tuned system. Whilst the Optimizer improves with each new release, few, if any, of the improvements are back-ported to older versions of Oracle. Thus, the majority of the customer base has to wait many years to benefit from the enhancements. In addition, the automatic transformations applied by the Optimizer tend to lag behind new features by one or two releases. Consequently, it is not always possible to utilize new features because of deficiencies in the Optimizer.

The DBI Query Rewrite Component addresses these weaknesses. All releases from 7.3.4 upwards are supported and transformations can be applied to all releases to which they are valid. The QRC has been designed to be highly extensible, so adding new transformations and also adding support for new Oracle versions is a straightforward process ensuring that the component can constantly improve. The QRC therefore allows users to maintain and protect their investment in performance tuning as they move through Oracle versions.