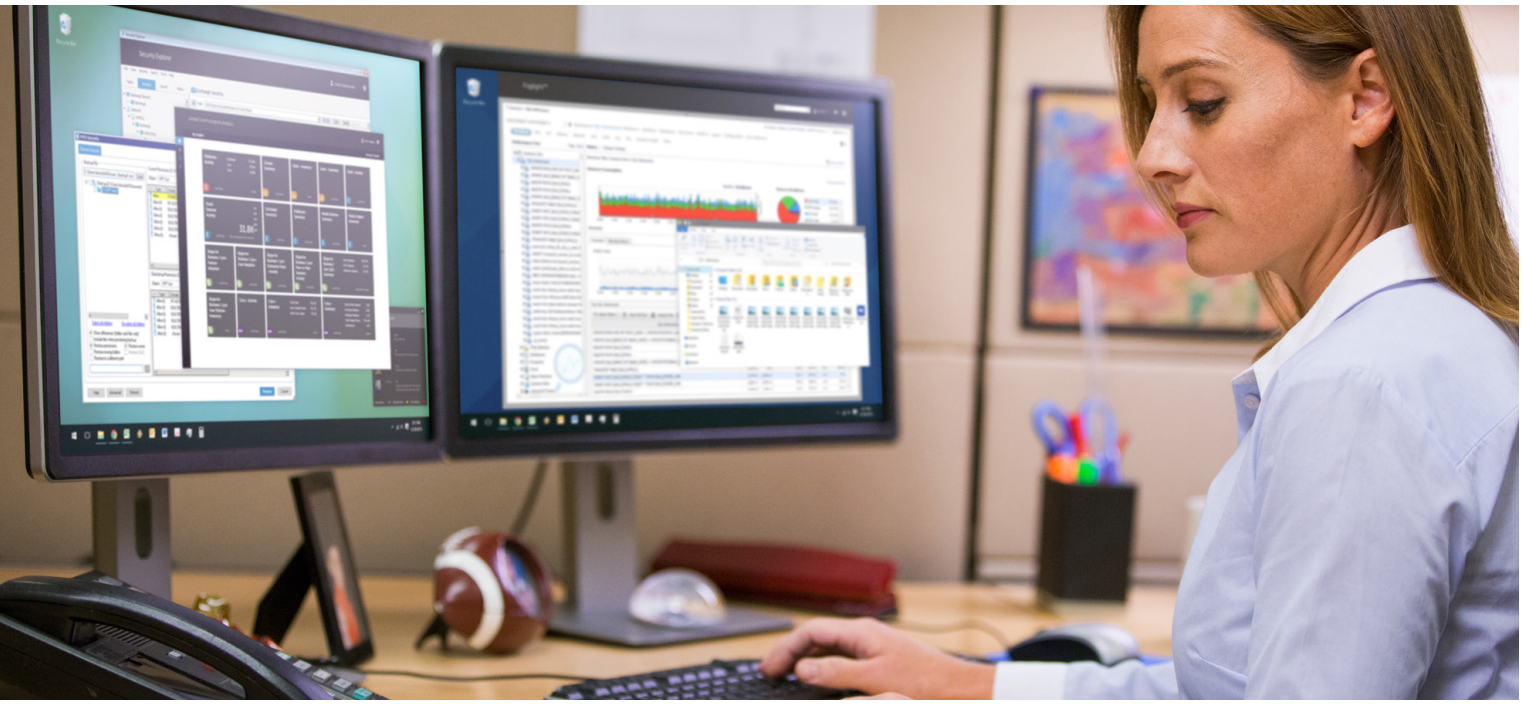
Solving the SQL Tuning Problem

Secrets of Quest® SQL Optimizer



INTRODUCTION

For years, commercial database manufacturers have fought an endless battle to improve the performance of inserting, updating, deleting and retrieving information stored in the database. Despite their ongoing efforts and hard work, we have not seen a significant improvement in the performance of most relational database management systems (RDBMS): users still suffer from under-performing SQL statements, and database experts are still spending countless hours tuning SQL statements.

Quest SQL Optimizer provides a solution for improving SQL statement performance that an internal database management system's SQL optimizer cannot provide. This paper explains how.

THE CHALLENGES WITH INTERNAL DATABASE SQL OPTIMIZERS

Major deficiencies cause the database's internal SQL optimizer to fail to find a good execution plan for a SQL statement. Two of these deficiencies include inaccurate database statistics and SQL cost estimation of the resources that may be used for the SQL statement.

Statistics and index statistics may not be up to date.

Batch mode statistics collection and a near real-time data sampling technique are two common methods to collect database statistics. The batch mode statistics collection method processes a large set of data and uses system resources. This

takes time to collect the data statistics. Consequently, these collections are normally executed during non-peak hours, when the systems administrator determines the statistics to update. If many INSERT or DELETE operations have occurred since the last collection of statistics, then these statistics will not reflect the most up-to-date information when a SQL statement is executed. This may cause the database SQL optimizer to generate a poorly performing execution plan for the current data distribution.

Fortunately, in the last few years, database vendors have improved the collection of statistics with a new real-time data sampling technique. This allows small amounts of data to be sampled for the statistics collection. This requires the data to be equally distributed so the sampling data can fully reflect the true data distribution. The batch mode statistics collection method is the alternative to generate accurate statistics for the database if the data is not equally distributed.

Even if the database statistics are accurate, the database SQL optimizer cannot correctly determine the best execution plan since it does not have all the pertinent information.

Cost estimation may not be accurate.

The second deficiency that causes the database SQL optimizer to select a poor execution plan is having inaccurate cost estimations. It is obvious that incorrect statistics will result in inaccurate cost estimation for a SQL statement. Even if the database statistics are accurate, the database SQL optimizer cannot correctly determine the best execution plan since it does not have all the pertinent information. Let's examine the following SQL:

```
SELECT *
FROM A, B, C
WHERE A.key1 = B.key1
AND B.key2 = C.key2
AND C.f2 = 0
AND A.f1 = 0
```

In this SQL statement, three tables are joined. We assume the database SQL optimizer will use a nested loop join only when all of the statistics are available for the tables and indexes used in this SQL statement. In this example, we find from reviewing the data that the driving path from table C➔B➔A (Figure 1) takes more time than that of the driving path A➔B➔C (Figure 2). But most of database SQL optimizers will pick up the driving path C➔B➔A. This happens because C.f2 has a unique index and the database SQL optimizer does not know which values will be returned from C.key2. As you can see from this example, the data itself is determining the number of records retrieved and the cost to process the execution plan. The database SQL optimizer does not have this information, and therefore it cannot choose the best execution plan.

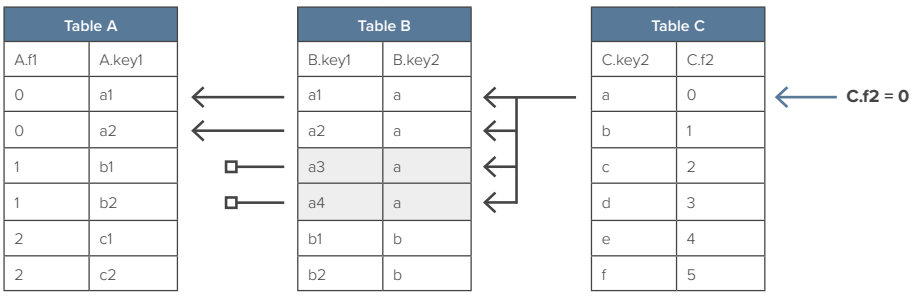


Figure 1. The driving path starts from table C to table B, which matches four rows from table B. Then those four rows are matched to the rows in table A. The rows highlighted in gray in table B show the two records that caused extra scan operations using this driving path.

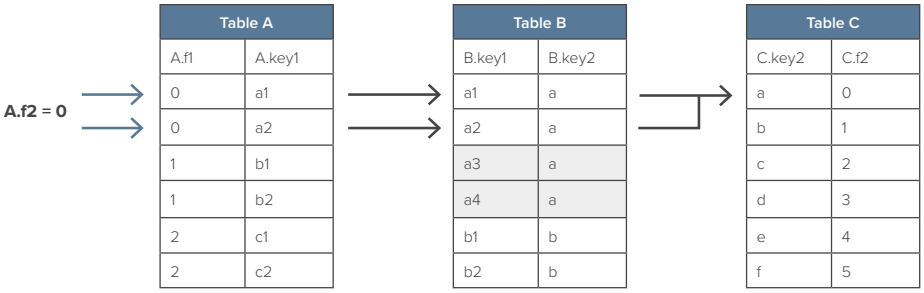


Figure 2. The driving path starts from table A to table B, which matches two rows. Then those two rows are used to match to the rows in table C. So with this driving path, no unnecessary table scans are performed. Though starting with a condition of lower selectivity, the overall scan operations are more optimistic.

THE CHALLENGE OF DESIGNING AN INTERNAL SQL REWRITE FUNCTION

Today, the internal SQL optimizer in most database management systems has a limited built-in SQL rewrite ability (see Figure 3). The internal rewrite is used to correct some obvious programming mistakes or to rewrite the input SQL to an internal SQL syntax. This internal SQL syntax will make the later stage of optimization easier for transforming an IN or EXISTS sub-query to a JOIN statement or making use of materialized views. Some databases have a weak internal SQL rewrite ability, which leaves room for database tuning experts to influence the database SQL optimizer to make a better choice in the later stage of the execution plan generation. Of all the databases, Oracle has the most open architecture to accept the user's influence through use of optimization "hints." Oracle is also the most syntax sensitive database that allows users to tune their SQL statements by restructuring the SQL syntax.

In contrast, a database like IBM DB2 UDB (Universal Database) has a very strong internal SQL rewrite ability. In IBM DB2 UDB, most SQL statements can be transformed into their internal SQL syntax before they are further optimized. This appears to be good news that would make it unnecessary for programmers to tune their SQL statements, since the database SQL optimizer would be doing the job for them. But the problem with this approach is that when the database SQL optimizer makes a mistake on a specific SQL statement, it is hard for the programmer to influence the database SQL optimizer to pick up other choices.

This illustrates the dilemma that database engineers always face: Do they make the database smart, so it makes intelligent choices for the user? Or do they give the user the control, since the database SQL optimizer will not always make an intelligent choice?

Should database engineers make the database smart, so it makes intelligent choices for the user? Or do they give the user the control, since the database SQL optimizer will not always make an intelligent choice?

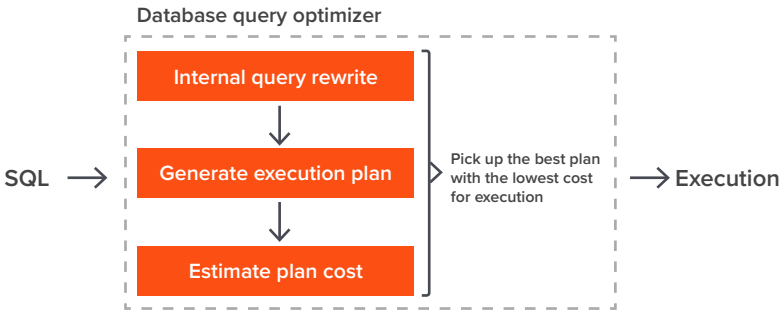


Figure 3. Database SQL optimizer processing steps

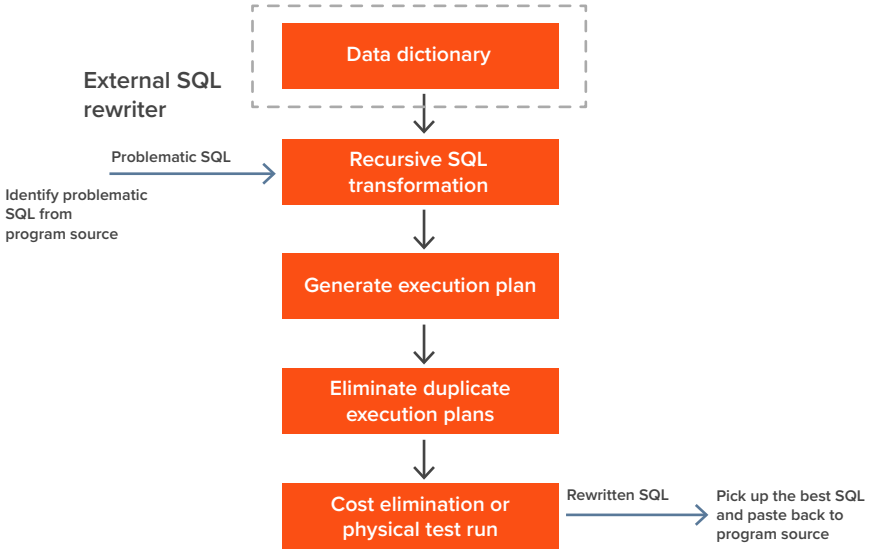


Figure 4. Quest SQL Optimizer is an external SQL rewriter that mimics human expertise to rewrite SQL statements to help the database's internal SQL optimizer make better decisions.

Quest SQL Optimizer mimics human expertise to rewrite SQL statements to help the database's internal SQL optimizer make better decisions.

HOW QUEST SQL OPTIMIZER SOLVES THE SQL PROBLEM

Quest SQL Optimizer mimics human SQL rewriting skills.

Quest SQL Optimizer is an external SQL rewriter that mimics human expertise to rewrite SQL statements to help the database's internal SQL optimizer make better decisions. A human expert may know more about the data in the database and its distribution. A database programmer with a good comprehension of SQL and its execution can rewrite a SQL statement to guide the internal database SQL optimizer to make a better choice among all its internally generated execution plans. Since the number of execution plans generated by the internal database SQL optimizer is limited by the syntax of the SQL statement, an experienced database tuner can rewrite the SQL syntax to force the internal database SQL optimizer to pick up a better execution plan.

Figure 4 shows how Quest SQL Optimizer mimics human SQL rewriting skills. Of course, a software program cannot know how to rewrite a SQL statement to the best syntax for a particular database environment, but Quest SQL Optimizer has the capability to try every possible rewrite (within the quota limits that you determine) and thereby find the best solution.

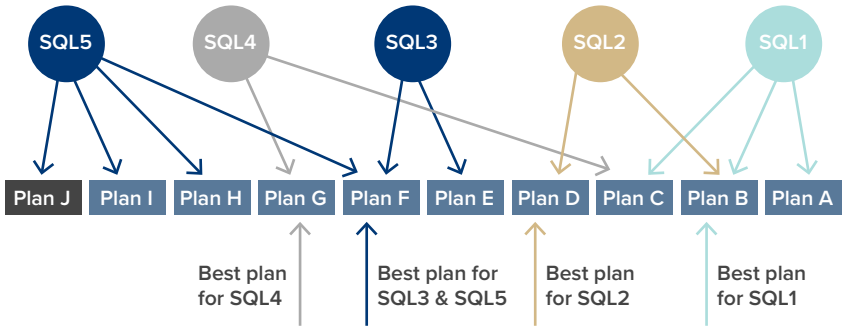


Figure 5. Relationships with SQL syntax changes and the execution plan generation

Quest

CHANGING SQL SYNTAX TO IMPROVE SQL PERFORMANCE

Limitations of internal SQL rewrite functionality

Due to limitations in the database's internal SQL rewrite capability, it cannot transform a SQL statement into very many semantically equivalent SQL alternatives. Therefore, it is not possible for it to generate every possible way to rewrite a SQL statement. For example, assume that five SQL statements from SQL 1 to SQL 5 are semantically equivalent but syntactically different. The database SQL optimizer may generate a different set of execution plans accordingly (see Figure 5). For each set of execution plans (SQL 1 has Plan A, B, C), the database SQL optimizer will carry out a cost estimation and will execute the execution plan with the lowest cost. If the database SQL optimizer fails to select a good execution plan due to an inaccurate cost estimation or because it is limited by the number of execution plans generated, the corresponding SQL statement's performance will be degraded.

In order to rectify this situation, a programmer can rewrite the original SQL multiple times. For each rewritten SQL, the database SQL optimizer creates a set of execution plans. Based on this new set of execution plans, the database SQL optimizer may select to use an execution plan that was not found in the original set of plans. If one of the execution plans from those rewritten SQL has better performance, the programmer can use the new syntax to replace the original SQL in the source to improve the SQL performance without changing the results. This is what we call SQL tuning without creating new indexes or database configuration changes.

SQL tuning experts may find that some rules of SQL syntax restructuring improve SQL statements in certain environments. For example, a nested loop join accessing a small table first and using an index search on a large table is normally better than joining the data from the large table to the small table. But for complex SQL statements with many table joins, a general rule like that may be hard to apply. This is especially true in situations with many join operations, where sorting and filtering methods are combined into one long and complex execution plan.

Quest SQL Optimizer's Recursive SQL Transformation Engine

The Recursive SQL Transformation technology used in Quest SQL Optimizer simulates human SQL transformation techniques. It incorporates a set of transformation rules to transform SQL statements on a section-by-section basis. This replaces the trial-and-error method used by a person to rewrite the syntax of a SQL statement.

The database's internal SQL rewrite capability cannot transform a SQL statement into very many semantically equivalent SQL alternatives.

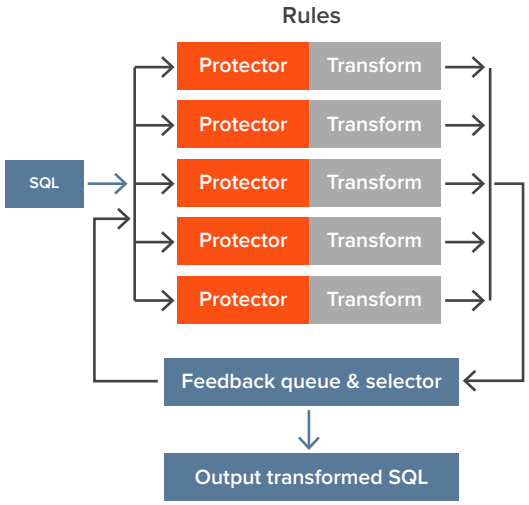


Figure 6. With Quest SQL Optimizer's Recursive Transformation Engine, each transformation rule in the optimization engine is independent from one another, like a capsule; the rule's 'capsule' can be opened only when all necessary conditions are satisfied.

The Recursive SQL Transformation Engine replaces the trial-and-error method used by a person to rewrite the syntax of a SQL statement.

Each transformation rule in the optimization engine is independent from one another, like a capsule; the rule's 'capsule' can be opened only when all necessary conditions are satisfied (see Figure 6). This guarantees the semantic equivalence of the rewritten SQL statements so they produce the same results as the original SQL. When a SQL statement is transformed by one rule to produce a new SQL syntax, the new syntax may now satisfy the requirements of another rule; hence, transformation is carried out in a recursive manner (see Figure 7).

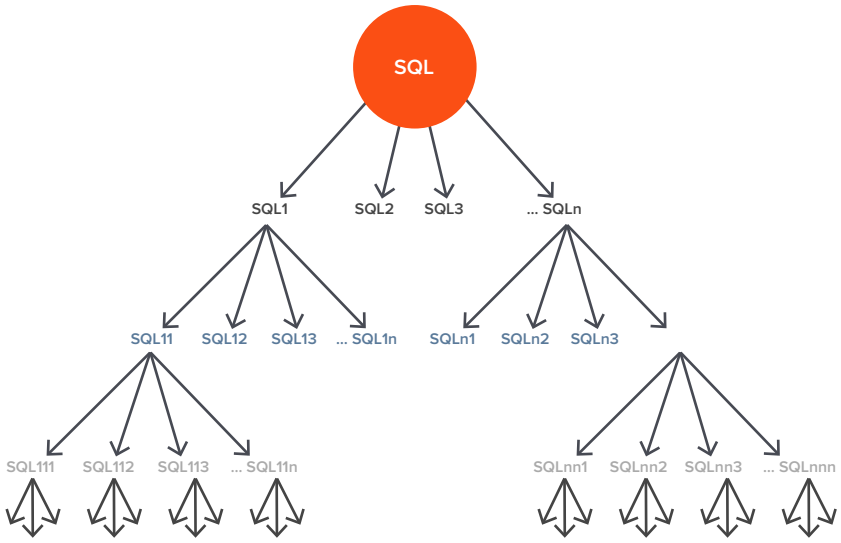


Figure 7. Chain effect of SQL transformation: when a SQL statement is transformed by one rule to produce a new SQL syntax, the new syntax may now satisfy the requirements of another rule; hence, transformation is carried out in a recursive manner.

Quest

```
SELECT *
        FROM A
                WHERE A.C1 IN (SELECT B.C1
                        FROM B
                                WHERE EXISTS (SELECT 'x'
                                        FROM C
                                                WHERE B.C2 = C.C2 ))
```

Example

Let's take a look at the following SQL statement and use two of the built-in transformation rules used by Quest SQL Optimizer to see how this recursive transformation works. We will use one rule to transform the IN condition to an EXISTS condition and then use another rule that does the reverse, changing the EXISTS condition to an IN condition. We will illustrate this with the following SQL statement.

The first two levels of transformation are shown in the left side of Figure 8. SQL statements with syntax different from the original can be produced by following a set of transformation rules. You can see that for each rule applied to the SQL statement, the newly transformed SQL will satisfy another rule. The order in which the rules are processed can result in different SQL alternatives. In this example, the source SQL has gone through two transformation rules executed in a recursive manner. If we do not stop the recursive transformation, the loop will continue indefinitely.

A total of four unique SQL statements (marked by the solid boxes in Figure 8) are generated by the two transformation rules. If each of these SQL statements ends up with a new execution plan, we potentially have three SQL statements that may give us different performance to be used as a benchmark to the original SQL statement.

SQL statements with syntax different from the original can be produced by following a set of transformation rules.

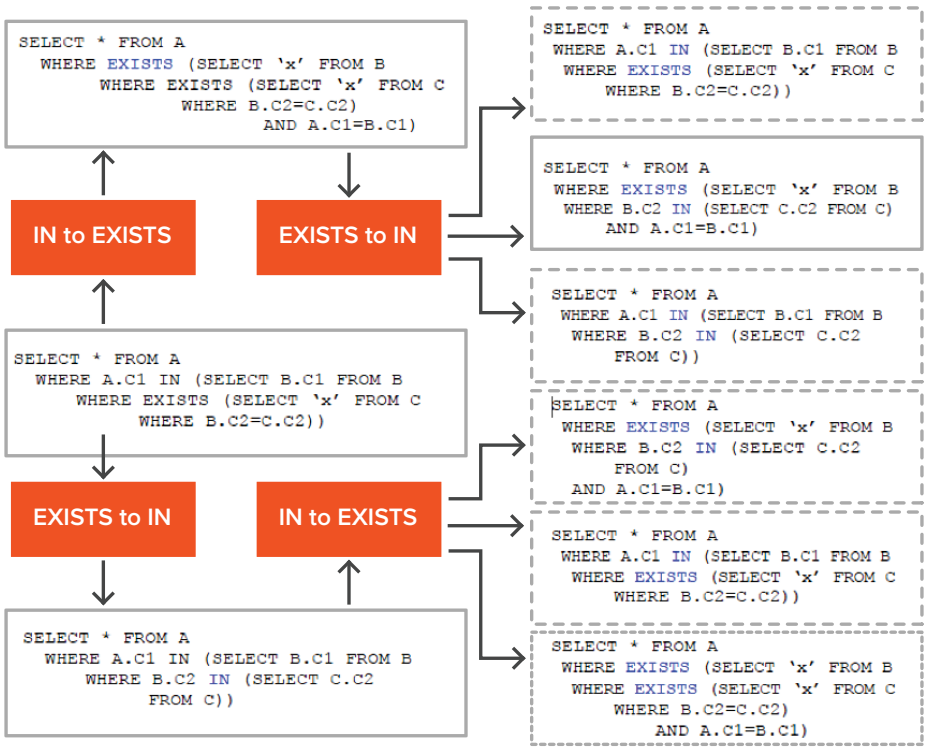


Figure 8. A SQL statement being transformed by two recursive transformation rules

Figure 9. If a programmer tells you that table B is not necessary, the SQL on the left can be rewritten into a SQL statement that eliminates table B and joins only tables A and C.

The sole purpose for transforming the syntax of a SQL statement is to find a new syntax that can potentially influence the database SQL optimizer to pick up a better execution plan.

Implementing a transformation rule requires a more complicated control than is shown in this illustration. For example, Quest SQL Optimizer has to check whether a set operator (UNION, MINUS or INTERSECT) is in a sub-query, whether multiple items are in the SELECT list, and much more. If these rules are self-protected, it means the transformation and conditional checking are encapsulated into a one-rule capsule to prevent generating incorrect SQL statements (SQL statements that do not produce the same result as the original SQL).

Quest SQL Optimizer's Recursive Transformation Engine has a multitude of transformation rules that can handle very complicated situations. The result of the recursive transformation for a complex SQL statement may sometimes exceed what you can imagine. For example, some transformation rules can be applied endlessly to transform a SQL statement to another semantically equivalent statement without limitation, so quotas must be used to control the number of SQL alternatives generated and available in Quest SQL Optimizer.

SQL transformation rules provide semantic equivalence.

Since a computer cannot understand the relationship and meaning of data stored in a database, a software program cannot transform a logically clumsy SQL statement into a logically simple one. For example, the following SQL statement joins three tables to select the A* records. If a programmer tells you that table B is not necessary, the SQL can be rewritten into a SQL statement that eliminates table B and joins only tables A and C.

If we review the syntax of the two SQL statements in Figure 9, we cannot determine that the two SQL statements are semantically equivalent. However, a programmer who understands the data can simplify the logic from a three-table join into a SQL statement with a two-table join.

Quest SQL Optimizer uses transformation rules to transform SQL statements to other syntax that is semantically equivalent to the original SQL statement—that is, where the SQL alternative produces the same result as the original SQL statement. The sole purpose for transforming the syntax of a SQL statement is to find a new syntax that can potentially influence the database SQL optimizer to pick up a better execution plan.

Example: Quest SQL Optimizer's transformation rules for correcting common errors

What are some examples of SQL transformation rules that are used in Quest SQL Optimizer? The common error rules are used to correct common, but inefficient, practices that programmers may use when creating SQL statements. Figure 10 shows two examples.

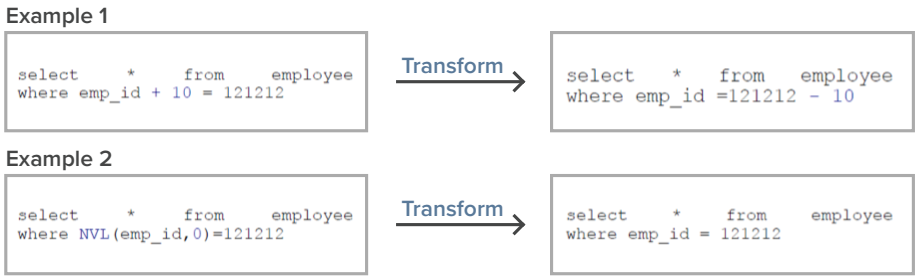


Figure 10. Quest SQL Optimizer includes rules to correct common but inefficient practices in writing SQL.

For Example 1 in Figure 10, the operation, +10, was applied to the emp_id column. This causes a full table scan since 10 must be added to the emp_id value for each row of the table. By transforming the syntax to subtract 10 from the other side of the operation, it makes it possible for an index to be used.

For Example 2 in Figure 10, Quest SQL Optimizer will check the emp_id to determine if the column was defined as NOT NULL. If it is a NOT NULL field, the NVL function can be eliminated. This transformation may also eliminate a full table scan by making it possible for an index to be used.

JOIN PATHS

Why evaluating join paths is complicated

The join path is the order in which the data is retrieved from two or more tables. Theoretically, the database SQL optimizer should find the best path to join two or more tables for a given SQL statement. The problem is that when there are many table operations in a SQL statement, there is no way for the database SQL optimizer to evaluate all possible paths for joining the tables, given the limited amount of time it has to select an execution plan. Let's take a look at the following SQL statement. It involves joining eight tables, and each table has a relationship with the seven other tables. If we consider only the Nested Loop join, the total permutation is 8!= 40,320 possibilities. So, you can see it would take a long time for the database SQL optimizer to evaluate all possible paths.

I have seen a SQL statement (in an electric power company's application) that involved more than a 13-table join. If you calculate all possible join paths, the permutation would be 13!= 6,227,020,800. This would almost require a supercomputer to do an optimization before the execution of the SQL statement. Consequently, the database SQL optimizer does a rough estimation and runs the SQL immediately, which is much faster than trying to do a comprehensive analysis to find the very best table join path before beginning the execution.

```
select * from
T1,T2,T3,T4,T5,T6,T7,T8
where T1.key=T2.key
and T2.key=T3.key
and T3.key=T4.key
and T4.key=T5.key
and T5.key=T6.key
and T6.key=T7.key
and T7.key=T8.key
```

Why join paths matter

The basic nested loop join operation is supported by most RDBMS since it requires less memory and temporary space. Normally, it provides faster data response time than other join operations. However, the path of a nested loop join will significantly affect the speed of the join operation. Let's use a two-table join as an example to understand how this works:

When there are many table operations in a SQL statement, there is no way for the database SQL optimizer to evaluate all possible paths for joining the tables.

```
select * from A,B          Let's assume that A.key and B.key are
                           unique B-Tree indexed
   where A.key = B.key     (assume B-tree has two nodes for each parent)
                           A table has 100,000 records
                           B table has 1000 records
```

If we focus on the Nested Loop join operation and assume these two tables are cached in memory, we can calculate the number of operations to retrieve both tables for the two possible join paths:

The path from table A to table B means that we open table A, looking at each row to then use an index to search for matching rows in table B:

Number of operations (A➔B) = 100,000 * RoundUp(LN(1000) / LN(2)) / 2 = 100,000 * 10 / 2 = 500,000

Where LN(1000)/LN(2) is the height of the B-tree for index of B.key, half the height is assumed as an average searching depth for a specific record from B.key.

The path from table B to table A means that we open B table, looking at each row to then use an index to search for matching rows in table A:

Number of operations (B➔A) = 1000 * RoundUp(LN(100,000) / LN(2)) / 2 = 1000 * 17 / 2 = 8,500

Where LN(100,000)/LN(2) is the height of the B-tree for index of A.key, half the height is assumed as an average searching depth for a specific record from A.key.

According to the calculation, you will find that the path from B➔A is around 500,000/8,500, or ~59 times faster than the speed of A➔B. This explains why some SQL statements with a wrong driving path can be tuned up to hundreds of times faster.

Many programmers may have learned from experience to use the small table to drive a bigger table for a nested loop join produces faster results. When a SQL statement is simple and natural, it is likely a programmer will write the SQL statement using the best driving path.

But in a real live application, the SQL statements can be far more complicated than a simple two-table join operation. For example, the key for both tables may not be unique. There may be some filtering criteria for both tables that make it so no histogram can be referenced. This makes it so that the database SQL optimizer cannot accurately estimate the cost of each join path. Human expertise is needed to solve the problem when the database SQL optimizer chooses a poor execution plan.

For other join methods, such as hash join or sort merge, the join path may not significantly affect the SQL speed for a two-table join. But in some situations, like those illustrated in Figure 1 and Figure 2, you may find that the join path still plays a major role in the performance of a SQL statement.

How to control the join path

The design concept used in Quest SQL Optimizer to apply recursive SQL transformation rules is different from common SQL tuning tools or human SQL tuning knowledge. Quest SQL Optimizer is not attempting to directly find the best alternative SQL rewrite like a human expert would. Quite frankly, no human knowledge can address all possible combinations of SQL syntax, hardware configurations and software configurations—or predict the behavior of the database SQL optimizer—to immediately know what the best SQL syntax is for a given SQL statement.

To control a join path, we cannot tell the internal database SQL optimizer which path is the best one to select. Instead, we add something to the syntax of the SQL statement that causes an increase to the cost of the current join path selected by the internal database SQL optimizer.

Quest SQL Optimizer is not attempting to directly find the best alternative SQL rewrite.

```
select * from A,B
 where A.key = B.key
```

Figure 11. Two-table join scenario

Example: a two-table join

Let's take a look at an example of a two-table join scenario, shown in Figure 11.

If we consider the nested loop join, two paths can be considered by the database SQL optimizer, which are A➔B and B➔A. After the cost estimation, the database SQL optimizer may think that B➔A has the lower cost, so, the database SQL optimizer will select the join path of B➔A. If we know that the join path selected by the database SQL optimizer is not the optimal path, we should be given an opportunity to influence the database SQL optimizer to select another path. For some databases, like Oracle, if you know which path is the best, you can use the optimization hints to influence the database SQL optimizer to pick the right path. In some databases, such as IBM DB2 UDB, there are no execution plan hints available. Therefore, the rewriting of the SQL syntax is the only tool that can be used to influence the database SQL optimizer to pick the right path.

```
select * from A,B
    where A.key + 0 = B.key
```

Let us rewrite the following SQL syntax and assume that the datatype for A.key and B.key is numeric.

In Oracle and Sybase Adaptive Server, the index search of A.key on table A is disabled by changing the syntax to A.key + 0. The addition of +0 does not affect the value of A.key, but it does cause a full table scan on A table. The cost estimation of the join "from table B to index search table A" will be artificially increased by this new syntax. If the new cost is higher than the path of A➔B, the database SQL optimizer will pick up the execution plan of "from table A to index search table B."

In a real-life situation, the database SQL optimizer may not actually select the expected nested loop execution plan "from table A to index search table B" when you change the syntax to A.key + 0. This syntax change would increase the cost of the nested loop execution plan "from table B to index search table A," but the database SQL optimizer may select the second lowest cost execution plan, which could be a hash join or a sort merge instead of the nested loop execution plan.

```
select * from A,B
     where coalesce(A.key,A.
     key) = B.key
```

The rewriting of the SQL syntax is the only tool that can be used to influence the database SQL optimizer to pick the right path.

```
select * from A,B,C
 where A.key = B.key
   and B.key = C.key
```

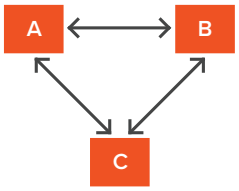


Figure 12. Three-table join scenario

How can we guide the database SQL optimizer to select a preferred path by rewriting the SQL syntax?

For Microsoft SQL Server and IBM DB2 UDB, the following syntax will increase the cost for a specific driving path.

Example: a three-table join

Now let's use a three-table join SQL statement to illustrate a more complicated scenario.

For a three-table join SQL statement, the database SQL optimizer will consider all the permutations, which is 3!=6. Assume that B➔A➔C is the lowest cost path and therefore is the path selected by database SQL optimizer. How can we guide the database SQL optimizer to select a preferred path by rewriting the SQL syntax? If we want to guide the database SQL optimizer to consider a path from A➔B➔C, we can try the syntax shown in Figure 13.

By changing the syntax to A.key + 0 and B.key + 0, three of the six table join permutations have an increase in cost: C ➔ B➔A, A➔C➔B and B➔A➔C. This leaves three remaining paths available for the SQL database optimizer to consider: A➔B➔C, B➔C➔A, and C➔A➔B. It will select the path of our choice, A➔B➔C, only if the estimated cost is the lowest cost; otherwise, the database SQL optimizer will opt for some other path.

With the lowering of the cost of today's CPU and memory, the database SQL optimizer designers are able to lower the cost of hash and sort merge joins, which use more processing power and memory than the nested loop join. This means that database SQL optimizer will more often select the hash or sort merge join instead of taking the risk to do a nested loop join, especially when the table size is small.

```
select * from A,B,C
 where A.key + 0 = B.key
   and B.key + 0 = C.key
```

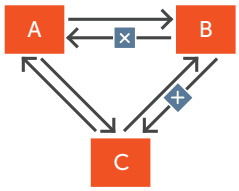


Figure 13. Rewriting the SQL syntax to guide the database SQL optimizer to select a preferred path.

```
select * from A,B,C
 where A.key + 0 = B.key
   and B.key + 0= C.key
   and A.f1 = :VAR
```

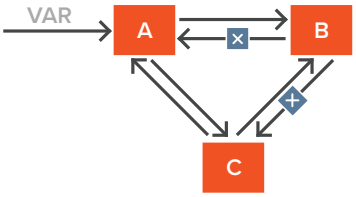


Figure 14. Rewriting the SQL syntax to include an external variable

Let's review the rewritten SQL in Figure 14, which also contains an external variable "A.f1=:VAR."

Since the database SQL optimizer always assumes that an external variable will narrow down the first result set from a table, the path VAR➔A➔B➔C normally should have the lowest cost.

INDEX USAGE

Transformation rules relating to index usage

Transformation rules relating to index usage are designed to guide the database SQL optimizer as to how it should use the indexes for a specific SQL statement. Transformations include enabling or disabling an index search, or telling the database SQL optimizer to use alternative indexes.

The two SQL statements shown in Figure 15 are quite often used in an online query system in which the user inputs values in a range from :c to :d to retrieve data from a table. If a user does not specify the range, the :c and :d values will be null. Due to the complexity of the SQL statement caused by using an OR condition plus some undetermined variables, the database SQL optimizer will usually choose a full table scan to process the SQL statement.

> Transformation rules can guide the database SQL optimizer as to how it should use the indexes for a specific SQL statement.

Example 1

```
select *
  from employee
 where (emp_id > :c or :c is null)
   and (emp_id < :d or :d is null)
```

Transform →

```
select *
  from employee
 where (emp_id > NVL(:c, -1E9))
   and (emp_id < NVL(:d, 1E9))
```

Example 2

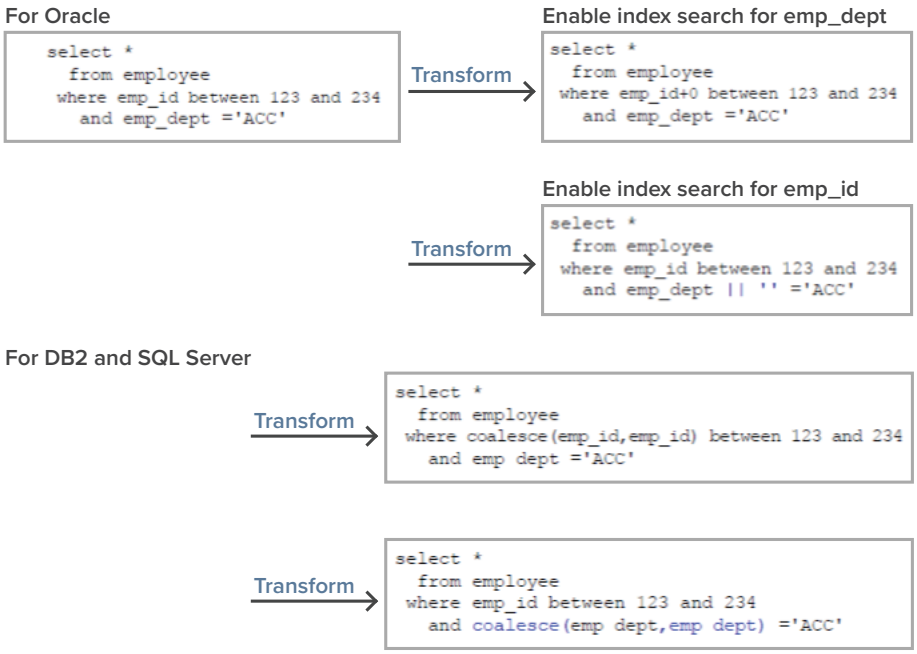
```
select *
  from employee
 where (emp_id >:c or :c is null)
   and (conditions)
```

Transform →

```
select *
  from employee
 where (emp_id > NVL(:c, -1E9))
   and (conditions)
```

Remark
This illustration assumes that emp_id is defined as NOT NULL and it is indexed. -1E9 is the lowest possible value and +1E9 is the highest possible value that can be entered into emp_id based on the length of the field.

Figure 15. SQL statements commonly used in an online query system



Remark
Since IBM DB2 UDB v8 and Microsoft SQL Server 2005 have stronger internal SQL rewrite abilities in their latest versions, the coalesce (col,col) can be resolved by the database SQL optimizer during parsing to a Case operation. Therefore, the index that you are trying to disable will remain in the execution plan. A deeper nested coalesce (coalesce (col,col),col) can be used to overload the parser and increase the specific cost weighting.

Figure 16. A transformation that can be used to enable any one of the indexes for a SQL statement having multiple indexes that can be used to search a table.

Quest SQL Optimizer also has rules for different platforms in order to deal with the behavior of the SQL optimizer for each database.

For a SQL statement having multiple indexes that can be used to search a table, the transformation shown in Figure 16 can be used to enable any one of the indexes.

To disable the index on the numeric emp_id field, zero was added to the field. This disables the index since zero must be added to emp_id for each row, requiring a full table scan or enabling a different index to be used.

The same process is used for the character field of emp_dept where nothing, represented by '' (single quotes with no value), was concatenated to the field. This also disables the index since the concatenate operation must be performed for each row, thereby requiring a full table scan or enabling a different index to be used.

The other technique for disabling the index is to use the COALESCE operation, which in this case, does nothing to the value in the field. Because it must be performed for each row in the table, it disables the index and causes a full table scan or enables a different index to be used.

DEALING WITH THE BEHAVIOR OF EACH PLATFORM'S SQL OPTIMIZER

Quest SQL Optimizer also has rules for different platforms in order to deal with the behavior of the SQL optimizer for each database. In order to understand the theory behind some of these transformations, you may need to have an in-depth understanding of database optimization theories, the design approach to optimization each database vendor has incorporated into the database SQL optimizer, and the platform-specific optimizer functions.

Here is a puzzling transformation: The original SQL statement uses a range scan of the employee table with the condition "emp_id > 123456." Both IBM DB2 UDB and Microsoft SQL Server have an intelligent algorithm that can preview the value "123456" before the execution plan is generated. Consequently, if "emp_id>123456" returns a small subset of records from the employee table, the database SQL optimizer should generate an execution plan that uses an index search.

In contrast, if the SQL statement returns almost all the records from the table, the database SQL optimizer should generate an execution plan using a full table scan to save the time of retrieving extra index pages. This works fine in most cases; however, there are several factors that can cause the database SQL optimizer to make a mistake. Three factors are:

- The statistics are not up to date.
- The data distribution is so skewed that the granularity of the histogram is too big to handle.
- The costing algorithm fails to take into account the configuration of different machines' I/O thru-put, CPU processing, memory speed and other system resources.

For Microsoft SQL Server, if you want to rectify the problem, you can use the INDEX hints to force the database SQL optimizer to pick up an index. But for IBM DB2 UDB, it is a little bit more difficult.

Let's look at the transformations in Figure 17, which use a dummy operation COALESCE (123456,123456) or add +0 to the literal 123456. The purpose of these dummy operations is to hide the value of 123456, so Microsoft SQL Server or IBM DB2 UDB will not be able to see the value while parsing the SQL statement. Therefore, they will make a rough estimation when they do not know the actual value. Erring on the side of caution, the database SQL optimizer will normally select the execution plan that uses an index search.

For the nested loop join case, the path plays an important role in determining the speed of the SQL; for hash join or sort merge join cases, the join path, may not be that significant.

Example 1

```
select *
  from employee
 where emp_id > 123456
```

Transform →

```
select *
  from employee
 where emp_id >
coalesce(123456,123456)
```

Transform →

```
select *
  from employee
 where emp_id > 123456 + 0
```

Remark
Since IBM DB2 UDB and Microsoft SQL Server have stronger internal SQL rewrite abilities in their latest versions, the coalesce (123,123) can be resolved by the database SQL optimizer during parsing to a Case operation. Therefore, the index you are trying to disable will remain in the execution plan. A deeper nested coalesce (coalesce(123,123),123) can be used to overload the parser and increase the specific cost weighting.

Figure 17. Transformation in DB2 or SQL

Example for all platforms

Remark
This transformation is valid only if there are no group, set or user-defined stored functions call in the IN sub-query.

Figure 18. Transformation rule adding a GROUP BY clause

In modern RDBMS SQL optimizers, the IN sub-query shown in Figure 18 can normally be transformed to a join SQL statement, which means the join path can be either from A_B or B_A. For the nested loop join case, the path plays an important role in determining the speed of the SQL; for hash join or sort merge join cases, the join path, may not be that significant.

Oracle provides a full set of optimization hints to help users rectify individual SQL performance problems.

The transformation rule shown in Figure 18, which adds a GROUP BY clause, serves two purposes:

- The first purpose is to force the sub-query to be processed individually. If the original execution plan is a nested loop join, after the transformation, the execution plan will normally be changed to a hash or a sort merge join.
- The additional GROUP BY function will also trim down the result set from B key (if B key is not unique) and the duplicate records will be eliminated first. This will sometimes help to improve join speed.

OPTIMIZATION HINTS

What are optimization hints?

Most database vendors provide optimization hints to enable the user to influence decisions made by the database SQL optimizer as to which execution plan to choose. Oracle provides a full set of optimization hints to help users rectify individual SQL performance problems, making it the most open of all the database platforms. This approach admits the database SQL optimizer cannot guarantee every SQL will perform well.

Upgrading the database SQL optimizer is a risky exercise for database vendors. No matter how good a new version of the database SQL optimizer is, it is going to have some negative impact. For example, if a new version of the database SQL optimizer can fix 50 percent of old SQL statements performance problems, but in the meantime it introduces 5 percent of all new performance problems for existing good SQL statements, mathematically, it is 10 times better than the old version. It should be a good deal to commit to the upgrade. Most systems are already running on an “adopted” status, which means users have accepted what they have, they know which functions are running slow, the database tuner may already have changed the system configuration to address the problems and sometimes even the users’ daily operations are changed to accommodate those slow SQL processes.

If there are any changes after upgrading to a new database version, I think you will agree that a 50 percent improvement will not stop the users from complaining about the 5 percent of new problems. So, that is why database vendors need to provide optimization hints to let users fix problems at the individual SQL statement level and not at the database SQL optimizer global level. This trend is becoming more popular among database vendors. Sybase Adaptive Server and Microsoft SQL Server provide more plan forces (like Oracle optimization hints) in their new versions. IBM DB2 UDB does not provide any optimization hints, but it does provide optimization

classes to control the intelligence levels of execution plans generated by the database SQL optimizer. Unless the database SQL optimizer can guarantee that each execution plan it generates is the best execution plan for each unique database environment, we like the approach that Oracle provides, which gives us the opportunity to help the database SQL optimizer choose the best execution plan with the optimization hints.

How hints work with SQL transformation

Optimization hints are used to guide the database SQL optimizer to select a specific method preferred by the user to process the SQL statement. But sometimes the SQL statement's syntax prevents the database SQL optimizer from using the method specified by the user. In this case, the database SQL optimizer will ignore the instruction given by the user.

Let's look at the examples in Figure 19.

In reviewing the execution plans, the example in Figure 19 shows you that the USE_MERGE hint does not cause the Oracle SQL optimizer to generate a sort merge join for this SQL statement. It is actually quite often that the database SQL optimizer will not follow your instruction due to the limitation of the SQL syntax. For complicated SQL statements, the situation is even more complex as we may not be able to tell whether the hints will be used or how good the result will be if the hint is applied.

Quest SQL Optimizer's approach

This is why Quest SQL Optimizer takes a different approach and does not follow the knowledge-oriented SQL tuning approach. The SQL Transformation Engine will try most of the possible combinations for rewriting the SQL syntax, combined with optimization hints, to explore the potential of a database SQL optimizer.

QUEST SQL OPTIMIZER FORECAST

Is SQL optimization an unsolvable problem?

In computability theory, there is a famous decision problem called the halting problem, which can be informally stated as follows: Given a description of a program and its initial input, this determines whether the program, when executing the input, will ever halt (complete); the alternative is that it runs forever without halting. In 1936, Alan Turing proved that a general algorithm to solve the halting problem for all possible inputs cannot exist. It is said that the halting problem is undecidable over Turing machines.

Perhaps you find the halting problem is similar to one of RDBMS SQL optimization problems. Actually, there are two major problems that modern RDBMS SQL optimizers encounter today.

It is actually quite often that the database SQL optimizer will not follow your instruction due to the limitation of the SQL syntax.

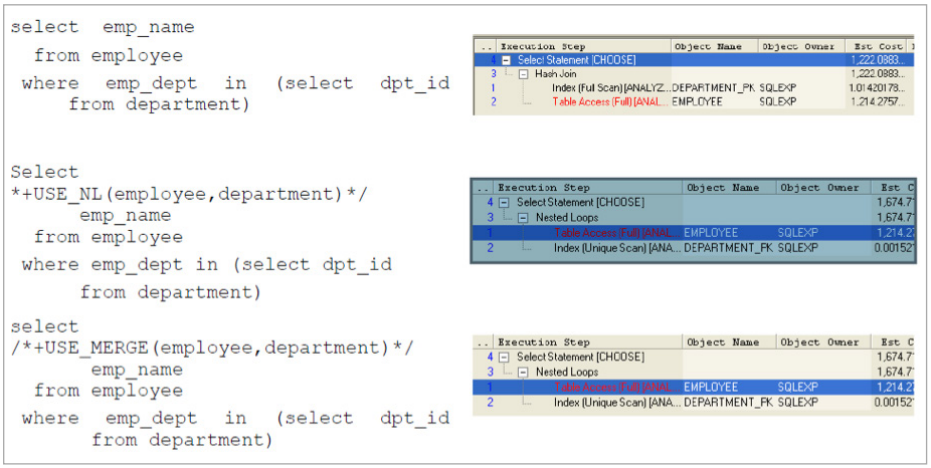


Figure 19.

Quest

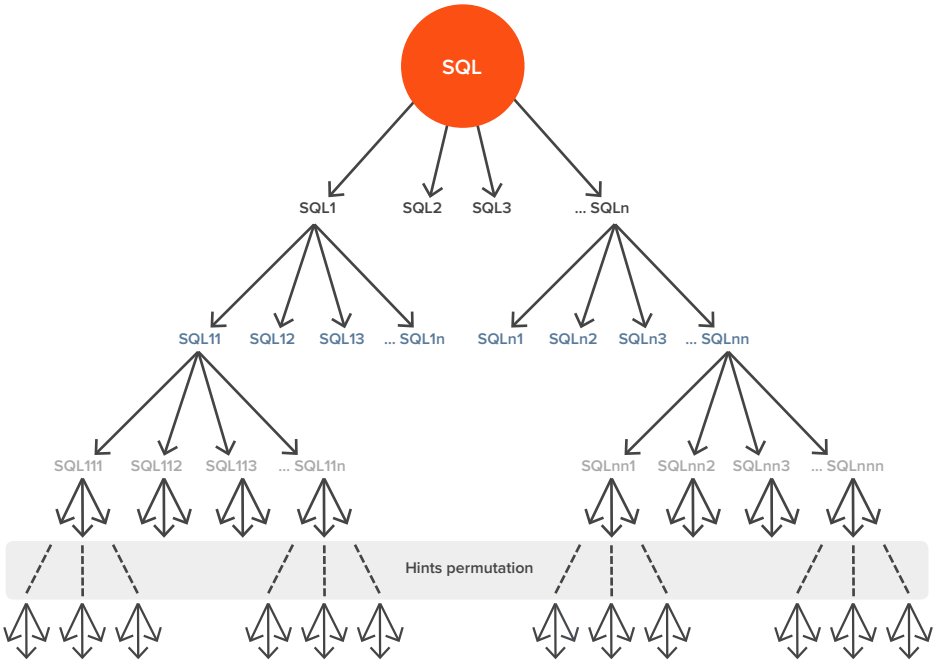


Figure 20. How optimization hints work with recursive SQL transformation

The SQL Transformation Engine will try most of the possible combinations for rewriting the SQL syntax, combined with optimization hints, to explore the potential of a database SQL optimizer.

- **The limited size of the plan space (the number of execution plans can be investigated during SQL optimization)**—Because the database SQL optimizer has to do real-time optimization, it is impossible for the database SQL optimizer to do an exhaustive plan space search (search all possible execution plans internally); otherwise, the optimization time will be much longer than the time it takes to execute the SQL statement even with a bad execution plan.
- **The limited accuracy of the cost estimation algorithm**—After the database SQL optimizer has generated all the internal SQL rewrites and their corresponding execution plans, the database SQL optimizer uses the cost estimation algorithm to choose the theoretical best execution plan, the one with the lowest cost, to execute. We can use tables, indexes, histograms, assumptions and other statistics to estimate the cost of an execution plan. The problem is lot to tell whether the SQL will halt; rather, we are facing a more difficult problem of determining how long a query will run with a specific execution plan. Database vendors have spent a lot of effort in this area, but the fact remains that we still have to tune SQL statements ourselves.

Accurate cost estimation versus plan space

The goal of a good database SQL optimizer is not only to provide accurate cost estimation for SQL statements, but to generate more internal execution steps to compose more execution plans. More internal execution plans mean the database SQL optimizer has a larger plan space during SQL optimization.

The following is an example to show you the relationship between plan space and cost estimation. Consider how you travel to your office and suppose you only have one route to get there. If the only way you go to the office is jammed due to weather or traffic conditions, you probably will not be able to get to your office on time.

Consequently, most people have multiple routes (plan space) in mind. Every morning, based on weather and traffic conditions, they select the best route (cost estimation) to the office. The more routes they have in mind, the higher the possibility they can overcome more complex traffic and weather conditions.

The point is, with more possible execution plans, every morning they will spend more time thinking about which path is the best way to go to their office. As the number of routes increase, the chance they select a non-optimal path gets higher.

This problem is similar to what the database SQL optimizer faces. The accuracy of the database SQL optimizer's cost estimation is opposite to the size of plan space that the database SQL optimizer generates. The bigger the plan space, the easier it is for the optimizer to select a non-optimal execution plan. That is why Oracle's SQL statement performance always has room for improvement, since Oracle has a relative larger plan space.

WHAT ABOUT A SELF-LEARNING SQL OPTIMIZER?

At least two database vendors are trying to build self-learning SQL optimizers. The idea is to use actual statistics from executed SQL statements to rectify the future cost estimation of the same or similar SQL statements. It seems like a good idea, but you will find their self-learning SQL optimizer is either turned off by default or built as an individual tuning advisor. Of course, we cannot say they will not provide a better and fully automatic solution in the future. But the fact is this technology is not mature enough today to be turned on automatically. Furthermore, database SQL optimizers have a lot of problems pending that still need to be solved. They should not just focus on the error of cost estimation without taking care of the small plan space problem.

To be frank, a self-learning database SQL optimizer is still only a dream. Using actual statistics to rectify future cost estimation may solve some problems, but it definitely cannot solve every SQL costing problem. Furthermore, new features will cause new problems. To my understanding, what they are doing is similar to providing a patch to the existing cost estimation problem. It will not fundamentally solve the database SQL optimizer problem.

One possible solution to address the cost and plan space problem is to build a SQL tuning agent with a query base statistics database that offloads the original SQL optimizer from real-time optimization. The agent should be running during nonpeak hours to review all executed SQL statements (or resource-intensive SQL). For each SQL statement, the agent should generate more execution plans than the database SQL optimizer generates, since the real-time SQL optimizer cannot spend much execution time during real-time optimization. For each execution plan it generates, the statistics can be collected by a test run or partial test run of the query. Of course, the database SQL optimizer still faces a lot of problems today that would have to be solved by a SQL tuning agent. But the beauty of a SQL tuning agent is that it has no response time limitation. Any complex estimation or test run algorithm can be built piece by piece in the agent.

THE ROLE OF QUEST SQL OPTIMIZER IN THE FUTURE

As long as the database execution plan space is being enlarged and more SQL optimization controls are provided by database vendors, users will have more room to improve the performance of their SQL and to maximize the power of the database. Quest SQL Optimizer will continue to play an important role by helping users optimize their SQL statements. Furthermore, database vendors are becoming more aware of the limitations of their SQL optimizer's intelligence. New features in the database upgrades are normally coming out faster than internal SQL optimizer upgrades.

Some examples, such as materialized view rewrite, no statistics for user-defined SQL function call by a SQL statement and domain indexes, are more or less out of step with the database upgrade speed. I believe some topics cannot even be solved within the next few years.

This is why most database vendors are willing to provide users with more control to optimize their SQL statements. Some vendors are making it even easier to control their database SQL optimizer

The bigger the plan space, the easier it is for the optimizer to select a non-optimal execution plan.

For Sybase Adaptive Server, we provide abstract plan tuning, in which alternative abstract plans are generated for poorly performing SQL.

without requiring a change to the source code. It is a source-less SQL tuning technique that is very important to package users since they do not own the source code. For example, Oracle provides Stored Outlines and SQL Profile; Sybase Adaptive Server provides Abstract Plan; and Microsoft SQL Server provides Plan Guide in Microsoft SQL Server 2005. You can see that the mainstream database vendors are going in the same direction to help users tune SQL without the need to change the source code. But the problem is that those new features are difficult to use unless you have in-depth knowledge of SQL optimization. I believe many people cannot accurately guide the database SQL optimizer to generate a good execution plan for a complicated SQL statement.

Since Quest SQL Optimizer is an external SQL rewriter that relies only on the feedback of the execution plan from the database SQL optimizer, the Quest SQL Optimizer Engine can generate alternative syntax to influence the database SQL optimizer to pick up a better execution plan for a SQL statement. For Sybase Adaptive Server, we already provide abstract plan tuning, in which alternative abstract plans are generated for poorly performing SQL. Once you are satisfied with a specific abstract plan, you can save it with the SQL text into an Adaptive Server database. The next time the same SQL is received by the Sybase Adaptive Server SQL optimizer, the stored abstract plan will be used to generate the expected execution plan. The beauty of this approach is that users do not need to change their source code: the abstract plan can be changed any time to accommodate database configuration changes or upgrades. Package providers can keep one source to fit different data distributions for different size companies.

Similar technology can be implemented into the Oracle and Microsoft SQL Server platforms in the near future. It is a new generation of source-less SQL tuning tools that enable users to deploy their tuning instruction for specific SQL statements over various database environments.

ABOUT QUEST®

Quest helps our customers reduce tedious administration tasks so they can focus on the innovation necessary for their businesses to grow. Quest solutions are scalable, affordable and simple-to-use, and they deliver unmatched efficiency and productivity. Combined with Quest's invitation to the global community to be a part of its innovation, as well as our firm commitment to ensuring customer satisfaction, Quest will continue to accelerate the delivery of the most comprehensive solutions for Azure cloud management, SaaS, security, workforce mobility and data-driven insight.

If you have any questions regarding your potential use of this material, contact:

Quest Software Inc.
Attn: LEGAL Dept
4 Polaris Way
Aliso Viejo, CA 92656

Refer to our Web site (www.quest.com) for regional and international office information.