Testing relational database query optimization strategies

Vlad Diaconita, Ion Lungu, Iuliana Botha
Department of Economic Informatics and Cybernetics
Academy of Economic Studies, Bucharest, Romania
{vlad.diaconita, ion.lungu, iuliana.botha}@ie.ase.ro

Abstract: Query optimization is one of the most important problems in databases. Query optimization can be talked about in different scenarios with different tools and different results. In this paper we’ll talk about query processing and optimization and test strategies in different context, starting from simple queries continuing with correlated queries, joins with or without different indexing strategies.

Keywords: optimization, testing, indexing, partitioning, relational algebra

1. Introduction

In practice, SQL is the query language that is used in most commercial RDBMSs. Like shown in many works, like [1] or [2], in any RDBS, a query written in SQL goes through the flow pictured in figure 1.

![Figure 1 Query flow in RDBMS](image)

Even if in this paper the tests are done using an Oracle environment, regardless of the RDBMS vendor the process is similar and so are the strategies depicted. First, the scanner finds the query tokens from the text of the query. The parser checks the query syntax if it complies with the syntax rules of the query language then translates it into an internal form, usually a relational calculus expression or something equivalent. The query in valid if the attributes and relation names are consistent with the database schema that is interrogated. An internal representation of the query is made, usually as a tree data structure called a query tree, or using a graph data structure called a query graph. The Database Management System (DBMS) must create an execution strategy plan for retrieving the results of the query from the database files. A query can have many possible execution strategies, and the optimization is the process of choosing a appropriate one.

The query optimizer module has to come up with a good execution plan, and the code generator executes that plan. The runtime database processor has the task of executing the query code, in compiled or in interpreted mode, and generate the results. In the event of a runtime error, a message is being generated by the runtime database processor. Like shown in [1], given a query, there are many different plans that a DBMS can follow to process it and produce the results. All plans produce the same result but can vary in the amount of time that they need to execute. Query optimization is absolutely necessary in a DBMS, the cost difference between two alternatives can be important.

The optimizer doesn’t always choose the best strategy, only a reasonably efficient strategy for executing the query. Finding the best strategy is usually too time-consuming. Also, determining the optimal query execution strategy can require detailed information on how the files are implemented and even on the contents of the files, information that may not be fully available in the DBMS catalogue.

A language like SQL provides the developer with a tool for specifying what the needed results of the query are without the need to specify how these results should be obtained.

A RDBMS must be able to compare query execution strategies and choose a near-optimal strategy. Each DBMS typically has a number of general database access algorithms that implement relational algebra operations such as Selection, Projection or Join or combinations of these operations. Only execution strategies that can be implemented by the DBMS access algorithms and that apply to the particular query, as well as to the particular physical database design, can be considered by the query optimization module.

As noted in [6] there are two separate stages to running a query. First, the optimizer works out what it “thinks” is going to happen, and then the
run-time engine does what it wants to do. In principle, the optimizer should know about, describe, and base its calculations on the activity that the run-time engine is supposed to perform. In practice, the optimizer and the run-time engine sometimes have different ideas about what they should be doing. There are many features available to the optimizer to manipulate the query before optimizing it such as predicate pushing, subquery unnesting and star transformation.

The query optimizer chooses an execution plan for each query block. In Oracle, all_rows goal means that the optimizer will attempt to find an execution plan that returns all the rows in the shortest possible time. A first_rows N goal will try to maximize the plan for returning a part (10,100, 10000) of the total number of rows. The Rule Based Optimizer (RBO) is obsolete starting Oracle 10g. It’s still present but is no longer supported and is present to provide backwards compatibility. The choose option, this gives the optimizer a run-time choice between rule based optimization that is currently obsolete and all_rows.

As shown in [7], Oracle9i Database has an I/O cost model that evaluated everything primarily in terms of single block reads, largely ignoring CPU costs (or using constants to estimate CPU costs). Starting with Oracle 10G the cost model for the optimizer is CPU+I/O, with the cost unit as time.

2. Processing a query

Consider the following statement in extended relational algebra:

\[ \pi_{ord, id_cc}(\sigma_{\text{Comis}>\mu \text{Comis}(\sigma_{\text{broker}='ANDR'(OCA))}}) \]

where:

- \( \mu \) (script \( F \)) is the symbol used for an aggregate function;
- \( \sigma \) is the symbol for the selection operation;
- \( \pi \) is the symbol for the projection operation.

This statement can be translated into SQL like this:

(1) select ord, id_cc from oca where comis > (select avg(comis) from oca where broker='ANDR')

This query retrieves all the orders, regardless of the broker, which have a commission that is greater than the average commission for the ANDR broker. The query includes a nested sub query so it should be decomposed into two blocks. The inner block is:

select avg(comis) from oca where broker='ANDR'

This retrieves the commission for ANDR. The outer query block is:

select * from oca where comis>c

where \( c \) represents the result returned from the inner block.

Like shown in [2], an SQL query is first translated into an equivalent extended relational algebra expression, represented as a query tree data structure which then optimized. SQL queries are split into query blocks, units that can be translated into the algebraic operators and then optimized. A query block contains a single SELECT-FROM-WHERE expression, as well as GROUP BY and HAVING clauses if these are part of the block. If they exist, nested queries within a query are identified as separate query blocks. Because SQL includes aggregate functions, such as MAX, MIN, SUM or COUNT, these functions must also be included in the extended algebra, like operators.

In this example, the inner block needs to be evaluated once to produce the average commission of the orders by the clients of a certain broker, which is then used by the outer block to return the needed results. This is also called a non-correlated nested. The execution plan for query (1) is the one in listing 1:

```
SELECT STATEMENT, GOAL = ALL_ROWS
Cost=2812 Cardinality=17927 Bytes=699153
TABLE ACCESS FULL owner=FORT
Object name=OCA Cost=1406 Cardinality=17927 Bytes=699153
SORT AGGREGATE Cardinality=1 Bytes=18
TABLE ACCESS FULL owner=FORT
Object name=OCA Cost=1407 Cardinality=5192 Bytes=93456
```

Listing 1 Non-correlated nested

So, the cost of this select is 2812, the estimated cardinality is 17923 and the real number of returned records being 93367.

The cost represents the optimizer’s best estimate of the time it will take to execute the statement, but this cost can be wrong because of many reasons, such as some wrong assumptions exist in the cost model or relevant statistics is missing or present but misleading.

With older databases it used to be much more costly to run correlated nested queries, where a tuple variable from the outer query block appears in the WHERE-clause of the inner query block:
In this query we search for the orders with the commission greater than the average for each broker. In this case, the inner query must be run for each tuple from the outer query. The execution plan for query (2) is the one in listing 2:

**SELECT STATEMENT, GOAL = ALL_ROWS**

Cost=4320 Cardinality=17927 Bytes=1111474

**HASH JOIN** Cost=4320 Cardinality=17927 Bytes=1111474

**VIEW** Object owner=SYS Object name=VW_SQ_1 Cost=1440 Cardinality=358530 Bytes=6453540

**HASH GROUP BY** Cost=1440 Cardinality=358530 Bytes=8246190

**TABLE ACCESS FULL** Object owner=FORT Object name=OCA Cost=1405 Cardinality=358530 Bytes=8246190

**TABLE ACCESS FULL** Object owner=FORT Object name=OCA Cost=1405 Cardinality=358530 Bytes=15775320

Listing 2 Correlated nested without indexes

These tests are done on copies of the original production environment tables and have no indexes or statistics at first. The cost of this select is 4320, which is 53% greater than in the first example given that the estimated cardinality being the same. The real number of records returned by this query is 108429.

3. **Optimizing strategies**

There are many algorithms for executing a SELECT operation, which is basically a search operation to locate the records in a disk file that satisfy a certain condition. Some of the search algorithms depend on the file having specific access paths, and they may apply only to certain types of selection conditions. Many rules for different query types (simple selection, different types of joins, multiple queries etc) and different environments (databases, data warehouses, virtual data warehouses, data marts etc) have been identified, discussed and tested in numerous works, such as [1], [2], [3] or [4].

A number of search algorithms are possible for selecting records from a file. These are also known as file scans, because they scan the records of a file to search for the records that comply with a condition. If the search uses an index, it is called an index scan. A more advanced index, a clustering one can be used to retrieve multiple records if the selection condition involves an equality comparison on a non key attribute with a clustering index. A cluster is a method for storing more than one table on the same block. Usually, a block contains data for exactly one table. In a cluster, there can be many tables sharing the same block.

If a condition of a SELECT operation is a conjunctive condition constructed of several simple conditions connected with the AND logical operator such as the forth method, the DBMS can use additional methods to implement the operation such as conjunctive selection that uses an individual index, a composite one or an intersection of record pointers.

When the selection has a single condition, the database management system will check if an access path exists on the attribute involved in that condition. If an access path (such as index or hash key or sorted file) is available, the method corresponding to that access path is used. If this is not the case, the brute force, linear search approach of method can be used. Query optimization for a SELECT operation is required mostly for conjunctive select conditions whenever more than one of the attributes involved in the conditions have an access path. The optimizer should choose the access path that retrieves the fewest records in the most efficient way by estimating the different costs and choosing the method with the least estimated cost.

When the optimizer has to choose from multiple simple conditions in a conjunctive select condition, it typically considers the selectivity of each condition. As shown in [1], selectivity is the ratio of the number of records (tuples) that satisfy the condition to the total number of records (tuples) in the file (table/relation). So, selectivity is a number between zero and one. At the extremes, zero selectivity means none of the records in the file satisfies the selection condition, and a selectivity of one means that all the records in the file satisfy the condition. Compared to a conjunctive selection condition, a disjunctive condition, where simple conditions are connected by the OR logical connective, is much harder to process and optimize. For example, given this statement:  \( \sigma_{\text{broker}='\text{ANDR}' \text{ OR Comis} > 8 \text{ OR Sym}='\text{EBS}'(\text{OCA})}. \) In this expression not much optimization can be made, because the records
satisfying the disjunctive condition are made from the union of the records satisfying the three conditions. So, if any one of the conditions does not have an access path, only the brute force, linear search approach can be used. If an access path exists on every simple condition in the disjunction, the selection can be optimized by retrieving the records satisfying each condition and then applying the union operation to eliminate duplicates. The execution plan for the above statement, if indexes exist on each column involved is the following uses the three indexes and has a cost of 1143 and an estimated cardinality of 6913:

A DBMS can use many of the methods discussed above, and some additional ones. The query optimizer must choose the appropriate one for executing each SELECT operation in a query which has the lowest estimated cost.

The JOIN operation is one of the most time-consuming operations in query processing.

A natural join is a type of equijoin where the join predicate arises implicitly by comparing all columns in both tables that have the same column names in the joined tables. The resulting joined table contains only one column for each pair of equally named columns.

There are many possible ways to implement a two-way join, which is a join on two files. Joins involving more than two files are called multi way joins. The number of possible ways to execute multi way joins grows very rapidly.

The nested-loop join is the basic “brute force” algorithm, useful when small subsets of data are being joined, as it does not require any special access paths on either file in the join. For each record in a table, retrieve every record another table and test whether the two records satisfy the join condition. A sort merge join can be used if an index exists on either one or both join attributes the join can be greatly improved by retrieving directly the matching records.

A partition-hash join implies that the records of the tables involved are partitioned in smaller parts. The partitioning of each table is done using the same hashing function on the join attributes. In the partitioning phase, a single pass through the table with fewer records hashes its records to the various partitions of the table. The collection of records with the same value of the hash function is at the same partition, which is a hash bucket in a hash. In the probing phase, a single pass through the other

Indexes exist to help speed up queries. Having the proper columns indexed can reduce the logical I/Os for queries. However, creating an index to make one query run faster may not be a good solution. There are costs associated with data changes when the indexes are involved and the maintenance requirements should be considered. The performance gains of adding the index should be more than the cost of maintaining the index. If we build too many indexes it can lead to performance issues, instead of solving problems. Indexes should be used selectively and their usage monitored.

Indexes are definitely a useful tool for improving access to the data in the database. Several types of indexes are available on both database platforms. Understanding which type of index is being used and how to improve that index will help in performance tuning. Knowing how the various index types affect data changes and improve SELECT statements will help you to decide if the benefits of the index outweigh the costs for putting it in place.

At first, a new index can lead to inexplicable results. For example, we add a primary key on the OCA table which shouldn’t affect the (2) query:

```
alter table oca add constraint oca_pk primary key(id_u,ord);
```

The cost and the cardinality for query (1) remains the same from listing 1 but the new execution plan for query (2) shows a cost of 39207 and a cardinality of 149469489 (!?) compared to 4320 and 17927 even if the execution plan remains the same.

We force Oracle to gather statistics using the following:

```
BEGIN
DBMS_STATS.CREATE_STAT_TABLE ('fort','saves');
DBMS_STATS.GATHER_TABLE_STATS
```

INDEXES
After this, the cost becomes 2884 and the cardinality 165157, much closer to the truth. The oldest and most popular type of Oracle indexing is a standard b-tree index, which excels at servicing simple queries. The b-tree index was introduced in the earliest releases of Oracle and remains widely used with Oracle.

While b-tree indexes can speed up simple queries, they are not a very good choice for low-cardinality columns that have fewer than 200 distinct values. They do not have the selectivity required in order to benefit from standard b-tree index structures. Also, there is no support for SQL functions. The B-tree indexes are not able to speed SQL queries using Oracle's built-in functions. Oracle provides a variety of built-in functions that allow SQL statements to query on a piece of an indexed column or on any one of a number of transformations against the indexed column.

An index-organized table (IOT) has a storage organization that is a variant of a primary B-tree. Unlike an ordinary (heap-organized) table whose data is stored as an unordered collection (heap), data for an index-organized table is stored in a B-tree index structure in a primary key sorted manner [10]. Each leaf block in the index structure stores both the key and non-key columns. Index-organized tables are ideal for OLTP applications, which require fast primary key access and high availability. Organizing a table like this would make it fast when using the primary key for the joins or using just the primary key as the part of the WHERE clause. If the table is normally searched by other columns than how the table is organized, creating an IOT might not be the solution.

We create an index organized table OCA2, a copy of OCA. After gathering statistics like shown above, we run the query (2) which doesn't any part of the primary key. The cost becomes 4840 instead of 2884. We run the following query on both the original table and the IOT one:

```sql
(3) select t.* from oca2 t, journal_executii j
    where t.id_u = j.id_u and t.ord = j.ord
```

The cost on the original query is 4300 and on the IOT is 5000 so in this case this type of index doesn't perform well.

**Bitmap indexes** aren't the same with the standard b-tree indexes and they are stored differently. Instead of storing the row ID, a bitmap for each key is used. Each column means a different value in the bitmapped index. This two-dimensional array represents each value within the index for each rows of the table. Because of this, these indexes are typically smaller in size and are useful for columns that have a low cardinality (few distinct values) and for read-only tables. They might be more expensive than other types of indexes for tables in which the data changes. Building multiple bitmapped indexes can provide fast answers for difficult SQL queries.

For the OCA table, column *status* can have only 9 different values. If we run this statement: \( \sigma_{\text{status} = \text{A}}(\text{OCA}) \) without any index, the estimated cost is 1409. If we build a bitmap index and then gather table stats, the CBO doesn’t use it for this query. The cost with full table scan is 1411. If we force the usage of the index with a hint the cost goes up to 5648:

```sql
select /*+ index(oca OCA_STATUS) */
* from oca where status = 'A'
```

**Bitmap join indexes** store the join of two tables. This type of index is useful in a data warehousing environment and with a star data model schema, because it will index the smaller table information on the larger fact table. The row IDs are stored for the corresponding row ID of the joined table.

**Reverse key indexes** can spread out index blocks for a sequenced column. By using a sequence, there can be many records that all start with the same number. Reversing the numbers will allow for the index to have different beginning values and use different blocks in the index b-tree structure. When loading data, the reverse index will minimize the concurrency on the index blocks.

One of the most important features in Oracle indexing is the introduction of **function-based indexing**. Function-based indexes allow creation of indexes on expressions, built-in functions, and user-written functions in PL/SQL or Java. Oracle's function-based index type can dramatically reduce query time. The function-based index can be a composite index with other columns included in the index. The function that is used needs to match what is being used in the WHERE clause.

Partitioning is a useful way to tune a large database environment. Oracle offers options for partitioning table, such as LIST, HASH, RANGE, and COMPOSITE. The partition key is how the table is partitioned and **partitioned indexes** can be created for these tables. The index can be a local
partitioned index based on the partition key and set up for each partition. The main objective of the partitioning technique is to radically decrease the amount of disk activity and to limit the amount of data to be examined or operated on and to enable parallel execution required to perform queries. Tables are partitioning using a partitioning key that is a set of columns which will determine by their conditions in which partition a given row will be store.

We build a copy of the OCA table, partitioned by range of data_i. Any query that involves this column has a much lower cost. For example:

A. \( \sigma_{data_i < '01/07/2007'}(OCA) \)

B. \( \sigma_{data_i < '01/07/2007'}(OCA \bowtie ord=ord TRADES) \)

We run these queries on the original table and on the partitioned one, the results being shown in table 1.

Table 1 Testing results

<table>
<thead>
<tr>
<th></th>
<th>Original</th>
<th>Partitioned</th>
</tr>
</thead>
<tbody>
<tr>
<td>A Query</td>
<td>4339</td>
<td>4373</td>
</tr>
<tr>
<td>B Query</td>
<td>2744</td>
<td>1851</td>
</tr>
</tbody>
</table>

In this case, the results are as expected. The cost is significant lower when the column on which the table is partition is used and a bit higher when it isn’t used.

Also, there can be use sub partitioning techniques in which the table in first partitioned by range/list/hash and then each partition is divided in sub partitions such as: composite range-hash partitioning and composite range-list partitioning. IOT can be partitioned by range, list, or hash.

4. Conclusions

We can say that tuning and optimizing is a process that is handled by the DBMS, the database administrator and the developers. The DBMS can choose an execution plan according to his rules but also provides different tools that can gather statistics and suggest changes (different execution plans, rewriting queries to take advantage of the indexes, new indexes) that can be implemented by the administrator. Developers can also test and override the default execution plans by using hints and also compare the performance of different indexing strategies.

Acknowledgment: This paper presents some results of the research project PN II, TE Program, Code 332: “Informatics Solutions for decision making support in the uncertain and unpredictable environments in order to integrate them within a Grid network”, financed within the framework of People research program.

References
[2] Yannis E. Ioannidis, Query Optimization, Computer Sciences Department, University of Wisconsin