Reducing Run-time Execution in Query Optimization

Rashmi Singh  
Galgotias College of Engineering and Technology  
Greater Noida

Somesh Sharma  
Galgotias College of Engineering and Technology  
Greater Noida

Saurabh Singh  
Galgotias College of Engineering and Technology  
Greater Noida

Bhawna Singh  
Galgotias College of Engineering and Technology  
Greater Noida

ABSTRACT
The main objective of query processor is to generate the most efficient query results. Using an apt execution plan, query minimizes cost of execution for results. The order of accessing a source table is very important during query execution. The best execution plan from possible ones is presented by Query optimizer. The paper discusses various stages of query optimization using execution plan. It gives the analysis of indexes, type of expressions & joins used in the execution plan of the query. The approach gets the estimate of the cost of query joins in a query at compile time. These estimates help in the construction of a query plan at compile time and then executed at run-time.

Keywords
Execution plan, query optimization, compile time, run time, joins.

1. INTRODUCTION
In today’s object oriented programming languages, need of optimization of query is increasing. Introduction of various processing methods and constructs are being analysed. These new methods have also shown great results in increasing the comprehensibility of programs and the ability of programmers. In SQL, different execution plans are present to optimize the queries. In other programming languages, the constructs allows queries to be written concisely and concretely. The advantages of using query constructs in place of typical representation are well justified. The queries written using other APIs are less clear and concrete than explicit queries. This method provides an avenue for developers to be more productive and work at a higher level of abstraction.

In query processing, there are well described as stages. Firstly, the query processor accepts SQL syntax, secondly selects a plan for executing the syntax, and then executes the chosen plan. Following diagram shows functioning of query processor [1][2][5].

![Functioning of query processor](image)

1.1 Parsing and Semantic Checking
For processing SQL query, the database server employs parser to parse the query to perform syntactic and semantic checking of the query. It is done by transforming the query into a parse tree which is the algebraic representation of the query [6][7][8].

1.2 Query Rewrite
This step involves only complex query which involves joins, predicates, grouping, etc. In this phase query is presented in the form of an annotated parse tree. Query experiences iterative transformations in accordance to the heuristics used. After that, other steps like joining elimination, predicating normalization, selecting operation are performed and at last, project operation and many transformation rules are performed [9].

1.3 Query Pre-optimization Phase
This stage involves pre-optimization phase and enumeration phase. In pre-optimization phase, all predicates, indexes, joins used in the access plan are studied by query analyse. Furthermore, include designing alternative plans for the query. In Enumeration phase, various possibilities of the access plans for the query are collected. The most favourable access plan is chosen utilizing join algorithm, cost estimation method, access method, and selectivity.

1.3.1 Join Selection

Join Selection is third in a series of steps for query optimization. In case of the query that is multi-table query or a self- join, the query optimizer analyses join selection and selects low cost join strategy. It evaluates the cost by using a number of factors, expected number of reads and the amount of memory required. The choice can be made between three basic strategies for processing joins: nested loop joins, merge joins, and hash joins.

- Nested Loop Joins
  Nested Loop Joins are mainly for smaller tables. Method of iteratively scanning the rows of one table and matching with a second table with the help of index is called nested loop join. Hash join is used in absence of nested join. There are 3 types of nested Join. The search scanning an entire table or index is called a naive nested loop join. The search using an index, called an index nested loops join. The index is made as part of the query plan, called a temporary index nested loops join. In temporary index nested loop join, the index is destroyed after execution of the query.

- Merge Joins
  Being more efficient for large tables with the key columns sorted, merge join is used in sorting of two inputs on the join column. In case of the inputs are already sorted then less I/O is required to process when the join used is one to many. Many merge join make use of a temporary table to store rows instead of discarding them. One of the inputs must rewind to start of the duplicates when duplicate values from each input. It is a fast joining method, but it can be expensive if sort operation is required.

- Hash Joins
  Hashing matched a particular data item with an already present value by division of the existing data into groups on the basis of similar property. Hash Bucket has the data with the same value. Hash bucket checks with existing data to find a match. The smaller input is taken as build input during hash joining. The storage in buckets, also known as hash tables, takes place in the form of linked lists, in which each entry contains only columns from building input that are needed. Hash joins are comparatively effective in set- matching operations such as intersection, union, semi-join, full outer join. The set of columns in the equality predicate is called hash key which contributes in the hash function. We have used Adventure Works database. From database we have used Sales.Customer and Sales.SalesOrderHeader tables for demonstrating hash join.

1.3.2 Index Selection

Association of table with an index improves the retrieval speed of rows from table [9-11]. There are two types of indexes: Clustered index and Non-Clustered index. Clustered Indexes simply sort and stores the data rows in the table centred on the column selected as the key. When there is a clustered index in the table, the table is known as clustered table. A table with no clustered index is called as a heap suggesting an unordered structure. Consider the table customer and the following query which creates an index on Cust_id of Customer table.

Non-clustered index comprises the non-clustered index key values and each key value entry has a pointer to the data row that contains the key value. The following query creates non-clustered index on Cust_Category of Customer table.

The non-clustered column has depended on the Clustered column in the database. The Cust_category column with distinct values will store the clustered index columns values along with it.

1.3.3 Plan Selection

It is the fourth step in query optimization. It is decided by the cost of a given plan, in terms of required CPU preprocessing and I/O, and query execution time. Hence it is known as Cost- Based plan.

The optimizer’s work involves the generation and evaluation of many plans and choosing the lowest cost plan. It solely depended on the plan as it will execute the query relatively fast, using the least amount of resources, CPU and I/O. The optimizer may take a less efficient plan in case of it will take more time to evaluate many plans in comparison to running a less efficient plan. For example, a simple query having a single table with no indexes and no aggregates or calculations is present in these the query, then the optimizer will use single, trivial plan for these types of queries to save time that will wasted in choosing an optimal plan. Otherwise, the optimizer will do cost based calculation to plan selection in case of non-trivial query. For carrying out this, optimizer depends upon on statistics of the execution plan [11].

1.4 Detailed Plan Execution

This phase takes the top plan and made the graphical view of the query. Graphical view is available in Interactive SQL. Graphical View has a tree like structure where each node is a physical operator implementing specific relational algebra operation.

1.5 Plan Execution

Best execution plans built in previous phase used in the computation of the result of the query.

The input to the optimizer involves the query, the database schema (table and index definitions), and the database statistics. The execution plan is the output of the query optimizer. Query optimizer based on the following components determines the execution plan of the optimizer.

- Access Method: decides how to access the data. It can be Table Scan or Index Scan etc.
- Join Method: Decides how to join tables to each other. It can be a hash join or merge join etc.
- Order of Join: decides the sequence of joining table.
- Cardinality: Depending upon the number of rows, selection of the access method, or join algorithm.
• Join Type: Execution plan also reviews the join type such as anti-join, semi-join etc.

• Partition Pruning: Query optimizer checks the partitions required for obtaining the query result.

• Parallel Execution: Query Optimizer checks each operation on the plan which is conducted in parallel or not, the data redistribution method used is right or not.

During dealing with query optimization, testing the execution plan for different execution strategies is a must. The selection of the best execution plan involves various factors such as selection of join algorithm, the use of indexes, the order of executing relational algebra operators etc.

2. QUERY OPTIMIZER ARCHITECTURE

In this part, a brief outlook on query optimization process in a DBMS has been provided. The execution plans can be decided for a database and a query, which can be used to answer the query. Analyses of all the plans need to be made for selecting the one with best estimated performance. This architecture can allow one to make optimizer, in real systems. Query optimization process has two stages: rewriting and planning. There is only one module in the first stage, the Rewrite and remaining modules are in the second stage.

2.1 Module Functionality

2.1.1 Rewriter

In this module, given query gets transformed and resulting in equivalent queries that can be more efficient, e.g., replacing views with their respective definition, attuning out of nested queries, etc. The Rewriter transforms query, taking account of the declarative, i.e., static, characteristics of queries without amounting for the actual query costs for the specific DBMS and database concerned. The original query is not processed given the rewriting is known or assumed to always be beneficial; or else, it is forwarded to the next stage. This stage functions at the declarative level of understanding nature of the rewriting transformations.

2.1.2 Planner

Being a main module of the ordering stage, it chooses the best execution plan, to be used to generate the answer of the original query, by analysing all other plans which are generated in the previous stage. It utilizes a search strategy; examining the space of execution plans in a descriptive way. The Algebraic Space and the Method-Structure Space are two other modules to decide the space. Additionally, these two modules and the search strategy also optimize the cost, i.e., running time. The Planner compares various execution plans on the basis of their estimated cost so that the cheapest may be chosen. These costs are decided upon by the last two modules of the optimizer, the Cost Model and the Size-Distribution Estimator.

2.1.3 Algebraic Space

This module decides execution action orders that are to be considered by the Planner. The actions present same result, differing on one parameter, i.e. performance and are represented in relational algebra as formulas or in tree form. Given the involvement of the algorithmic side of the objects produced in module and eventually forwarded to the Planner, the whole planning stage is seen as operating at the procedural level.

2.1.4 Method Structure Space

This module defines the choices made to implement the processing of each ordered series of actions decided by the Algebraic Space. This choice is made in accordance with the present join methods for each join (e.g., nested loops, merge scan, and hash join), if supported data structures are built on the y, if/when duplicates are eliminated, and other implementation characteristics of this sort, which are predetermined by the DBMS implementation. This choice is decided by available indices for involving each relation, which is controlled by the physical representation of each database present in its catalogues. This module generates complete execution plans using an algebraic formula or tree from the Algebraic Space, which specify the implementation of each algebraic operator and the use of any indices.

2.1.5 Cost Model

This module presents the arithmetic formulas that are deployed for cost – estimation of execution plans. Each different join method, each different index type access, and in general each distinct kind of step have a cost finding formula. Most of these formulas are simply approximations of what the system actually does and are based on certain assumptions regarding issues like buffer management, disk-CPU overlap, sequential vs. random I/O, etc. The essential input parameters to a formula are the size of the buffer pool used by the corresponding step, the sizes of relations or indices accessed, and possibly various distributions of values in these relations. First one is decided by the DBMS for each query; the other two are estimated by the Size-Distribution Estimator.

2.1.6 Size Distribution Estimator

This module determines the sizes (and possibly frequency distributions of attribute values) of database relations and estimation of indices as well as (sub) query results. The Cost Model needs these estimates. The module adopts an estimation approach to determine the form of statistics for maintaining catalogs of each database, if any.

2.2 Plan Guide

Plan guides are used for performance optimization of queries when we may not want to change the text of the query directly. Plan guides decide optimization of queries by attaching query hints or a fixed query plan to them. Plan guides can be created to match queries that are executed in the following contexts:

An OBJECT plan guide matches queries that are executed in accordance of Transact-SQL stored procedures, user-defined scalar functions, multi-statement user-defined table-valued functions, and DML triggers.

An SQL plan guide matches queries executed in the context of stand-alone Transact-SQL statements and batches that are not part of a database object. SQL-based plan guides can also be used to match queries that parameterize to a specified form.

A TEMPLATE plan guide matches stand-alone queries that parameterize to a specified form. These plan guides are used to override the current PARAMETERIZATION database SET option of a database for a class of queries.

Plan guide uses any combination of valid query hints. The OPTION clause given in the hints clause is added to the query before it compiles and optimizes during the matching of query in
plan guide. In a few cases, a query that was matched to a plan
guide already has an OPTION clause; the plan guide’s query
hints replace those in the query. For a plan guide to match a
query that already has an OPTION clause, you must include the
OPTION clause which specifies the query’s text to match in the
sp_create_plan_guide statement. If the hints that are specified in
the plan guide to be added to the hints that is already present
in the query, then instead of replacing them, both the original
hints and the additional hints are specified in the OPTION
clause of the plan guide.

Some plan guides apply a fixed query plan when aware of an
existing execution plan to perform better than the one selected
by the optimizer for that query. Taking note of the fact that
fixing a plan to a query suggests that adapting the plan by the
query optimizer cannot longer change in statistics and index
of query. When considering plan guides using fixed query plans,
the advantages of applying a fixed plan with the inability to
adapt the plan automatically as data distribution and available
indexes change are compared.

3. ADVANCED TYPE OF OPTIMIZATION
In this section, a brief description of advanced types of
optimization is provided that have been given over the past few
years [12]. A lot of interesting work has been done on them, e.g.,
nested query optimization, rule-based query optimization, query
optimizer generators, object-oriented query optimization,
optimization with materialized views, heterogeneous query
optimization, recursive query optimization, aggregate query
optimization, optimization with the expensive selection
predicates, and query optimizer validation [13].

3.1 Semantic Query Optimization
As a form of optimization, Semantic query optimization is
mostly related to the Rewriter module.
The core theme deals with usage of integrity constraints present
in the database for rewriting a given query into semantically
equivalent ones. Further, Planner optimizes them as regular
queries and generated the efficient plan to solve the original
query. Stating an example, using a hypothetical SQL–like syntax,
consider the following integrity constraint:
assert sal-constraint on emp:

\[
\text{sal} > 100K \text{ where } \text{job} = \text{‘Sr. Programmer’}
\]

Also consider the following query:

select name, oor from emp, dept
where emp.dno = dept.dno and job = ‘Sr. Programmer’.

Using the above integrity constraint, the query can be
rewritten into a semantically equivalent one to include a
selection on sal:

select name, oor from emp, dept
where emp.dno = dept.dno and job = ‘Sr. Programmer’ and
sal > 100K.
The extra selection can assist greatly to find a better plan to
answer the query, even during the case of only index in the
database being a B+-tree on emp.sal. On the other side, it would make of no use if required index is absent. Owing
to all these issues, all proposals for semantic query optimization
present and deciding various heuristics or rules making
rewritings beneficial and should be applied.

3.2 Global Query Optimization

Till now, optimizing individual queries were discussed. Still,
many a times during optimization, multiple queries are present
simultaneously like queries with unions, queries from multiple
concurrent users, queries embedded in a single program, or
queries in a deductive system. Optimizing each query alone is a
heavy task, one have to devise a global plan to solve this,
although possibly suboptimal for each individual query, is ideal
for the execution of all of them as a group. Numerous

techniques have been mentioned for global query optimization.
As a simple example of the problem of global optimization
consider the following two queries:

select name, oor from emp, dept
where emp.dno = dept.dno and job = ‘Sr. Programmer’;

select name from emp, dept
where emp.dno = dept.dno and budget > 1M.

Depending on the sizes of the emp and dept relations and the
selectivities of the selections, it will be favourable that
computations of the entire join once and processing separately
the two selections to get the results of the two queries is more
efficient than performing the join twice by taking the time of
the corresponding selection in consideration. Planner modules
are so developed that can analyse all the available global
plans and give the optimal one, ultimately it is the objective
of global/multiple query optimizers.

3.3 Dynamic/Parametric Query Optimization
As mentioned earlier, optimization of embedded takes place
once at compile time and are executed multiple times at run

time. Owing to the temporal separation between optimization
and execution, various parameter’s value may be very different
during execution from those during the optimization time. This
may cause the invalidation of the chosen plan (e.g., if indices
used in the plan are no longer available) or simply making it
optimal. To resolve this issue, many techniques [14,15,16]
have been mentioned that utilize search strategies (e.g.,
randomized algorithms or the strategy of volcano) to optimize
queries along with taking care of all likely values that
corresponded parameters have during runtime. These techniques
deploy usage of the true parameter values at runtime, and then
take the optimal plan with little or no overhead. Of a drastically
different case, plan switching may take place during query
execution through our technique of RdB/VMS [17], whereby
dynamically monitoring how the probability distribution of plan
costs changes.

4. EXPERIMENTAL EVALUATION
In this section, the performance of the query proposed is
evaluated. Several experiments were conducted over certain
queries and joins. Following were the observations:

A query can be optimized using HINT query by reducing the
cost of performing the join operation. For example,

SELECT OC.Customer, OA.AddressID FROM
SalesLT.Customer AS OC JOIN SalesLT.CustomerAddress AS
OA ON OC.CustomerID=OA.CustomerID;
After using HINT query for the join (Merge Join),
```
SELECT OC.Customer, OA.AddressID FROM SalesLT.Customer AS OC JOIN SalesLT.CustomerAddress AS OA ON OC.CustomerID=OA.CustomerID
OPTION (MERGE JOIN);
```

This shows that the cost incurred in using other type of join method using HINT query is reduced, thereby reducing the time taken for execution of the query at runtime.

SQL Server evaluates some constant expressions early to improve query performance, referred to as foldable expressions. This includes:
- Arithmetic expressions
- Logical expressions, that contain only constants.
- Built-in functions that are considered foldable by SQL Server. Generally, an intrinsic function is foldable if it is a function of its inputs only and not other contextual information, such as SET options, language settings, database options, and encryption keys.

5. CONCLUSION
Given the declarative nature of SQL, there are numerous alternative ways with performance level to execute a given query. When query is submitted to the database, evaluation of different possible plans for executing the query is done by optimizer and generates the best alternative.

In Query Analysis phase, optimizer will find search type of expression and Joins. In Index Selection phase, optimizer chooses the best index to use for each table. In join reordering phase, optimizer computes cost. In plan selection, optimizer will select the best plan suitable for a given query.

So while processing the query, a cost based query optimizer in SQL server gives the effective access path to reduce the total time of execution of the query during runtime.

6. REFERENCES


