January 2014

NEW DYNAMIC QUERY OPTIMIZATION TECHNIQUE IN RELATIONAL DATABASE MANAGEMENT SYSTEMS

N. SATYANARAYANA

Dept CSE/IT LITS Khammam, A. P INDIA., satyauce234@gmail.com

SK. SHARFUDDIN

Dept CSE/IT LITS Khammam, A. P INDIA, shariff_it@yahoo.com

SK.JAN BHASHA

Dept CSE/IT LITS Khammam, A. P INDIA., afreen.jbasha@gmail.com

Follow this and additional works at: <https://www.interscience.in/ijcns>

Part of the [Computer Engineering Commons](#), and the [Systems and Communications Commons](#)

Recommended Citation

SATYANARAYANA, N.; SHARFUDDIN, SK.; and BHASHA, SK.JAN (2014) "NEW DYNAMIC QUERY OPTIMIZATION TECHNIQUE IN RELATIONAL DATABASE MANAGEMENT SYSTEMS," *International Journal of Communication Networks and Security*. Vol. 2 : Iss. 3 , Article 9.

DOI: 10.47893/IJCNS.2014.1095

Available at: <https://www.interscience.in/ijcns/vol2/iss3/9>

This Article is brought to you for free and open access by the Interscience Journals at Interscience Research Network. It has been accepted for inclusion in International Journal of Communication Networks and Security by an authorized editor of Interscience Research Network. For more information, please contact sritampatnaik@gmail.com.

NEW DYNAMIC QUERY OPTIMIZATION TECHNIQUE IN RELATIONAL DATABASE MANAGEMENT SYSTEMS

N. SATYANARAYANA¹, SK.SHARFUDDIN², SK.JAN BHASHA³

^{1,2,3}M.Tech, Dept CSE/IT LITS Khammam, A. P INDIA.

E-mail: satyauce234@gmail.com, shariff_it@yahoo.com, afreen.jbasha@gmail.com

Abstract- Query optimizer is an important component in the architecture of relational data base management system. This component is responsible for translating user submitted query into an efficient query evolution program which can be executed against the database. The present query evolution existing algorithm tries to find the best possible plan to execute a query with a minimum amount of time using mostly semi accurate statistical information (e.g. sizes of temporary relations, selectivity factors, and availability of resources). It is a static approach for generating optimal or close to optimal execution plan. Which in turn increases the execution cost of the query to reduce the execution cost of the query; I propose a new dynamic query optimization algorithm which is based on greedy dynamic programming algorithm uses randomized strategies and reduces the execution cost of the queries and system resources and also it works efficiently with distributed and centralized databases.

General Terms- Database, relation, resources, algorithm, selectivity factor, join Conditions, etc...

Keywords- Query optimizer, relational database, static query optimization, dynamic query optimization.

1. INTRODUCTION

The query optimizer is the component of a database management system that attempts to determine the most efficient way to execute a query. The optimizer considers the possible query plans for a given input query, and attempts to determine which of those plans will be the most efficient. Cost-based query optimizers assign an estimated "cost" to each possible query plan, and choose the plan with the smallest cost. Costs are used to estimate the runtime cost of evaluating the query, in terms of the number of I/O operations required, the CPU requirements, and other factors determined from the data dictionary. The set of query plans examined is formed by examining the possible access paths (e.g. index scan, sequential scan) and join algorithms (e.g. sort-merge join, hash join, nested loop join). The search space can become quite large depending on the complexity of the SQL query.

Generally, the query optimizer cannot be accessed directly by users: once queries are submitted to database server, and parsed by the parser, they are then passed to the query optimizer where optimization occurs. However, some database engines allow guiding the query optimizer with hints. Not every aspect of SQL execution can be optimally planned ahead of time. Oracle thus makes dynamic adjustments to its query-processing strategies based on the current database workload. The goal of dynamic optimizations is to achieve optimal performance even when each query may not be able to obtain the ideal amount of CPU or memory resources. Three steps are involved for query processing: decomposition, optimization and

execution. The first step decomposes a relational query using logical schema into an algebraic query. During this step syntactic, semantic and authorization are done. The second step is responsible for generating an efficient execution plan for the given SQL query from the considered search space. The third step consists in implementing the efficient execution plan. Most of the DBMSs have used the static optimization approach which consists of generating an optimal (or close to the optimal) execution plan, then executing it until the termination. All the methods, using this approach, suppose that the values of the parameters used (e.g. sizes of temporary relations, selectivity factors, availability of resources) to generate the execution plan are always valid during its execution. However, this hypothesis is often unwarranted. Indeed, the values of these parameters can become invalid during the execution due to several causes.

1.1 Estimation errors: the estimation on the sizes of the temporary relations and the relational operator costs of an execution plan can be erroneous because of the absence, the obsolescence, and the inaccuracy of the statistics describing the data, or the errors on the hypotheses made by the cost model. For instance, the dependence or the independence between the attributes member of a selective clause (e.g. town='Paris' and country = 'France'). These estimation errors are propagated in the rest of the execution plan. Moreover, showed that the propagation of these errors is exponential with the number of joins.

1.2 Unavailability of resources: at compile-time, the optimizer does not have any information about the

system state when the query will run, in particular, about the availability of resources to allocate (e.g. available memory, CPU load). Because of reasons quoted previously, the execution plans generated by a static optimizer can be sub-optimal. To correct this sub-optimality, some recent research suggest improving the accuracy of parameter values used during the choice of the execution plan. A first solution consists in improving the quality of the statistics on the data by using the previous executions. This solution was used by to improve the estimation accuracy of the operator selectivity factors and by to estimate the correlation between predicates

2. STATIC QUERY OPTIMIZATION

Query optimization is a function of many relational database management systems in which multiple query plans for satisfying a query are examined and a good query plan is identified. This may or not be the absolute best strategy because there are many ways of doing plans. There is a trade-off between the amount of time spent figuring out the best plan and the amount running the plan. Different qualities of database management systems have different ways of balancing these two. Cost based query optimizers evaluate the resource footprint of various query plans and use this as the basis for plan selection.

Typically the resources which are costed are CPU path length, amount of disk buffer space, disk storage service time, and interconnect usage between units of parallelism. The set of query plans examined is formed by examining possible access paths (e.g., primary index access, secondary index access, full file scan) and various relational table join techniques (e.g., merge join, hash join, product join). The search space can become quite large depending on the complexity of the SQL query. There are two types of optimization. These consist of logical optimization which generates a sequence of relational algebra to solve the query. In addition there is physical optimization which is used to determine the means of carrying out each operation.

A query plan (or query execution plan) is an ordered set of steps used to access or modify information in a SQL relational database management system. This is a specific case of the relational model concept of access plans.

Since SQL is declarative, there are typically a large number of alternative ways to execute a given query, with widely varying performance. When a query is submitted to the database, the query optimizer evaluates some of the different, correct possible plans for executing the query and returns what it considers the best alternative. Because query optimizers are imperfect, database users and

administrators sometimes need to manually examine and tune the plans produced by the optimizer to get better performance.

Consider each possible plan in turn.

Run it & measure performance.

The one that was fastest is the keeper.

2.1 Search techniques

In relational database systems each query execution plan can be represented by a processing tree where the leaf nodes are the base relations and the internal nodes represent operations. A search space can be restricted according to the nature of the execution plans and the applied search strategy. The nature of execution plans is determined according to two criteria: the shape of the tree structures (i.e. left-deep tree, right-deep tree and bushy tree) and the consideration of plans with Cartesian products. The queries with a large number of join predicates make the difficulty to manage associated search space which becomes too large. Space. This choice is due to the fact that this valid space represents a significant portion of the search space, which is the optimal solution. However, this assertion was never validated. Others think that these methods decrease the chances to obtain optimal solutions.

2.1 Search Domain size

The importance of the query shape and of the nature of the execution plans is due to their incidence on the size of the search space. If we have N relations in a multi-join query, the question is to know how many execution plans being able to be built, taking into account the nature of the search space. The size of this space also varies according to the shape of the query.

2.2 Cost estimation

It is one of the hardest problems in query optimization is to accurately estimate the costs of alternative query plans. Optimizers cost query plans using a mathematical model of query execution costs that relies heavily on estimates of the cardinality, or number of tuples, flowing through each edge in a query plan. Cardinality estimation in turn depends on estimates of the selection factor of predicates in the query. Traditionally, database systems estimate selectivity's through fairly detailed statistics on the distribution of values in each column, such as histograms. This technique works well for estimation of selectivities of individual predicates. However many queries have conjunctions of predicates such as `select count(*) from R where R.make='Honda' and R.model='Accord'`. Query predicates are often highly correlated (for example, `model='Accord'` implies `make='Honda'`), and it is very hard to estimate the selectivity of the conjunct in general. Poor cardinality estimates and uncaught correlation are one of the main reasons why

query optimizers pick poor query plans. This is one reason why a database administrator should regularly update the database statistics, especially after major data loads/unloads.

2.3 Query planning for nested SQL queries

A SQL query to a modern relational DBMS does more than just selections and joins. In particular, SQL queries often nest several layers of SPJ blocks (Select-Project-Join), by means of group by, exists, and not exists operators. In some cases such nested SQL queries can be flattened into a select-project-join query, but not always. Query plans for nested SQL queries can also be chosen using the same dynamic programming algorithm as used for join ordering, but this can lead to an enormous escalation in query optimization time. So some database management systems use an alternative rule-based approach that uses a query graph model.

2.4 Query execution

Once the execution plan is generated, the action switches to the storage engine, where the query is actually executed, according to the plan.

We will not go into detail here, except to note that the carefully generated execution may be subject to change during the actual execution process. For example, this might happen if:

- A determination is made that the plan exceeds the threshold for a parallel execution (an execution that takes advantage of multiple processors on the machine – more on parallel execution in the book).
- The statistics used to generate the plan were out of date, or have changed since the original execution plan was created by the optimizer.
- The results of the query are returned to you after the relational engine changes the format to match that requested in your T-SQL statement, assuming it was a SELECT.

2.5 Estimated and Actual Execution Plans

As discussed previously, there are two distinct types of execution plan. First, there is the plan that represents the output from the optimizer. This is known as an Estimated execution plan. The operators, or steps, within the plan will be labeled as logical, because they're representative of the optimizer's view of the plan.

Next is the plan that represents the output from the actual query execution. This type of plan is known, funnily enough, as the Actual execution plan. It shows what actually happened when the query executed

2.6 Execution Plan Reuse

It is expensive for the Server to generate execution plans so SQL Server will keep and reuse plans

wherever possible. As they are created, plans are stored in a section of memory called the plan cache).

When a query is submitted to the server, an estimated execution plan is created by the optimizer. Once that plan is created, and before it gets passed to the storage engine, the optimizer compares this estimated plan to actual execution plans that already exist in the plan cache. If an actual plan is found that matches the estimated one, then the optimizer will reuse the existing plan, since it's already been used before by the query engine. This reuse avoids the overhead of creating actual execution plans for large and complex queries or even simple plans for small queries called thousands of times in a minute.

Each plan is stored once, unless the cost of the plan lets the optimizer know that a parallel execution might result in better performance. If the optimizer sees parallelism as an option, then a second plan is created and stored with a different set of operations to support parallelism. In this instance, one query gets two plans.

Execution plans are not kept in memory forever. They are slowly aged out of the system using an "age" formula that multiplies the estimated cost of the plan by the number of times it has been used (e.g. a plan with a cost of 10 that has been referenced 5 times has an "age" value of 50). The lazywriter process, an internal process that works to free all types of cache (including plan cache), periodically scans the objects in the cache and decreases this value by one each time.

If the following criteria are met, the plan is removed from memory:

- More memory is required by the system
- The "age" of the plan has reached zero

The plan isn't currently being referenced by an existing connection Execution plans are not sacrosanct. Certain events and actions can cause a plan to be recompiled. It is important to remember this because recompiling execution plans can be a very expensive operation. The following actions can lead to recompilation of an execution plan:

- Changing the structure or schema of a table referenced by the query
- Changing an index used by the query
- Dropping an index used by the query
- Updating the statistics used by the query
- Calling the function, sp_recompile
- Subjecting the keys in tables referenced by the query to a large number of inserts or deletes
- For tables with triggers, significant growth of the inserted or deleted tables

- Mixing DDL and DML within a single query, often called a deferred compile
- Changing the SET options within the execution of the query
- Changing the structure or schema of temporary tables used by the query
- Changes to dynamic views used by the query
- Changes to cursor options within the query
- Changes to a remote row set, like in a distributed partitioned view
- When using client side cursors, if the FOR BROWSE options are changed

3. DYNAMIC QUERY OPTIMIZATION

The main motivations to introduce 'dynamicity' into query optimization, in particular to reduce the query execution time or to speed up the query response time of the Database management system. we propose a new algorithm which will work efficiently with simple, nested, correlated queries. the algorithm will exhibit the dynamism in calculating query execution plan for the last inner nested query. same technique is again applied for outer layer nested query. Simultaneously algorithm is generating the optimal query execution plan and executing the plan for each nested query. It works in bottom-up way by building more complex sub-plans from simpler sub-plans until the complete plan is constructed.

3. 1.Example of simple conditional Query
 SELECT {DISTINCT} <list of columns>
 FROM <list of tables>
 {WHERE <list of "Boolean Factors" (predicates)>}
 {GROUP BY <list of columns>
 {HAVING <list of Boolean Factors>}}
 {ORDER BY <list of columns>;}

The algorithm generates optimal query execution plan based on the following semantics:

1. take Cartesian product (cross-product) of tables in FROM clause, projecting to
2. only those columns that appear in other clauses
3. if there's a WHERE clause, apply all filters in it
4. if there's a GROUP BY clause, form groups on the result
5. if there's a HAVING clause, filter groups with it
6. if there's an ORDER BY clause, make sure output is in the right order
7. if there's a DISTINCT modifier,
8. remove dups

3.2 .Sample pseudo code

```

Input: query q on relations R1, R2, ..., Rn
Output: A optimal query plan for q
1 for I = 1 to n do
2  optPlan ({Ri})= accessPlans (Ri)
3  prunePlans (optPlan ({Ri}))
4 End for
```

```

5 for I = 2 to n do
6 for all S Ì { R1, R2, ..., Rn} such that |S|
  = I do
7  optPlan (S) = f
8  for all O Ì S do
9  optPlan (S) = opPlan (S) È joinPlans
  (optPlan (O), optPlan (S-O))
10 prunePlans (optPlan (S))
11 End for
12 End for
13 End for
14 finalizePlans (optPlan ({R1, R2, ...,
Rn}))
15 prunePlans (optPlan ({R1, R2, ..., Rn}))
16 return optPlan ({R1, R2, ..., Rn})
Dynamic programming algorithm.
```

4. CONCLUSION

Most of the query optimizers of database management systems are using static approaches to generate optimal execution plan for executing queries against the databases. it increases the execution cost of the queries and also consumes more system resources like CPU & Memory. The proposed algorithm which is based on greedy dynamic programming algorithm uses randomized strategies and reduces the execution cost of the queries and system resources.

5. ACKNOWLEDGMENTS

Our sincere thanks to management, principal and colleagues who supported to make it possible.

BIBLIOGRAPHY

- [1]. Evolution of Query Optimization Methods by Abdelkader Hameurlain and Franck Morvan, <http://www.csd.uoc.gr/~hy460/pdf/EvolutionofQueryOptimizationMethods.pdf>
- [2]. Alpdemir, M.N., Mukherjee, A., Gounaris, A., Paton, N.W., Fernandes, A.A.A., Sakellariou, R., Watson, P., Li, P.: Using OGSA-DQP to support scientific applications for the grid. In: Herrero, P., S. Pérez, M., Robles, V. (eds.) SAG 2004. LNCS, vol. 3458, pp. 13–24. Springer, Heidelberg (2005)
- [3]. Antonioletti, M., et al.: The design and implementation of Grid database services in OGSA-DAI. In: Concurrency and Computation: Practice & Experience, vol. 17, pp. 357–376. Wiley InterScience, Hoboken (2005) Arcangeli, J.-P., Hameurlain, A., Migeon, F., Morvan, F.: Mobile Agent Based Self-Adaptive Join for Wide-Area Distributed Query Processing. Jour. of Database Management 15(4), 25–44 (2004).
- [4]. Avnur, R., Hellerstein, J.-M.: Eddies: Continuously Adaptive Query Processing. In: Proc. of the ACM SIGMOD Intl. Conf. on Management of Data, vol. 29, pp. 261–272. ACM Press, New York (2000).
- [5]. Babu, S., Bizarro, P., De Witt, D.J.: Proactive re-optimization. In: Proc. of the ACM SIGMOD Intl. Conf. on Management of Data, pp. 107–118. ACM Press, New York (2005).

- [6] Bizarro, P., Bruno, N., De Witt, D.J.: Progressive Parametric Query Optimization. *IEEE Transactions on Knowledge and Data Engineering* 21(4), 582–594 (2009).
- [7] Bose, S.K., Krishnamoorthy, S., Ranade, N.: Allocating Resources to Parallel Query Plans in Data Grids. In: *Proc. of the 6th Intl. Conf. on Grid and Cooperative Computing*, pp. 210–220. IEEE CS, Los Alamitos (2007)
- [8] Bratbergsengen, K.: Hashing Methods and Relational Algebra Operations. In: *Proc. of*
- [9] Bruno, N., Chaudhuri, S.: Efficient Creation of Statistics over Query Expressions. In: *Proc. of the 19th Intl. Conf. on Data Engineering, Bangalore, India*, pp. 201–212. IEEE CS, Los Alamitos (2003).