An Optimal Query Execution Plan for Database Systems

Vahab Pournaghshband, University of California-Berkeley, USA

Shahriar Movafaghi, Southern New Hampshire University, USA; E-mail: s.movafaghi@snhu.edu

ABSTRACT

A major decision for the query processor of the database management system in centralized as well as distributed environments is how a query can produce the result as efficiently as possible. The typical query optimizer will not necessarily produce an optimal query plan. It simply tries to find the best possible plan within a minimum amount of time using mostly semi-accurate statistical information. In this paper, we discuss major issues regarding query optimization for relational database management systems, and expand the optimization issues for distributed database systems (DDBSs) to show how the query optimizer can choose an optimal plan for efficient execution of those queries that require multiple-site participations for producing the result. An algorithm that can be used toward more efficient query processing is presented. Our algorithm examines frequently used queries, and identifies two categories of groups of queries. First, a group of queries requiring the same procedures (including operations used), and second, a group of queries requiring data from the same site (or set of sites) for producing the result

1. INTRODUCTION

Query processing in a database environment refers to a series of activities involved in updating and retrieving data from database and it can be divided into four major phases: decomposition phase (consisting of scanning, parsing and validation), optimization phase, code generation phase, and execution phase [1]. Even though in this paper, we are mainly concerned about optimization and related issues regarding this phase of query processing, we believe a brief examination of all phases can be valuable [also see references 2, 3, 4, and 5].

Decomposition (Scanning, Parsing, and Validation) – The job of the *scanner* is to identify the language tokens found in the query, whereas the *parser* checks the query syntax. *Validation* is done to make sure that all relations and their attributes are valid and meaningful.

Optimization –There are generally many different methods that can be used to process a query and compute the result. *Query optimization* is the process of choosing the most efficient strategy for computing the result.

Query Code Generation – Once an "optimal" execution plan is produced by the query optimizer, it is the job of the *code generator* to generate the code for executing the plan.

Query Execution – The runtime database processor is responsible for executing the code (generated by the query code generator), whether in compiled mode or interpreted mode, to produce the response to the query.

We now return our attention on query optimization which is the focal point of this paper. Query optimization sub-module of the query processing module in centralized as well as distributed environment has been a subject for significant research and development. The term *optimization* is in a sense a misnomer (as claimed in [4]) because in some cases the efficient execution plan selected by the query optimizer is not necessarily the optimal strategy, but it is in fact just a *reasonably efficient* strategy for executing the query [4]. Hence, when dealing with query optimization, it would necessary to examine plans for different execution strategies. The process of selecting the execution plan for a given query can be divided into several detailed plans such as designing an efficient algorithm for executing an operation, the order of executing relational algebra operations, choosing the specific indices to use, and so on. In addition, there are other issues that are of particular interest for a distributed environment that are discussed in the next section.

2. DISTRIBUTED QUERY PROCESSING

The query optimizer chooses the most efficient query execution plan at the relational algebra level. In other words, the query optimizer attempts to find a relational algebra expression that is equivalent to the given expression, but it is more efficient to execute. This issue of finding *equivalent expression* needs to be discussed in great details and it is beyond the scope of this paper. Interested readers are encouraged to see [5]. In a distributed environment there are essential aspects of the query processor that have to be considered alongside those for centralized databases [6]. While distributing data across different sites allows those data to reside where they are most needed, but it also makes them accessible from other sites. Therefore, to process a query initiated at one site, we might need to make some data movements among several sites. And since transmission of data and messages across communications lines has a tendency to slow down the whole process, the order of data movement (that is, what data from which site should be moved first, what data should be moved next, and so on) among sites must be considered as an essential aspect of query processing for distributed database systems. One other essential aspect, worthy to consider, regarding query optimization for distributed database systems is the existence of multiple processor in the network. This allows for parallel processing of queries (and sub-queries) and data transmission which could lead to a faster response [6]. In our approach, as we will see in Section 3, these issues play a vital role in producing the optimal execution plan.

3. "OPTIMAL" EXECUTION PLAN

In both, centralized and distributed database systems, it is the responsibility of the query optimizer to transform the query as submitted by the user into an equivalent query that can be executed more efficiently. To do this, the query optimizer estimates the evaluation cost of each strategy and decides if the chosen strategy has the least cost. One process in estimating the execution cost of a query is to estimate the result size of each operation in each possible execution sequence. This is of prime interest because the size of the intermediate relations plays a significant role in the performance of an execution strategy. Unfortunately, there is no general consensus on the method of estimating the size of intermediate results. Among different techniques that have been proposed in the literature the one that is given in [7] is based on the Discrete Fourier Transformation. Their algorithms present tradeoffs between accuracy of the approximation and memory requirements. The estimation of size of the intermediate relations is based on statistical information about the relations, their attributes, and indexes. One problem regarding this approach is that most systems do not update the statistics on every change. This could lead to inaccurate estimates, and thus selection of strategies far from optimal. An alternative approach has been examined in {8 and 9] and others. Discussion of their approach, which in fact is dynamic query execution, is beyond the scope of this study.

The selection of a good strategy statically can be made effectively by the prediction of execution costs of the alternative plans prior to actually the executing the query. The execution cost is basically expressed as the combination cost of CPU, I/O and the communication costs (for distributed systems). In centralized systems, many cost functions ignore the CPU factor and emphasize on I/O cost. They compare different evaluation plans in terms of the number of block transfers between secondary storage and main memory. This (i.e., efficient memory management) has attracted the interest of many researchers such as [10, 11, and 12]. Whatever the cost factor (CPU or I/O, or both, plus communications for DDBSs), to estimate the cost for a given plan, the query optimizer estimates the cost of individual procedures making up the plan, and adds them together to get the total cost of executing the query. The process of cost estimation for individual procedures is repeated for those procedures used in different execution plans, and thus, it could become a very time consuming task. To overcome this problem, authors in [13] suggest the design of a query optimizer that examines current queries and generates a master plan for each group of queries requiring the same set (or subset) of individual procedures. The task of identifying similar queries that can be grouped together can in turn become cumbersome if not done efficiently. The fact that in the distributed database system the data reside in different locations can be a cause of many difficulties in query processing and optimization. In a DDBS each site may initiate a query, and may access data at that site and/or at several other sites on the network. In fact, the query may be broken into a set of sub queries that must be executed in order to produce the result of the query. In our approach, for an efficient process, we design an optimizer which examines frequently used queries, and identifies the following two categories of groups of queries: 1) Groups of queries requiring the same procedures (including operations used), and 2) groups of queries requiring data from the same site (or set of sites) for producing the result. Each of these two categories of queries is explained below and an algorithm that generates these groups from a set of queries is given in Section 4.

1. Groups of queries requiring the same procedures (including operations used) - For these groups, the query optimizer generates a super-query execution plan for each group before breaking it into a set of sub queries for execution. Of course, as mentioned earlier, the task of grouping queries can become costly if not done efficiently. Since Join operation is one of the most time consuming and costly operations in query processing, we take into account the Join operation as the first criterion for grouping queries in this category. That is, the query optimizer must begin this process by grouping queries that have Join operation in common (i.e., Joining the same relations as their operand.) Next, the query optimizer identifies those queries that have in common, binary operations (other than the join) such as the Cartesian Product. Finally, it reviews the remaining queries to identify and group those queries that share the same predicates (or part of them for compound predicates) for Selection operation. For each of the above groups, the query optimizer combines participating queries into a super-query before breaking it into a set of sub-queries for efficient execution of each query member of the set.
2. Groups of queries requiring data from the same site (or set of sites) for producing the result - For these groups, the query optimizer generates a super query execution plan according to data being used, and then breaks it into a set of sub-queries (one sub-query for each site participating in the plan) for efficient execution of sub-queries. This minimizes the amount of data movement among sites. In addition, the nature of DDBSs and the existence of multiple processors in the network allows for parallel processing of these sub-queries and simultaneous data transmission between sites. This could significantly speed up the process of producing the result. Furthermore, the query optimizer can follow the same process discussed earlier for the first category of groups for even more efficient execution of these sub-queries. That is, each sub-query of a group in second category, can be treated as a *base* query of the first category. In other words, each sub-query of the second category becomes a super-query discussed for the first category of groups of queries.

4. THE ALGORITHM

This algorithm generates two groups of queries. A group of queries requiring the same procedures (including operations used) and a group of queries requiring data from the same site (or set of sites) for producing the result. It also generates a group of queries belonging to both of the above groups

The algorithm reads as its input a set of queries and generates as its output the sets of "groups" of queries discussed above.

Algorithm:
Input: A set of queries Q = {q1, q2, …, qn}.

Output: Three sets of groups of queries G_1, G_2 and G_3.

Step 1: Scan queries in Q to identify

a. those queries requiring the same relations that must be JOINed for producing the results. Call this set P (P is a subset of Q) and
b. those queries requiring data from the same site(s) for producing the results. Call this set T (T is also a subset of Q).

Step 2: Scan queries in (Q – P) to identify queries requiring the same relations that are operands for *binary* operations (other than JOIN) for producing results. Call this set R (R is a subset of (Q – P))

Step 3: Scan queries in (Q – P – R) to identify queries requiring the same relations that must use the SELECT operation with common set (or subset, if compound predicate) of predicates. Call this set S (S is a subset of (Q – P – R))

$G_1 = \{P, R, S\}$

$G_2 = \{T\}$

$G_3 = \{G_1 \Omega G_2\}$

End Algorithm

5. CONCLUSIONS

There are different techniques used by the DBMSs in processing and optimizing high-level queries submitted by users. In this paper, we first discussed the major issues regarding query plan evaluation for query processing and showed how the query optimizer can choose an optimal plan for efficient execution of queries. We then discussed an efficient process for designing an optimizer which examines frequently used queries, and identifies two categories of groups of queries. One groups of queries requiring the same procedures (including operations used), and one groups of queries requiring data from the same site (or set of sites) for producing the result. Finally, we presented an algorithm which examines frequently used queries, and identifies those two groups of queries.

REFERENCES

1. C. Connoly, C. Begg, and A. Strachan, "Database Systems," 2nd Edition, Addison-Wesley, 1999.
2. S. Movafaghi and H. Pournaghshband, "Data Warehouse Query Processing and Optimization Architecture," Proceedings of the Software Engineering-Research and Practice Conference, June 2004.
3. J. Yang, et. al., "Tracking the Challenges of Materialized View Design in Data Warehouse Environment", Proceedings of the 7th International Workshop on Research Issues in Data Engineering," 1997.
4. R. Elmasri, et. al., "Fundamental of Database Systems," 4th Edition, Addison-Wesley, 2003.
5. A. Silberschatz, H. F. Korth, and S. Sudarshan, "Database System Concepts," 5th Edition, McGraw-Hill, 2006.
6. D. Bell, and J. Grimson "Distributed Database Systems," Addison-Wesley, 1992.
7. K. Srac, O. Egecioglu, and A. E. Abbdi, "Iterated DFT Based Techniques for Join Size Estimation," Proceedings of ACM-CIKM Conference, 1998.
8. P. Bodrik, J. S. Riordon, and C. Jacob, "Dynamic Distributed Query Processing Techniques," Proceedings of ACM-CS Conference, 1989.
9. R. L. Cole and G. Graefe, "Optimization of Dynamic Query Evaluation Plans," Proceedings of ACM-SIGMOD Conference, 1994.
10. B. Nag and D. J. DeWitt, "Memory Allocation Strategies for Complex Decision Support Queries," Proceedings of ACM-CIKM Conference, 1998.
11. L. Bouganin, O. Kapitskaia, and P. Valdurier, "Memory-Adaptive Scheduling for Large Query Execution," Proceedings of ACM-CIKM Conference, 1998.
12. L.D. Shapiro, "Join Processing in Database Systems with Large Main Memories," ACM-TODS, vol 11, no 3, 1986.
13. H. Pournaghshband, A. R. Salehnia, "What makes a Query Processor Efficient: Optimization Issues for DBMSs," Proceedings of Association of Management Conference, 1999.