Query Optimization in Distributed Databases

Dilşat ABDULLAH
1302108

Department of Computer Engineering
Middle East Technical University
December 2003

ABSTRACT

Query optimization refers to the process of producing a query execution plan (QEP) which represents an execution strategy for the query. The selected plan minimizes an objective cost function. Selecting the optimal execution strategy for a query is NP-hard in the number of relations. For this research, different operators will be implemented, and we will try to find an optimal strategy (optimizer), and perhaps more important, to avoid bad strategy.

TABLE OF CONTENTS

1. INTRODUCTION

The great commercial success of database systems is partly due to the development of sophisticated query optimization technology: users pose queries in a declarative way using SQL, and the optimizer of the database system finds a good way to execute these queries. The optimizer, for example, determines which indices should be used to execute a query and in which order the operations of a query (e.g., joins and group-bys) should be executed. To this end, the optimizer enumerates alternative plans, estimates the cost of every plan using a cost model, and chooses the plan with the lowest estimated cost.

A more detailed view of the query optimization and execution component in a typical DBMS architecture is shown in Figure 1. Queries are parsed and then presented to a query optimizer, which is responsible for identifying an efficient execution plan for evaluating the query. The optimizer generates alternative plans and chooses the plan with the least estimated cost. To estimate the cost of a plan, the optimizer uses information in the system catalog.

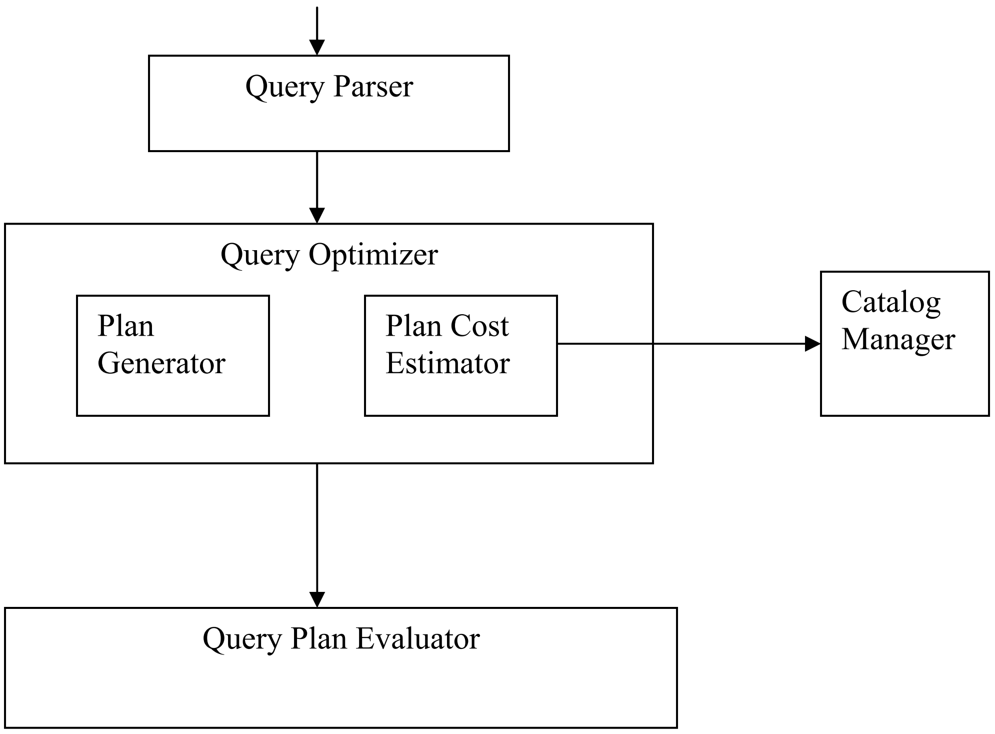

Figure 1: Query Paring, Optimization, and Execution

The query optimizer implements a set of physical operators. An operator takes as input one or more data streams and produces an output data stream. Examples of physical operators are (external) sort, sequential scan, index-scan, nested-loop join, and sort-merge join. We refer to such operators as physical operators since they are not necessarily tied one-to-one with relational operators, The simplest way to think of physical operators is as pieces of code that are used as building blocks to make possible the execution of SQL queries. An abstract representation of such an execution is a physical operator tree, as illustrated in Figure 2. The edges in an operator tree represent the data flow among the physical operators. We use the terms physical operator tree and execution plan (or, simply plan) interchangeably. The execution engine is responsible for the execution of the plan that results in generating answers to the query. Therefore, the capabilities of the query execution engine determine the structure of the operator trees that are feasible.

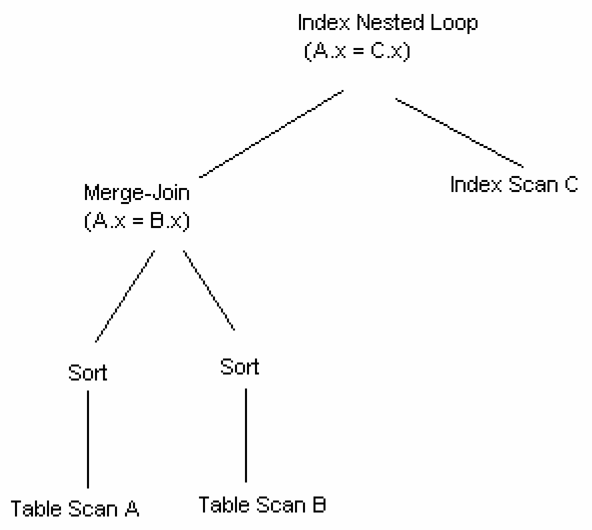

Figure 2. Operator Tree

The query optimizer is responsible for generating the input for the execution engine. It takes a parsed representation of a SQL query as input and is responsible for generating an efficient execution plan for the given SQL query from the space of possible execution plans. The task of an optimizer is nontrivial since for a given SQL query, there can be a large number of possible operator trees:

- The algebraic representation of the given query can be transformed into many other logically equivalent algebraic representations: e.g.,
  Join(Join(A,B),C)= Join(Join(B,C),A).

- For a given algebraic representation, there may be many operator trees that implement the algebraic expression, e.g., typically there are several join algorithms supported in a database system.

Furthermore, the throughput or the response times for the execution of these plans may be widely different. Therefore, a judicious choice of an execution by the optimizer is of critical importance. Thus, query optimization can be viewed as a difficult search problem. In order to solve this problem, we need to provide:

- A space of plans (search space).

- A cost estimation technique so that a cost may be assigned to each plan in the search space. Intuitively, this is an estimation of the resources needed for the execution of the plan.
- An enumeration algorithm that can search through the execution space.

A good optimizer is one where (1) the search space includes plans that have low cost (2) the costing technique is accurate (3) the enumeration algorithm is efficient. Each of these three tasks is nontrivial and that is why building a good optimizer is an enormous undertaking.

2. QUERY OPTIMIZATION DESCRIPTION

Relational query optimization refers to the process of building an optimal (i.e., a minimum cost) execution plan, given an input query.

A tuple is a list of elements, called attributes. The number of attributes of a tuple is called its arity. The set of attributes of a tuple is called its scheme. A relation is a set of elements, each of which is a tuple with the same arity. Suppose we have two relations R and S, with sets of attributes $\{A_1\ldots A_m\}$ and $\{B_1,\ldots,B_p\}$, respectively. The join of R and S with respect to attributes A_i and B_j , respectively, is formed by taking each tuple r from R and each tuple s from S and comparing them. If the attribute A_i of r equals the attribute B_j of s, then a tuple is formed from r and s by taking the attributes of r followed by all attributes of s, omitting the repeated attribute B_j. The scheme of the joined relation is then $\{A_1,\ldots, A_m, B_1,\ldots, B_{j-1}, B_{j+1},\ldots, B_p\}$. A relational query language is a non-procedural one, i.e., a query specifies the wanted data and not the way the data is obtained. The following is a typical relational query:

Select *R*2.*
From *R*1, *R*2, *R*3, *R*4
Where *R*1:*A* = *R*2:*A* **and** *R*2:*B* = *R*3:*B* **and** *R*3:*C* = *R*4:*C*

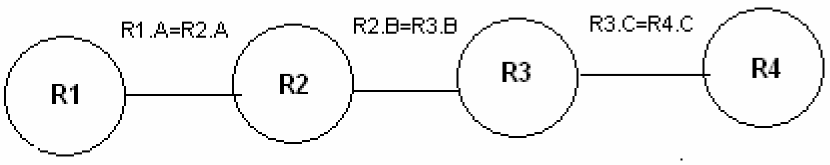

Figure 3 Query Graph

This query involves three joins between four relations. The query graph depicted in Figure 3 is a subgraph of the database schema and establishes the join predicates (edges) between the relations (nodes) involved in the query. Here, we want to select all tuples from R2 such that their A attribute is equal to the A attribute of a tuple from R1, and the B attribute of the former is also equal to the B attribute of a tuple from R3 whose C attribute is equal to the C attribute of a tuple from R4. The join is an expensive operation, because it requires the matching of tuples of relations according to their join attributes. The equal comparison characterizes the equijoin, the most frequent operation, due to the need of normalizing relations, inherent to the relational model.

The first task of the query optimizer in a relational context is to decide in which order the relations are accessed. The join ordering problem is NP-hard if the number of relations of the input query is not fixed. Furthermore, the optimizer must choose the access strategy for each individual relation. An access strategy involves, for example, choosing an access path, e.g., sequential scan or index. In a distributed environment, the relations may be located at different servers. Furthermore, relations may be partitioned among several servers, according to a range criterion. To perform a join between two relations, they must be located at the same site, or at least their corresponding fragments (when two relations are partitioned on the same servers and the same criterion is applied to the join attributes, the join may be concurrently executed on several servers and the partial results unioned and sent to the demander). In this case, the access strategy must also specify the location in which the join takes place. Either both relations or their corresponding fragments are located on the same site and the join may be performed without any redistribution, or a redistribution must be performed prior to the join.

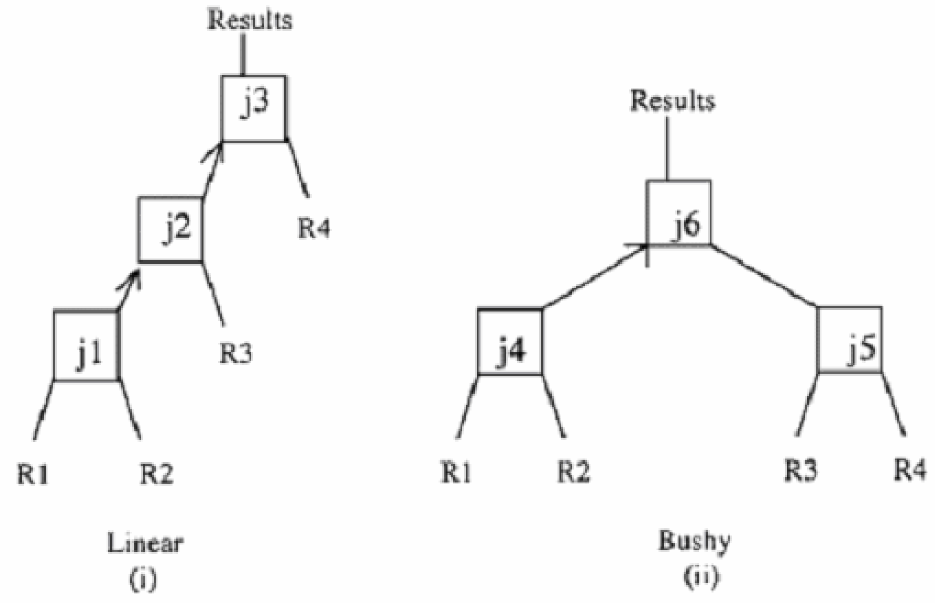

Figure 4 Execution Plan as operator trees

An execution plan captures the join ordering, the access strategy for each individual relation and additional execution information. It is typically represented as a binary operator tree, where the leaves are base relations and the internal nodes are join (or union) operator nodes, whose result is captured by transient relations. The transient relations may or may not be materialized. Unary relational operators, such as selections and projections, are not represented, because they are applied "on-the-fly", before the tuple is passed to the join or union operation. Figure 4 shows two different execution plans for the sample query. Although not represented for simplicity, each join node captures a join algorithm (i.e., nested loop, merge join, or hash join) and an indication of redistribution, when needed. In our example, depicted in Figure 4, relations R1 and R2 are located at server S1, while relations R3 and R4 are located at server S2, i.e., the relations are pairwise located at different servers. Thus, both joins j4 and j5 can be executed respectively at servers S1 and S2 without any redistribution. However, join j6 needs a redistribution prior to the operation: either the result of j4 is sent to S2 or the result of j5 to S1.

Directed arcs linking two operator nodes express the fact that the transient relation produced by the first is consumed in pipeline by the second, i.e., in a tuple-by-tuple basis. Otherwise, the transient relation is completely stored, before it is consumed. Pipeline is obviously less expensive, and is chosen whenever possible. For example, all joins in figure 4(i) are executed using the nested loop algorithm. Thus, there is no need for materializing intermediate results. On the other hand, some join algorithms, e.g., merge join, require both relations to be sorted according to their join attributes. In this case, a previous materialization and sorting of an operand relation may be required.

We say that an operator tree corresponds to a complete execution plan if it captures all the relations of the input query (the two trees in figure 4 represent complete plans). Otherwise, we say that the plan is partial.

3. COST MODEL

3.1 Cost Model in DBMS

The cost model assigns an estimated cost to any partial or complete plan in the search space. It also determines the estimated size of the data stream for output of every operator in the plan. It relies on:

a) A set of statistics maintained on relations and indexes, e.g., number of data pages in a relation, number of pages in an index, number of distinct values in a column

b) Formulas to estimate selectivity of predicates and to project the size of the output data stream for every operator node. For example, the size of the output of a join is estimated by taking the product of the sizes of the two relations and then applying the joint selectivity of all applicable predicates.

c) Formulas to estimate the CPU and I/O costs of query execution for every operator. These formulas take into account the statistical properties of its input data streams, existing access methods over the input data streams, and any available order on the data stream (e.g., if a data stream is ordered, then the cost of a sort-merge join on that stream may be significantly reduced). In addition, it is also checked if the output data stream will have any order.

The cost model gives a cost estimate for each operator tree. The cost refers to resource consumption, either space or time. Typically, query optimizers estimate the cost as time consumption.

3.2 Cost Model in Distributed DBMS

In a distributed execution environment, there are two different time consumption estimates to be considered: *total time* or *response time*. The former is the sum of the time consumed by each processor, regardless of concurrency, while the latter considers that operations may be carried out concurrently. Thus, response time is a more appropriate estimate, since it corresponds to the time the user has to wait for an answer to the query. In a distributed environment, the execution of an operator tree ***S*** is split into several phases. Pipelined operations are executed in the same phase, whereas a storing indication establishes the boundary between one phase and the subsequent one. For example, the operator tree in Figure 4(i) is executed in one single phase, while the one in Figure 4(ii) in two phases. Resource contention is also another reason for splitting an operator tree into different phases. For instance, if a sequence of operations that could be concurrently executed require more memory than available (e.g., if the memory is not sufficient to store the entire hash tables for pipelined operations in the hash join algorithm), then it is split into two or more phases. An operator tree is also split into different phases if independent operations (which, in principle, could remain in the same phase) should be executed at the same home site: in this case, the operations are not concurrently executed just because the homes are the same and, accordingly, they are scheduled at different phases. Let $\phi(s)$ denote the set of phases of plan s, while $\Phi \in \phi(s)$ denotes one individual phase. Moreover, let $o(\phi)$ denote the set of operations (i.e., the set of join operator nodes within the corresponding phase ϕ of the operator tree). Since the operations within the same phase are carried in parallel, the response time *respTime(* ϕ *)* of each phase $\Phi \in \phi(s)$ is the time needed to execute the most expensive operation, plus the necessary delay to start the pipeline, given by:

$$respTime(\phi) = \max_{i \in O(\phi)} \{execTime(i) + pipeDelay(i)\}$$

where $i \in O(\phi)$ denotes an operator node, *execTime(i)* is the execution time of operation i , and *pipeDelay(i)* is the waiting period of operation i (time needed for the producer to deliver the first result tuples). The latter is null if the input relations of operation i are stored.

The cost *execTime(i)* of an operator node i is built up from two components: the time to execute an operation and the time to redistribute transient results (transmission time). The time to execute an operation depends on the chosen algorithm, e.g., nested loop for join. To compute it, the optimizer uses statistics stored on a metabase, that are periodically updated by the database administrator. Examples of such statistics are the cardinalities of relations and the number of distinct values of an attribute. As an example of a cost function, we show below the cost function *execTime(j2)* associated with the join node j2 in figure 2(i). It consumes its left operand in pipeline, while the right operand is stored. Thus, the cost is:

$$execTime(J_s) = transmit(R_s) + \max(\cos t_{nestedLoop}(join(j_1, R_3)), transmit(j_1))$$

where *transmit(R)* refers to the cost of redistributing R if needed. In a distributed environment, it is crucial to minimize this transmission time, since it impacts considerably the response time. Thus, our optimizer tries to exploit the placement of the operands to avoid redistribution. The transmission cost is estimated using system-dependent parameters, such as the network speed. In the current example, either j1 or R3 must be redistributed, since they are located at different servers. Thus, in the formula above one of the two transmission costs is different from zero and the other is null. The cost of computing the join of node j1 with relation R3 using, for example, the nested loop algorithm is:

$$Cost_{nestedLoop}(join(j_1, R_3)) = card(j_1) * accessCost(j_1) + card(j_1) * card(R_3) * accessCost(R_3)$$

where, for any relation R, *accessCost(R)* is equal to zero if relation R is not stored (i.e., it resides in the main memory) and *card(R)* denotes the number of tuples in relation R.

To estimate the cost f(s) of an operator tree s, the optimizer sums up the costs of all phases $\phi \in \phi(s)$, which gives the total response time for executing the operator tree s:

$$f(s) = \sum_{\phi \in \phi(s0} [respTime(\phi) + storeDelay(\phi)]$$

where $storeDelay(\phi)$ is the time necessary to store the output results produced by phase ϕ. The latter is null if ϕ is the last phase, assuming that the results are delivered as soon as they are produced.

4. CONCLUSION

Optimization is much more than transformations and query equivalence. The infrastructure for optimization is significant. Designing effective and correct SQL transformations is hard, developing a robust cost metric is elusive, and building extensible enumeration architecture is a significant undertaking. Despite many years of work, significant open problems remain. However, an understanding of the existing engineering framework is necessary for making effective contribution to the area of query optimization.

REFERENCES

[1] Ibaraki, T. and T. Kameda. (1984). "On the Optimal Nesting Order for Computing N-Relational Joins." ACM Transactions on Data Bases 9, 482–541.

[2] Chaudhuri, S., Shim K. Optimization of Queries with Userdefined Predicates. In Proc. of VLDB. Mumbai. 1996.

[3] Ganski. R.A., Long, H.K.T. Optimization of Nested SQL Queries Revisited. In Proc. of ACM SIGMOD. San Francisco, 1987.

[4] Gassner, P.. Lohman. G., Schiefer. K.B. Query Optimization in the IBM DB2 Family. IEEE Data Engineering Bulletin, Dec. 1993.

[5] Graefe. G.. Dewitt D..L The Exodus Optimizer Generator. In Proc. of ACM IGMOD, San Francisco,. 1987.

[6] Ozsu M.T., Valduriez, P. Principles of Distributed Database Systems. Prentice-Hall, 1999.

[7] Grnefe, G,, McKenna, W.J. The Volcano Optimizer Generator: Extensibility and Efficient Search. In Proc. of the IEEE Conference on Data Engineering, Vienna, 1993

[8] Pirahesh, H., Hellerstein J.M., Hasan. W. Extensible/Rule Based Query Rewrite Optimization in Starburst. In Proc. of ACM SIGMOD 1992.

[9] Mnckert, L.F., Lohman, G.M. R* Optimizer Validation and Performance Evnluation for Local Queries. In Proc. of ACM SIGMOD, 1986.

[10] Hasan, W, Optimization of SQL Queries for Parallel Machines.LNCS 1182, Springer-Verlag, 1996.

[11] Haas, L., Freytag, J.C., Lehman, G.M., Pirahesh, H. Extensible Query Processing in Starburst. In Proc. of ACM SIGMOD,Portlnnd, 1989.

[12] Q. Zhu an P.A. Larson. Developing Regression Cost Model for Mutidatabase Systems. *In Pro. 4th Int. Conf. on Parallel and Distributed Information Systems* (PDIS), December 1996, p220-231

[13] C. T. Yu and C. C. Chang. Distributed Query Processing. ACM *Comput. Surveys* December 1984.

[14] M. M. Zloof. Query-by-Example: A Data Base Language. IBM Syst. J. 1997.

[15] L. Shapiro. Join Processing in Database Systems with Large Main Memories. ACM *Trans. Databse Syst*. September 1986

[16] E. J. Shekita and M. J. Carey. A performance Evaluation of Pointer-Based Joins. In Pro. ACM SIGMOD *Int. Conf. On Management of Data*, may 1990.

[17] D. Shasha and T.-L. Wang. Optimizing Equijoin Queries in Distributed Database Where Relations are Hash Partitioned. *ACM Trans. Database* Syst. June 1991.

[18] P. Valduriez. Semi-Join Algorithms for Distributed Database Machines. In J-J. Schneider(ed.), *Distributed Data Bases*, Amsterdam: North-Holland, 1982.

[19]P. Valduriez. Join Indices. *ACM Trans. Database Syst*. June 1987

[20] P. Valduriez and G. Gardarin. Join and Semi-join Algorithms fro a Multi Processor Database Machine. *ACM Trans. Database Systems*. 1986

[21]P. Valduriez. Optimization of Complex Queries Using Join Indices. IEEE Q. Bull. Databse Eng. December 1986.