Query Processing in Distributed Database System

Issue No. 03 - May (1979 vol. 5)
ISSN: 0098-5589
pp: 177-187
DOI Bookmark: http://doi.ieeecomputersociety.org/10.1109/TSE.1979.234179

S.B. Yao
A.R. Hevner , Department of Computer Science, Purdue University

ABSTRACT
Query processing in a distributed system requires the transmission f data between computers in a network. The arrangement of data transmissions and local data processing is known as a distribution strategy for a query. Two cost measures, response time and total time are used to judge the quality of a distribution strategy. Simple algorithms are presented that derive distribution strategies which have minimal response time and minimal total time, for a special class of queries. These optimal algorithms are used as a basis to develop a general query processing algorithm. Distributed query examples are presented and the complexity of the general algorithm is analyzed. The integration of a query processing subsystem into a distributed database management system is discussed.

INDEX TERMS
system modeling, Computer network, database, distributed database systems, distributed processing, distribution strategy, heuristic algorithms, query processing, redundant data, relational data model

Discuss the objectives of Query Processing.

Ans. Objectives of Query Processing

- The main objectives of query processing in a distributed environment is to form a high level query on a distributed database, which is seen as a single database by the users, into an efficient execution strategy expressed in a low level language in local databases.

- An important point of query processing is query optimization. Because many execution strategies are correct transformations of the same high level query the one that optimizes (minimizes) resource consumption should be retained.
- The good measure of resource consumption are:
  - The total cost that will be incurred in processing the query. It is the dome of all times incurred in processing the operations of the query at various sites and intrinsic communication.
  - The resource time of the query. This is the time elapsed for executing the query. Since operations can be executed in parallel at different sited, the response time of a query may be significantly less than its cost.
- Obviously the total cost should be minimized.
  - In a distributed system, the total cost to be minimized includes CPU, I\O, and communication costs. This cost can be minimized by reducing the number of I\O operation through fast access methods to the data and efficient use of main memory. The communication cost is the time needed for exchanging the data between sited participating in the execution of the query.
  - In centralized systems, only CPU and I\O cost have to be considered.

Layers of Query Processing

The problem of query processing can itself be decomposed into several subproblems, corresponding to various layers. A generic layering scheme for query processing is shown where each layer solves a well-defined subproblem. To simplify the discussion, let us assume a static and semicentralized query processor that does not exploit replicated fragments. The input is a query on global data expressed in relational calculus. This query is posed on global (distributed) relations, meaning that data distribution is hidden. Four main layers are involved in distributed query processing. The first three layers map the input query into an optimized distributed query execution plan. They perform the functions of *query decomposition*, *data localization*, and *global query optimization*. Query decomposition and data localization correspond to query rewriting. The first three layers are performed by a central control site and use schema information stored in the global directory. The fourth layer performs *distributed query execution* by executing the plan and returns the answer to the query. It is done by the local sites and the control site.

Generic Layering Scheme for Distributed Query Processing

Query Decomposition

The first layer decomposes the calculus query into an algebraic query on global relations. The information needed for this transformation is found in the global conceptual schema describing the global relations. However, the information about data distribution is not used here but in the next layer. Thus the techniques used by this layer are those of a centralized DBMS.

Query decomposition can be viewed as four successive steps. First, the calculus query is rewritten in a *normalized form* that is suitable for subsequent manipulation. Normalization of a query generally involves the manipulation of the query quantifiers and of the query qualification by applying logical operator priority.

Second, the normalized query is analyzed semantically so that incorrect queries are detected and rejected as early as possible. Techniques to detect incorrect queries exist only for a subset of relational calculus. Typically, they use some sort of graph that captures the semantics of the query.

Third, the correct query (still expressed in *relational calculus*) is simplified. One way to simplify a query is to eliminate redundant predicates. Note that redundant queries are likely to arise when a query is the result of system transformations applied to the user query. Such transformations are used for performing semantic data control (views, protection, and semantic integrity control).

Fourth, the calculus query is *restructured* as an *algebraic query*. That several algebraic queries can be derived from the same calculus query, and that some algebraic queries are “better” than others. The quality of an algebraic query is defined in terms of expected performance. The traditional way to do this transformation toward a “better” algebraic specification is to start with an initial algebraic query and transform it in order to find a “good” one. The initial algebraic query is derived immediately from the calculus query by translating the predicates and the target statement into relational operators as they appear in the query. This directly translated algebra query is then restructured through transformation rules. The algebraic query generated by this layer is good in the sense that the worse executions are typically avoided. For instance, a relation will be accessed only once, even if there are several select predicates. However, this query is generally far from providing an optimal execution, since information about data distribution and fragment allocation is not used at this layer.

Data Localization

The input to the second layer is an algebraic query on global relations. The main role of the second layer is to localize the query's data using data distribution information in the fragment schema. We saw that relations are fragmented and stored in disjoint subsets, called *fragments*, each being stored at a different site. This layer determines which fragments are involved in the query and transforms the distributed query into a query on fragments. Fragmentation is defined by fragmentation predicates that can be expressed through relational operators. A global relation can be reconstructed by applying the fragmentation rules, and then deriving a program, called a *localization program*, of relational algebra operators, which then act on fragments. Generating a query on fragments is done in two steps. First, the query is mapped into a *fragment query* by substituting each relation by its reconstruction program (also called *materialization program*). Second, the fragment query is simplified and restructured to produce another "good" query. Simplification and restructuring may be done according to the same rules used in the decomposition layer. As in the decomposition layer, the final fragment query is generally far from optimal because information regarding fragments is not utilized.

Global Query Optimization

The input to the third layer is an algebraic query on fragments. The goal of query optimization is to find an execution strategy for the query which is close to optimal. Remember that finding the optimal solution is computationally intractable. An execution strategy for a distributed query can be described with relational algebra operators and *communication primitives* (send/receive operators) for transferring data between sites. The previous layers have already optimized the query, for example, by eliminating redundant expressions. However, this optimization is independent of fragment characteristics such as fragment allocation and cardinalities. In addition, communication operators are not yet specified. By permuting the ordering of operators within one query on fragments, many equivalent queries may be found.

Query optimization consists of finding the "best" ordering of operators in the query, including communication operators that minimize a cost function. The cost function, often defined in terms of time units, refers to computing resources such as disk space, disk I/Os, buffer space, CPU cost, communication cost, and so on. Generally, it is a weighted combination of I/O, CPU, and communication costs. Nevertheless, a typical simplification made by the early distributed DBMSs, as we mentioned before, was to consider communication cost as the most significant factor. This used to be valid for wide area networks, where the limited bandwidth made communication much more costly than local processing. This is not true anymore today and communication cost can be lower than I/O cost. To select the ordering of operators it is necessary to predict execution costs of alternative candidate orderings. Determining execution costs before query execution (i.e., static optimization) is based on fragment statistics and the formulas for estimating the cardinalities of results of relational operators. Thus the optimization decisions depend on the allocation of fragments and available statistics on fragments which are recorder in the allocation schema.

An important aspect of query optimization is *join ordering*, since permutations of the joins within the query may lead to improvements of orders of magnitude. One basic technique for optimizing a sequence of distributed join operators is through the *semijoin* operator. The main value of the semijoin in a distributed system is to reduce the size of the join operands and then the communication cost. However, techniques which consider local processing costs as well as communication costs may not use semijoins because they might increase local processing costs. The output of the query optimization layer is a optimized algebraic query with communication operators included on fragments. It is typically represented and saved (for future executions) as a *distributed query execution plan*.

Distributed Query Execution

The last layer is performed by all the sites having fragments involved in the query. Each subquery executing at one site, called a *local query*, is then optimized using the local schema of the site and executed. At this time, the algorithms to perform the relational operators may be chosen. Local optimization uses the algorithms of centralized systems.

The goal of distributed query processing may be summarized as follows: given a calculus query on a distributed database, find a corresponding execution strategy that minimizes a system cost function that includes I/O, CPU, and communication costs. An execution strategy is specified in terms of relational algebra operators and communication primitives (send/receive) applied to the local databases (i.e., the relation fragments). Therefore, the complexity of relational operators that affect the performance of query execution is of major importance in the design of a query processor.