Chapter 6
Overview of Query Processing

The success of relational database technology in data processing is due, in part, to the availability of non-procedural languages (i.e., SQL), which can significantly improve application development and end-user productivity. By hiding the low-level details about the physical organization of the data, relational database languages allow the expression of complex queries in a concise and simple fashion. In particular, to construct the answer to the query, the user does not precisely specify the procedure to follow. This procedure is actually devised by a DBMS module, usually called a *query processor*. This relieves the user from query optimization, a time-consuming task that is best handled by the query processor, since it can exploit a large amount of useful information about the data.

Because it is a critical performance issue, query processing has received (and continues to receive) considerable attention in the context of both centralized and distributed DBMSs. However, the query processing problem is much more difficult in distributed environments than in centralized ones, because a larger number of parameters affect the performance of distributed queries. In particular, the relations involved in a distributed query may be fragmented and/or replicated, thereby inducing communication overhead costs. Furthermore, with many sites to access, query response time may become very high.

In this chapter we give an overview of query processing in distributed DBMSs, leaving the details of the important aspects of distributed query processing to the next two chapters. The context chosen is that of relational calculus and relational algebra, because of their generality and wide use in distributed DBMSs. As we saw in Chapter 3, distributed relations are implemented by fragments. Distributed database design is of major importance for query processing since the definition of fragments is based on the objective of increasing reference locality, and sometimes parallel execution for the most important queries. The role of a distributed query processor is to map a high-level query (assumed to be expressed in relational calculus) on a distributed database (i.e., a set of global relations) into a sequence of database operators (of relational algebra) on relation fragments. Several important functions characterize this mapping. First, the *calculus query* must be *decomposed* into a sequence of relational operators called an *algebraic query*. Second, the data accessed by the

M.T. Özsu and P. Valduriez, *Principles of Distributed Database Systems: Third Edition*,
DOI 10.1007/978-1-4419-8834-8_6, © Springer Science+Business Media, LLC 2011

query must be *localized* so that the operators on relations are translated to bear on local data (fragments). Finally, the algebraic query on fragments must be extended with communication operators and *optimized* with respect to a cost function to be minimized. This cost function typically refers to computing resources such as disk I/Os, CPUs, and communication networks.

The chapter is organized as follows. In Section 6.1 we illustrate the query processing problem. In Section 6.2 we define precisely the objectives of query processing algorithms. The complexity of relational algebra operators, which affect mainly the performance of query processing, is given in Section 6.3. In Section 6.4 we provide a characterization of query processors based on their implementation choices. Finally, in Section 6.5 we introduce the different layers of query processing starting from a distributed query down to the execution of operators on local sites and communication between sites. The layers introduced in Section 6.5 are described in detail in the next two chapters.

6.1 Query Processing Problem

The main function of a relational query processor is to transform a high-level query (typically, in relational calculus) into an equivalent lower-level query (typically, in some variation of relational algebra). The low-level query actually implements the execution strategy for the query. The transformation must achieve both correctness and efficiency. It is correct if the low-level query has the same semantics as the original query, that is, if both queries produce the same result. The well-defined mapping from relational calculus to relational algebra (see Chapter 2) makes the correctness issue easy. But producing an efficient execution strategy is more involved. A relational calculus query may have many equivalent and correct transformations into relational algebra. Since each equivalent execution strategy can lead to very different consumptions of computer resources, the main difficulty is to select the execution strategy that minimizes resource consumption.

Example 6.1. We consider the following subset of the engineering database schema given in Figure 2.3:

EMP(ENO, ENAME, TITLE)
ASG(ENO, PNO, RESP, DUR)

and the following simple user query:

"Find the names of employees who are managing a project"

The expression of the query in relational calculus using the SQL syntax is

```
SELECT ENAME
FROM   EMP,ASG
WHERE  EMP.ENO = ASG.ENO
AND    RESP = ''Manager''
```

Two equivalent relational algebra queries that are correct transformations of the query above are

$$\Pi_{\text{ENAME}}(\sigma_{\text{RESP="Manager"} \wedge \text{EMP.ENO=ASG.ENO}} (\text{EMP} \times \text{ASG}))$$

and

$$\Pi_{\text{ENAME}}(\text{EMP} \bowtie_{\text{ENO}} (\sigma_{\text{RESP="Manager"}} (\text{ASG})))$$

It is intuitively obvious that the second query, which avoids the Cartesian product of EMP and ASG, consumes much less computing resources than the first, and thus should be retained. ♦

In a centralized context, query execution strategies can be well expressed in an extension of relational algebra. The main role of a centralized query processor is to choose, for a given query, the best relational algebra query among all equivalent ones. Since the problem is computationally intractable with a large number of relations [Ibaraki and Kameda, 1984], it is generally reduced to choosing a solution close to the optimum.

In a distributed system, relational algebra is not enough to express execution strategies. It must be supplemented with operators for exchanging data between sites. Besides the choice of ordering relational algebra operators, the distributed query processor must also select the best sites to process data, and possibly the way data should be transformed. This increases the solution space from which to choose the distributed execution strategy, making distributed query processing significantly more difficult.

Example 6.2. This example illustrates the importance of site selection and communication for a chosen relational algebra query against a fragmented database. We consider the following query of Example 6.1:

$$\Pi_{\text{ENAME}} (\text{EMP} \bowtie_{\text{ENO}} (\sigma_{\text{RESP="Manager"}} (\text{ASG})))$$

We assume that relations EMP and ASG are horizontally fragmented as follows:

$$\text{EMP}_1 = \sigma_{\text{ENO} \leq \text{"E3"}} (\text{EMP})$$
$$\text{EMP}_2 = \sigma_{\text{ENO} > \text{"E3"}} (\text{EMP})$$
$$\text{ASG}_1 = \sigma_{\text{ENO} \leq \text{"E3"}} (\text{ASG})$$
$$\text{ASG}_2 = \sigma_{\text{ENO} > \text{"E3"}} (\text{ASG})$$

Fragments ASG_1, ASG_2, EMP_1, and EMP_2 are stored at sites 1, 2, 3, and 4, respectively, and the result is expected at site 5.

For the sake of pedagogical simplicity, we ignore the project operator in the following. Two equivalent distributed execution strategies for the above query are

shown in Figure 6.1. An arrow from site i to site j labeled with R indicates that relation R is transferred from site i to site j. Strategy A exploits the fact that relations EMP and ASG are fragmented the same way in order to perform the select and join operator in parallel. Strategy B centralizes all the operand data at the result site before processing the query.

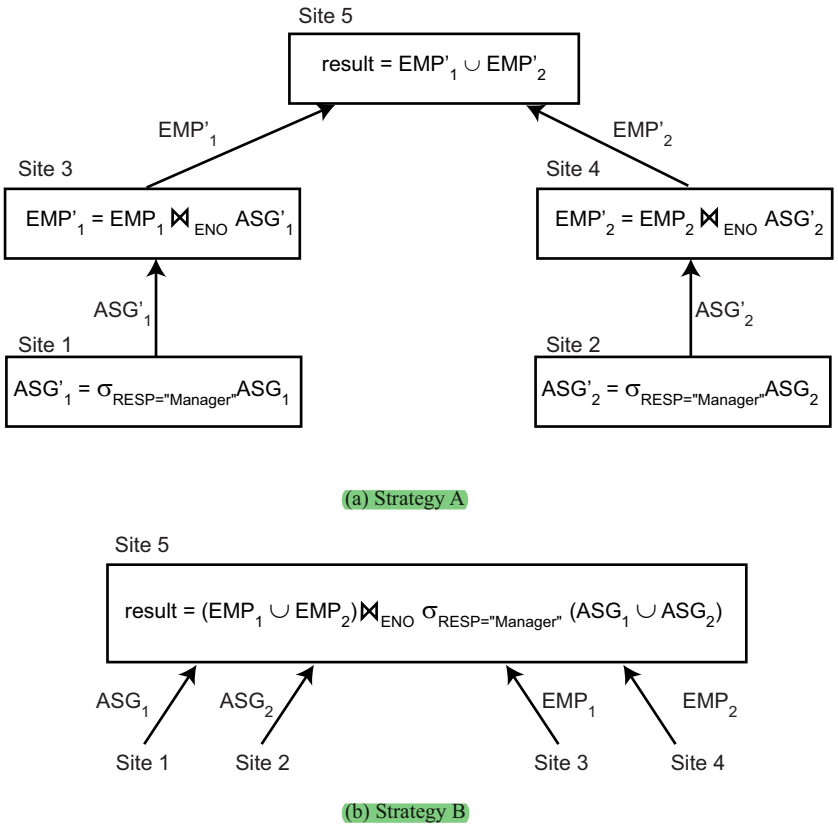

Fig. 6.1 Equivalent Distributed Execution Strategies

To evaluate the resource consumption of these two strategies, we use a simple cost model. We assume that a tuple access, denoted by *tupacc*, is 1 unit (which we leave unspecified) and a tuple transfer, denoted *tuptrans*, is 10 units. We assume that relations EMP and ASG have 400 and 1000 tuples, respectively, and that there are 20 managers in relation ASG. We also assume that data is uniformly distributed among sites. Finally, we assume that relations ASG and EMP are locally clustered on attributes RESP and ENO, respectively. Therefore, there is direct access to tuples of ASG (respectively, EMP) based on the value of attribute RESP (respectively, ENO).

The total cost of strategy A can be derived as follows:

1. Produce ASG′ by selecting ASG requires $(10+10)*tupacc$	=	20
2. Transfer ASG′ to the sites of EMP requires $(10+10)*tuptrans$	=	200
3. Produce EMP′ by joining ASG′ and EMP requires $(10+10)*tupacc*2$	=	40
4. Transfer EMP′ to result site requires $(10+10)*tuptrans$	=	200
The total cost is		460

The cost of strategy B can be derived as follows:

1. Transfer EMP to site 5 requires $400*tuptrans$	=	4,000
2. Transfer ASG to site 5 requires $1000*tuptrans$	=	10,000
3. Produce ASG′ by selecting ASG requires $1000*tupacc$	=	1,000
4. Join EMP and ASG′ requires $400*20*tupacc$	=	8,000
The total cost is		23,000

In strategy A, the join of ASG′ and EMP (step 3) can exploit the cluster index on ENO of EMP. Thus, EMP is accessed only once for each tuple of ASG′. In strategy B, we assume that the access methods to relations EMP and ASG based on attributes RESP and ENO are lost because of data transfer. This is a reasonable assumption in practice. We assume that the join of EMP and ASG′ in step 4 is done by the default nested loop algorithm (that simply performs the Cartesian product of the two input relations). Strategy A is better by a factor of 50, which is quite significant. Furthermore, it provides better distribution of work among sites. The difference would be even higher if we assumed slower communication and/or higher degree of fragmentation. ♦

6.2 Objectives of Query Processing

As stated before, the objective of query processing in a distributed context is to transform 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 on local databases. We assume that the high-level language is relational calculus, while the low-level language is an extension of relational algebra with communication operators. The different layers involved in the query transformation are detailed in Section 6.5. An important aspect 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.

A good measure of resource consumption is the *total cost* that will be incurred in processing the query [Sacco and Yao, 1982]. Total cost is the sum of all times incurred in processing the operators of the query at various sites and in intersite communication. Another good measure is the *response time* of the query [Epstein et al., 1978], which is the time elapsed for executing the query. Since operators

can be executed in parallel at different sites, the response time of a query may be significantly less than its total cost.

In a distributed database system, the total cost to be minimized includes CPU, I/O, and communication costs. The CPU cost is incurred when performing operators on data in main memory. The I/O cost is the time necessary for disk accesses. This cost can be minimized by reducing the number of disk accesses through fast access methods to the data and efficient use of main memory (buffer management). The communication cost is the time needed for exchanging data between sites participating in the execution of the query. This cost is incurred in processing the messages (formatting/deformatting), and in transmitting the data on the communication network.

The first two cost components (I/O and CPU cost) are the only factors considered by centralized DBMSs. The communication cost component is equally important factor considered in distributed databases. Most of the early proposals for distributed query optimization assume that the communication cost largely dominates local processing cost (I/O and CPU cost), and thus ignore the latter. This assumption is based on very slow communication networks (e.g., wide area networks that used to have a bandwidth of a few kilobytes per second) rather than on networks with bandwidths that are comparable to disk connection bandwidth. Therefore, the aim of distributed query optimization reduces to the problem of minimizing communication costs generally at the expense of local processing. The advantage is that local optimization can be done independently using the known methods for centralized systems. However, modern distributed processing environments have much faster communication networks, as discussed in Chapter 2, whose bandwidth is comparable to that of disks. Therefore, more recent research efforts consider a weighted combination of these three cost components since they all contribute significantly to the total cost of evaluating a query[1] [Page and Popek, 1985]. Nevertheless, in distributed environments with high bandwidths, the overhead cost incurred for communication between sites (e.g., software protocols) makes communication cost still an important factor.

6.3 Complexity of Relational Algebra Operations

In this chapter we consider relational algebra as a basis to express the output of query processing. Therefore, the complexity of relational algebra operators, which directly affects their execution time, dictates some principles useful to a query processor. These principles can help in choosing the final execution strategy.

The simplest way of defining complexity is in terms of relation cardinalities independent of physical implementation details such as fragmentation and storage

[1] There are some studies that investigate the feasibility of retrieving data from a neighboring nodes' main memory cache rather than accessing them from a local disk [Franklin et al., 1992; Dahlin et al., 1994; Freeley et al., 1995]. These approaches would have a significant impact on query optimization.

structures. Figure 6.2 shows the complexity of unary and binary operators in the order of increasing complexity, and thus of increasing execution time. Complexity is $O(n)$ for unary operators, where n denotes the relation cardinality, if the resulting tuples may be obtained independently of each other. Complexity is $O(n * \log n)$ for binary operators if each tuple of one relation must be compared with each tuple of the other on the basis of the equality of selected attributes. This complexity assumes that tuples of each relation must be sorted on the comparison attributes. However, using hashing and enough memory to hold one hashed relation can reduce the complexity of binary operators $O(n)$ [Bratbergsengen, 1984]. Projects with duplicate elimination and grouping operators require that each tuple of the relation be compared with each other tuple, and thus also have $O(n * \log n)$ complexity. Finally, complexity is $O(n^2)$ for the Cartesian product of two relations because each tuple of one relation must be combined with each tuple of the other.

Operation	Complexity
Select Project (without duplicate elimination)	$O(n)$
Project (with duplicate elimination) Group by	$O(n^*\log\ n)$
Join Semijoin Division Set Operators	$O(n^*\log\ n)$
Cartesian Product	$O(n^2)$

Fig. 6.2 Complexity of Relational Algebra Operations

This simple look at operator complexity suggests two principles. First, because complexity is relative to relation cardinalities, the most selective operators that reduce cardinalities (e.g., selection) should be performed first. Second, operators should be ordered by increasing complexity so that Cartesian products can be avoided or delayed.

6.4 Characterization of Query Processors

It is quite difficult to evaluate and compare query processors in the context of both centralized systems [Jarke and Koch, 1984] and distributed systems [Sacco and

Yao, 1982; Apers et al., 1983; Kossmann, 2000] because they may differ in many aspects. In what follows, we list important characteristics of query processors that can be used as a basis for comparison. The first four characteristics hold for both centralized and distributed query processors while the next four characteristics are particular to distributed query processors in tightly-integrated distributed DBMSs. This characterization is used in Chapter 8 to compare various algorithms.

6.4.1 Languages

Initially, most work on query processing was done in the context of relational DBMSs because their high-level languages give the system many opportunities for optimization. The input language to the query processor is thus based on relational calculus. With object DBMSs, the language is based on object calculus which is merely an extension of relational calculus. Thus, decomposition to object algebra is also needed (see Chapter 15). XML, another data model that we consider in this book, has its own languages, primarily in XQuery and XPath. Their execution requires special care that we discuss in Chapter 17.

The former requires an additional phase to decompose a query expressed in relational calculus into relational algebra. In a distributed context, the output language is generally some internal form of relational algebra augmented with communication primitives. The operators of the output language are implemented directly in the system. Query processing must perform efficient mapping from the input language to the output language.

6.4.2 Types of Optimization

Conceptually, query optimization aims at choosing the "best" point in the solution space of all possible execution strategies. An immediate method for query optimization is to search the solution space, exhaustively predict the cost of each strategy, and select the strategy with minimum cost. Although this method is effective in selecting the best strategy, it may incur a significant processing cost for the optimization itself. The problem is that the solution space can be large; that is, there may be many equivalent strategies, even with a small number of relations. The problem becomes worse as the number of relations or fragments increases (e.g., becomes greater than 5 or 6). Having high optimization cost is not necessarily bad, particularly if query optimization is done once for many subsequent executions of the query. Therefore, an "exhaustive" search approach is often used whereby (almost) all possible execution strategies are considered [Selinger et al., 1979].

To avoid the high cost of exhaustive search, *randomized* strategies, such as *iterative improvement* [Swami, 1989] and *simulated annealing* [Ioannidis and Wong, 1987]

have been proposed. They try to find a very good solution, not necessarily the best one, but avoid the high cost of optimization, in terms of memory and time consumption.

Another popular way of reducing the cost of exhaustive search is the use of heuristics, whose effect is to restrict the solution space so that only a few strategies are considered. In both centralized and distributed systems, a common heuristic is to minimize the size of intermediate relations. This can be done by performing unary operators first, and ordering the binary operators by the increasing sizes of their intermediate relations. An important heuristic in distributed systems is to replace join operators by combinations of semijoins to minimize data communication.

6.4.3 Optimization Timing

A query may be optimized at different times relative to the actual time of query execution. Optimization can be done *statically* before executing the query or *dynamically* as the query is executed. Static query optimization is done at query compilation time. Thus the cost of optimization may be amortized over multiple query executions. Therefore, this timing is appropriate for use with the exhaustive search method. Since the sizes of the intermediate relations of a strategy are not known until run time, they must be estimated using database statistics. Errors in these estimates can lead to the choice of suboptimal strategies.

Dynamic query optimization proceeds at query execution time. At any point of execution, the choice of the best next operator can be based on accurate knowledge of the results of the operators executed previously. Therefore, database statistics are not needed to estimate the size of intermediate results. However, they may still be useful in choosing the first operators. The main advantage over static query optimization is that the actual sizes of intermediate relations are available to the query processor, thereby minimizing the probability of a bad choice. The main shortcoming is that query optimization, an expensive task, must be repeated for each execution of the query. Therefore, this approach is best for ad-hoc queries.

Hybrid query optimization attempts to provide the advantages of static query optimization while avoiding the issues generated by inaccurate estimates. The approach is basically static, but dynamic query optimization may take place at run time when a high difference between predicted sizes and actual size of intermediate relations is detected.

6.4.4 Statistics

The effectiveness of query optimization relies on *statistics* on the database. Dynamic query optimization requires statistics in order to choose which operators should be done first. Static query optimization is even more demanding since the size of intermediate relations must also be estimated based on statistical information. In a

distributed database, statistics for query optimization typically bear on fragments, and include fragment cardinality and size as well as the size and number of distinct values of each attribute. To minimize the probability of error, more detailed statistics such as histograms of attribute values are sometimes used at the expense of higher management cost. The accuracy of statistics is achieved by periodic updating. With static optimization, significant changes in statistics used to optimize a query might result in query reoptimization.

6.4.5 Decision Sites

When static optimization is used, either a single site or several sites may participate in the selection of the strategy to be applied for answering the query. Most systems use the centralized decision approach, in which a single site generates the strategy. However, the decision process could be distributed among various sites participating in the elaboration of the best strategy. The centralized approach is simpler but requires knowledge of the entire distributed database, while the distributed approach requires only local information. Hybrid approaches where one site makes the major decisions and other sites can make local decisions are also frequent. For example, System R* [Williams et al., 1982] uses a hybrid approach.

6.4.6 Exploitation of the Network Topology

The network topology is generally exploited by the distributed query processor. With wide area networks, the cost function to be minimized can be restricted to the data communication cost, which is considered to be the dominant factor. This assumption greatly simplifies distributed query optimization, which can be divided into two separate problems: selection of the global execution strategy, based on intersite communication, and selection of each local execution strategy, based on a centralized query processing algorithm.

With local area networks, communication costs are comparable to I/O costs. Therefore, it is reasonable for the distributed query processor to increase parallel execution at the expense of communication cost. The broadcasting capability of some local area networks can be exploited successfully to optimize the processing of join operators [Özsoyoglu and Zhou, 1987; Wah and Lien, 1985]. Other algorithms specialized to take advantage of the network topology are discussed by Kerschberg et al. [1982] for star networks and by LaChimia [1984] for satellite networks.

In a client-server environment, the power of the client workstation can be exploited to perform database operators using *data shipping* [Franklin et al., 1996]. The optimization problem becomes to decide which part of the query should be performed on the client and which part on the server using query shipping.

6.4.7 Exploitation of Replicated Fragments

A distributed relation is usually divided into relation fragments as described in Chapter 3. Distributed queries expressed on global relations are mapped into queries on physical fragments of relations by translating relations into fragments. We call this process *localization* because its main function is to localize the data involved in the query. For higher reliability and better read performance, it is useful to have fragments replicated at different sites. Most optimization algorithms consider the localization process independently of optimization. However, some algorithms exploit the existence of replicated fragments at run time in order to minimize communication times. The optimization algorithm is then more complex because there are a larger number of possible strategies.

6.4.8 Use of Semijoins

The semijoin operator has the important property of reducing the size of the operand relation. When the main cost component considered by the query processor is communication, a semijoin is particularly useful for improving the processing of distributed join operators as it reduces the size of data exchanged between sites. However, using semijoins may result in an increase in the number of messages and in the local processing time. The early distributed DBMSs, such as SDD-1 [Bernstein et al., 1981], which were designed for slow wide area networks, make extensive use of semijoins. Some later systems, such as R* [Williams et al., 1982], assume faster networks and do not employ semijoins. Rather, they perform joins directly since using joins leads to lower local processing costs. Nevertheless, semijoins are still beneficial in the context of fast networks when they induce a strong reduction of the join operand. Therefore, some query processing algorithms aim at selecting an optimal combination of joins and semijoins [Özsoyoglu and Zhou, 1987; Wah and Lien, 1985].

6.5 Layers of Query Processing

In Chapter 1 we have seen where query processing fits within the distributed DBMS architecture. The problem of query processing can itself be decomposed into several subproblems, corresponding to various layers. In Figure 6.3 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

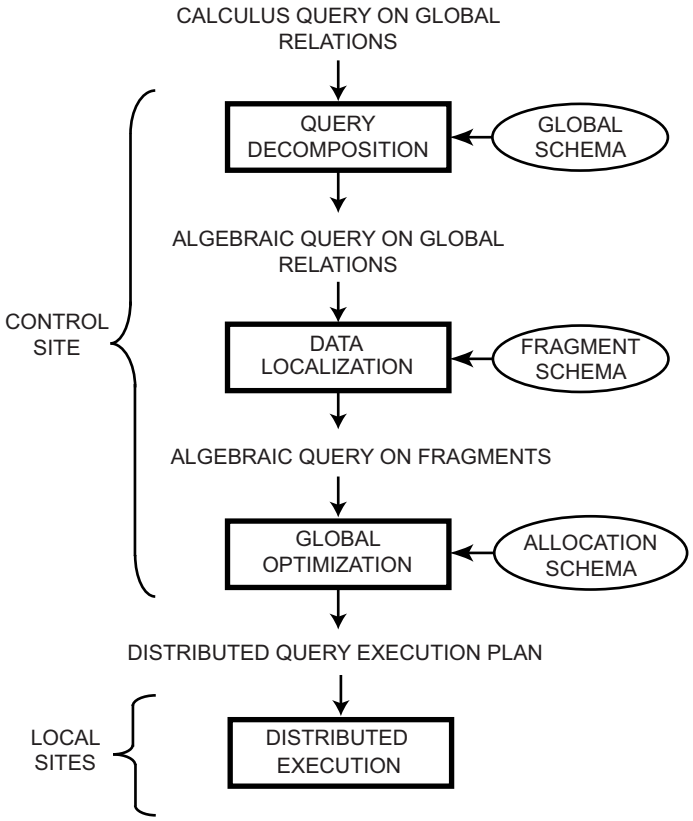

Fig. 6.3 Generic Layering Scheme for Distributed Query Processing

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. The first two layers are treated extensively in Chapter 7, while the two last layers are detailed in Chapter 8. In the remainder of this chapter we present an overview of these four layers.

6.5.1 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. As seen in Chapter 5, 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. Recall from Section 6.1 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.

6.5.2 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. In Chapter 3 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*), discussed in Chapter 3. 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.

6.5.3 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* .

6.5.4 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 (see Chapter 8).

6.6 Conclusion

In this chapter we provided an overview of query processing in distributed DBMSs. We first introduced the function and objectives of query processing. The main assumption is that the input query is expressed in relational calculus since that is the case with most current distributed DBMS. The complexity of the problem is proportional to the expressive power and the abstraction capability of the query language. For instance, the problem is even harder with important extensions such as the transitive closure operator [Valduriez and Boral, 1986].

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.

We gave a characterization of query processors based on their implementation choices. Query processors may differ in various aspects such as type of algorithm, optimization granularity, optimization timing, use of statistics, choice of decision site(s), exploitation of the network topology, exploitation of replicated fragments, and use of semijoins. This characterization is useful for comparing alternative query processor designs and to understand the trade-offs between efficiency and complexity.

The query processing problem is very difficult to understand in distributed environments because many elements are involved. However, the problem may be divided into several subproblems which are easier to solve individually. Therefore, we have proposed a generic layering scheme for describing distributed query processing. Four main functions have been isolated: query decomposition, data localization, global query optimization, and distributed query execution. These functions successively refine the query by adding more details about the processing environment. Query

decomposition and data localization are treated in detail in Chapter 7. Distributed query optimization and execution is the topic of Chapter 8.

6.7 Bibliographic Notes

Kim et al. [1985] provide a comprehensive set of papers presenting the results of research and development in query processing within the context of the relational model. After a survey of the state of the art in query processing, the book treats most of the important topics in the area. In particular, there are three papers on distributed query processing.

Ibaraki and Kameda [1984] have formally shown that finding the optimal execution strategy for a query is computationally intractable. Assuming a simplified cost function including the number of page accesses, it is proven that the minimization of this cost function for a multiple-join query is NP-complete.

Ceri and Pelagatti [1984] deal extensively with distributed query processing by treating the problem of localization and optimization separately in two chapters. The main assumption is that the query is expressed in relational algebra, so the decomposition phase that maps a calculus query into an algebraic query is ignored.

There are several survey papers on query processing and query optimization in the context of the relational model. A detailed survey is by Graefe [1993]. An earlier survey is [Jarke and Koch, 1984]. Both of these mainly deal with centralized query processing. The initial solutions to distributed query processing are extensively compiled in [Sacco and Yao, 1982; Yu and Chang, 1984]. Many query processing techniques are compiled in the book [Freytag et al., 1994].

The most complete survey on distributed query processing is by Kossmann [2000] and deals with both distributed DBMSs and multidatabase systems. The paper presents the traditional phases of query processing in centralized and distributed systems, and describes the various techniques for distributed query processing. It also discusses different distributed architectures such as client-server, multi-tier, and multidatabases.