Chapter 8
Optimization of Distributed Queries

Chapter 7 shows how a calculus query expressed on global relations can be mapped into a query on relation fragments by decomposition and data localization. This mapping uses the global and fragment schemas. During this process, the application of transformation rules permits the simplification of the query by eliminating common subexpressions and useless expressions. This type of optimization is independent of fragment characteristics such as cardinalities. The query resulting from decomposition and localization can be executed in that form simply by adding communication primitives in a systematic way. However, the permutation of the ordering of operations within the query can provide many equivalent strategies to execute it. Finding an "optimal" ordering of operations for a given query is the main role of the query optimization layer, or *optimizer* for short.

Selecting the optimal execution strategy for a query is NP-hard in the number of relations [Ibaraki and Kameda, 1984]. For complex queries with many relations, this can incur a prohibitive optimization cost. Therefore, the actual objective of the optimizer is to find a strategy close to optimal and, perhaps more important, to avoid bad strategies. In this chapter we refer to the strategy (or operation ordering) produced by the optimizer as the *optimal strategy* (or *optimal ordering*). The output of the optimizer is an optimized *query execution plan* consisting of the algebraic query specified on fragments and the communication operations to support the execution of the query over the fragment sites.

The selection of the optimal strategy generally requires the prediction of execution costs of the alternative candidate orderings prior to actually executing the query. The execution cost is expressed as a weighted combination of I/O, CPU, and communication costs. A typical simplification of the earlier distributed query optimizers was to ignore local processing cost (I/O and CPU costs) by assuming that the communication cost is dominant. Important inputs to the optimizer for estimating execution costs are fragment statistics and formulas for estimating the cardinalities of results of relational operations. In this chapter we focus mostly on the ordering of join operations for two reasons: it is a well-understood problem, and queries involving joins, selections, and projections are usually considered to be the most frequent type. Furthermore, it is easier to generalize the basic algorithm for other binary operations, such as union, intersection and difference. We also discuss how the semijoin operation can help to process join queries efficiently.

This chapter is organized as follows. In Section 8.1 we introduce the main components of query optimization, including the search space, the search strategy and the cost model. Query optimization in centralized systems is described in Section 8.2 as a prerequisite to understand distributed query optimization, which is more complex. In Section 8.3 we discuss the major optimization issue, which deals with the join ordering in distributed queries. We also examine alternative join strategies based on semijoin. In Section 8.4 we illustrate the use of the techniques and concepts in four basic distributed query optimization algorithms.

8.1 Query Optimization

This section introduces query optimization in general, i.e., independent of whether the environment is centralized or distributed. The input query is supposed to be expressed in relational algebra on database relations (which can obviously be fragments) after query rewriting from a calculus expression.

Query optimization refers to the process of producing a query execution plan (QEP) which represents an execution strategy for the query. This QEP minimizes an objective cost function. A query optimizer, the software module that performs query optimization, is usually seen as consisting of three components: a search space, a cost model, and a search strategy (see Figure 8.1). The *search space* is the set of alternative execution plans that represent the input query. These plans are equivalent, in the sense that they yield the same result, but they differ in the execution order of operations and the way these operations are implemented, and therefore in their performance. The search space is obtained by applying transformation rules, such as those for relational algebra described in Section 7.1.4. The *cost model* predicts the cost of a given execution plan. To be accurate, the cost model must have good knowledge about the distributed execution environment. The *search strategy* explores the search space and selects the best plan, using the cost model. It defines which plans are examined and in which order. The details of the environment (centralized versus distributed) are captured by the search space and the cost model.

8.1.1 Search Space

Query execution plans are typically abstracted by means of operator trees (see Section 7.1.4), which define the order in which the operations are executed. They are enriched with additional information, such as the best algorithm chosen for each operation. For a given query, the search space can thus be defined as the set of equivalent operator trees that can be produced using transformation rules. To characterize query optimizers, it is useful to concentrate on *join trees*, which are operator trees whose

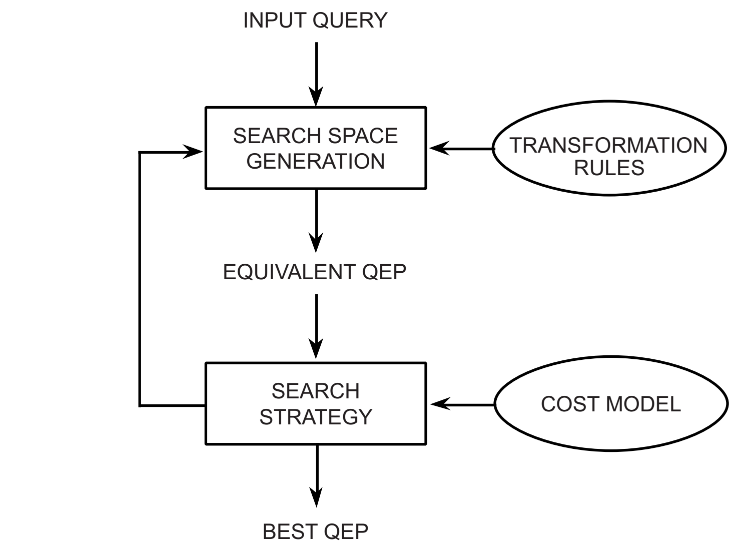


Fig. 8.1 Query Optimization Process

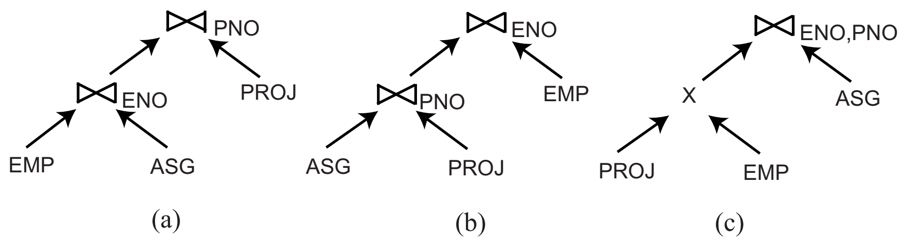


Fig. 8.2 Equivalent Join Trees

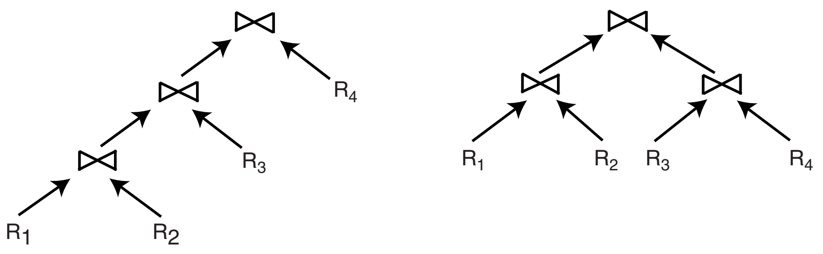
operators are join or Cartesian product. This is because permutations of the join order have the most important effect on performance of relational queries.

Example 8.1. Consider the following query:

```
SELECT ENAME, RESP
FROM   EMP, ASG, PROJ
WHERE  EMP.ENO=ASG.ENO
AND    ASG.PNO=PROJ.PNO
```

Figure 8.2 illustrates three equivalent join trees for that query, which are obtained by exploiting the associativity of binary operators. Each of these join trees can be assigned a cost based on the estimated cost of each operator. Join tree (c) which starts with a Cartesian product may have a much higher cost than the other join trees. ♦

For a complex query (involving many relations and many operators), the number of equivalent operator trees can be very high. For instance, the number of alternative



(a) linear join tree (b) bushy join tree

Fig. 8.3 The Two Major Shapes of Join Trees

join trees that can be produced by applying the commutativity and associativity rules is $O(N!)$ for N relations. Investigating a large search space may make optimization time prohibitive, sometimes much more expensive than the actual execution time. Therefore, query optimizers typically restrict the size of the search space they consider. The first restriction is to use heuristics. The most common heuristic is to perform selection and projection when accessing base relations. Another common heuristic is to avoid Cartesian products that are not required by the query. For instance, in Figure 8.2, operator tree (c) would not be part of the search space considered by the optimizer.

Another important restriction is with respect to the shape of the join tree. Two kinds of join trees are usually distinguished: linear versus bushy trees (see Figure 8.3). A *linear tree* is a tree such that at least one operand of each operator node is a base relation. A *bushy tree* is more general and may have operators with no base relations as operands (i.e., both operands are intermediate relations). By considering only linear trees, the size of the search space is reduced to $O(2^N)$. However, in a distributed environment, bushy trees are useful in exhibiting parallelism. For example, in join tree (b) of Figure 8.3, operations $R_1 \bowtie R_2$ and $R_3 \bowtie R_4$ can be done in parallel.

8.1.2 Search Strategy

The most popular search strategy used by query optimizers is *dynamic programming*, which is *deterministic*. Deterministic strategies proceed by *building* plans, starting from base relations, joining one more relation at each step until complete plans are obtained, as in Figure 8.4. Dynamic programming builds all possible plans, breadth-first, before it chooses the "best" plan. To reduce the optimization cost, partial plans that are not likely to lead to the optimal plan are *pruned* (i.e., discarded) as soon as possible. By contrast, another deterministic strategy, the greedy algorithm, builds only one plan, depth-first.

Dynamic programming is almost exhaustive and assures that the "best" of all plans is found. It incurs an acceptable optimization cost (in terms of time and space)

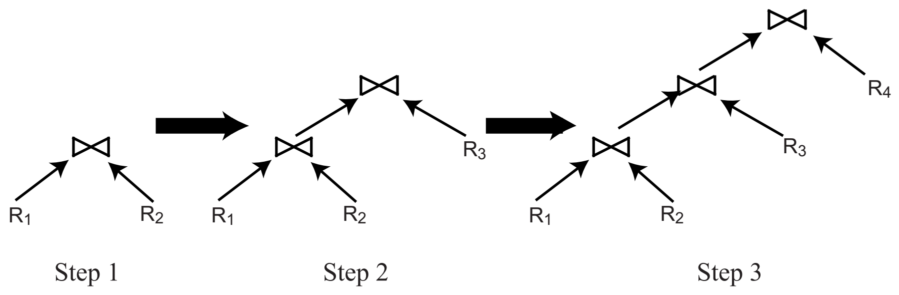


Fig. 8.4 Optimizer Actions in a Deterministic Strategy

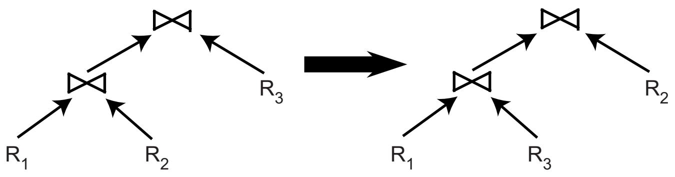


Fig. 8.5 Optimizer Action in a Randomized Strategy

when the number of relations in the query is small. However, this approach becomes too expensive when the number of relations is greater than 5 or 6. For more complex queries, *randomized* strategies have been proposed, which reduce the optimization complexity but do not guarantee the best of all plans. Unlike deterministic strategies, *randomized* strategies allow the optimizer to trade optimization time for execution time [Lanzelotte et al., 1993].

Randomized strategies, such as Simulated Annealing [Ioannidis and Wong, 1987] and Iterative Improvement [Swami, 1989] concentrate on searching for the optimal solution around some particular points. They do not guarantee that the best solution is obtained, but avoid the high cost of optimization, in terms of memory and time consumption. First, one or more *start* plans are built by a greedy strategy. Then, the algorithm tries to improve the start plan by visiting its *neighbors*. A neighbor is obtained by applying a random *transformation* to a plan. An example of a typical transformation consists in exchanging two randomly chosen operand relations of the plan, as in Figure 8.5. It has been shown experimentally that randomized strategies provide better performance than deterministic strategies as soon as the query involves more than several relations[Lanzelotte et al., 1993].

8.1.3 Distributed Cost Model

An optimizer's cost model includes cost functions to predict the cost of operators, statistics and base data, and formulas to evaluate the sizes of intermediate results.

The cost is in terms of execution time, so a cost function represents the execution time of a query.

8.1.3.1 Cost Functions

The cost of a distributed execution strategy can be expressed with respect to either the total time or the response time. The total time is the sum of all time (also referred to as cost) components, while the response time is the elapsed time from the initiation to the completion of the query. A general formula for determining the total time can be specified as follows [Lohman et al., 1985]:

$$Total\_time = T_{CPU} * \#insts + T_{I/O} * \#I/Os + T_{MSG} * \#msgs + T_{TR} * \#bytes$$

The two first components measure the local processing time, where T_{CPU} is the time of a CPU instruction and $T_{I/O}$ is the time of a disk I/O. The communication time is depicted by the two last components. T_{MSG} is the fixed time of initiating and receiving a message, while T_{TR} is the time it takes to transmit a data unit from one site to another. The data unit is given here in terms of bytes ($\#bytes$ is the sum of the sizes of all messages), but could be in different units (e.g., packets). A typical assumption is that T_{TR} is constant. This might not be true for wide area networks, where some sites are farther away than others. However, this assumption greatly simplifies query optimization. Thus the communication time of transferring $\#bytes$ of data from one site to another is assumed to be a linear function of $\#bytes$:

$$CT(\#bytes) = T_{MSG} + T_{TR} * \#bytes$$

Costs are generally expressed in terms of time units, which in turn, can be translated into other units (e.g., dollars).

The relative values of the cost coefficients characterize the distributed database environment. The topology of the network greatly influences the ratio between these components. In a wide area network such as the Internet, the communication time is generally the dominant factor. In local area networks, however, there is more of a balance among the components. Earlier studies cite ratios of communication time to I/O time for one page to be on the order of 20:1 for wide area networks [Selinger and Adiba, 1980] while it is 1:1.6 for a typical early generation Ethernet (10Mbps) [Page and Popek, 1985]. Thus, most early distributed DBMSs designed for wide area networks have ignored the local processing cost and concentrated on minimizing the communication cost. Distributed DBMSs designed for local area networks, on the other hand, consider all three cost components. The new faster networks, both at the wide area network and at the local area network levels, have improved the above ratios in favor of communication cost when all things are equal. However, communication is still the dominant time factor in wide area networks such as the Internet because of the longer distances that data are retrieved from (or shipped to).

When the response time of the query is the objective function of the optimizer, parallel local processing and parallel communications must also be considered

[Khoshafian and Valduriez, 1987]. A general formula for response time is

$$\begin{aligned} Response\_time = & T_{CPU} * seq\_\#insts + T_{I/O} * seq\_\#I/Os \\ & + T_{MSG} * seq\_\#msgs + T_{TR} * seq\_\#bytes \end{aligned}$$

where $seq\_\#x$, in which x can be instructions ($insts$), I/O, messages ($msgs$) or $bytes$, is the maximum number of x which must be done sequentially for the execution of the query. Thus any processing and communication done in parallel is ignored.

Example 8.2. Let us illustrate the difference between total cost and response time using the example of Figure 8.6, which computes the answer to a query at site 3 with data from sites 1 and 2. For simplicity, we assume that only communication cost is considered.

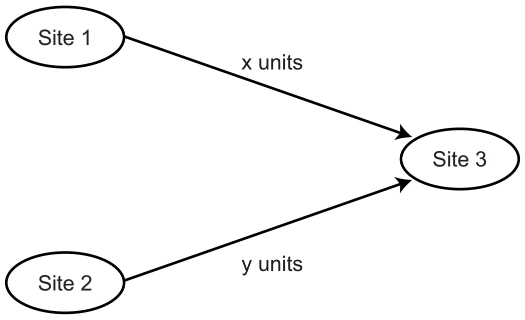


Fig. 8.6 Example of Data Transfers for a Query

Assume that $T_{MSG}$and T_{TR} are expressed in time units. The total time of transferring x data units from site 1 to site 3 and y data units from site 2 to site 3 is

$$Total\_time = 2\ T_{MSG} + T_{TR} * (x + y)$$

The response time of the same query can be approximated as

$$Response\_time = max\{T_{MSG} + T_{TR} * x, T_{MSG} + T_{TR} * y\}$$

since the transfers can be done in parallel. ♦

Minimizing response time is achieved by increasing the degree of parallel execution. This does not, however, imply that the total time is also minimized. On the contrary, it can increase the total time, for example, by having more parallel local processing and transmissions. Minimizing the total time implies that the utilization of the resources improves, thus increasing the system throughput. In practice, a compromise between the two is desired. In Section 8.4 we present algorithms that can optimize a combination of total time and response time, with more weight on one of them.

8.1.3.2 Database Statistics

The main factor affecting the performance of an execution strategy is the size of the intermediate relations that are produced during the execution. When a subsequent operation is located at a different site, the intermediate relation must be transmitted over the network. Therefore, it is of prime interest to estimate the size of the intermediate results of relational algebra operations in order to minimize the size of data transfers. This estimation is based on statistical information about the base relations and formulas to predict the cardinalities of the results of the relational operations. There is a direct trade-off between the precision of the statistics and the cost of managing them, the more precise statistics being the more costly [Piatetsky-Shapiro and Connell, 1984]. For a relation R defined over the attributes $A = \{A_1, A_2, \ldots, A_n\}$ and fragmented as $R_1, R_2, \ldots, R_r$, the statistical data typically are the following:

1. For each attribute A_i, its length (in number of bytes), denoted by $length(A_i)$, and for each attribute A_i of each fragment R_j, the number of distinct values of A_i, with the cardinality of the projection of fragment R_j on A_i, denoted by $card(\Pi_{A_i}(R_j))$.
2. For the domain of each attribute A_i, which is defined on a set of values that can be ordered (e.g., integers or reals), the minimum and maximum possible values, denoted by $min(A_i)$ and $max(A_i)$.
3. For the domain of each attribute A_i, the cardinality of the domain of A_i, denoted by $card(dom[A_i])$. This value gives the number of unique values in the $dom[A_i]$.
4. The number of tuples in each fragment R_j, denoted by $card(R_j)$.

In addition, for each attribute A_i, there may be a histogram that approximates the frequency distribution of the attribute within a number of buckets, each corresponding to a range of values.

Sometimes, the statistical data also include the join selectivity factor for some pairs of relations, that is the proportion of tuples participating in the join. The *join selectivity factor*, denoted SF_J, of relations R and S is a real value between 0 and 1:

$$SF_J(R,S) = \frac{card(R \bowtie S)}{card(R) * card(S)}$$

For example, a join selectivity factor of 0.5 corresponds to a very large joined relation, while 0.001 corresponds to a small one. We say that the join has bad (or low) selectivity in the former case and good (or high) selectivity in the latter case.

These statistics are useful to predict the size of intermediate relations. Remember that in Chapter 3 we defined the size of an intermediate relation R as follows:

$$size(R) = card(R) * length(R)$$

where $length(R)$ is the length (in bytes) of a tuple of R, computed from the lengths of its attributes. The estimation of $card(R)$, the number of tuples in R, requires the use of the formulas given in the following section.

8.1.3.3 Cardinalities of Intermediate Results

Database statistics are useful in evaluating the cardinalities of the intermediate results of queries. Two simplifying assumptions are commonly made about the database. The distribution of attribute values in a relation is supposed to be uniform, and all attributes are independent, meaning that the value of an attribute does not affect the value of any other attribute. These two assumptions are often wrong in practice, but they make the problem tractable. In what follows we give the formulas for estimating the cardinalities of the results of the basic relational algebra operations (selection, projection, Cartesian product, join, semijoin, union, and difference). The operand relations are denoted by R and S. The *selectivity factor* of an operation, that is, the proportion of tuples of an operand relation that participate in the result of that operation, is denoted SF_{OP}, where OP denotes the operation.

Selection.

The cardinality of selection is

$$card(\sigma_F(R)) = SF_S(F) * card(R)$$

where $SF_S(F)$ is dependent on the selection formula and can be computed as follows [Selinger et al., 1979], where $p(A_i)$ and $p(A_j)$ indicate predicates over attributes A_i and A_j, respectively:

$$SF_S(A = value) = \frac{1}{card(\Pi_A(R))}$$

$$SF_S(A > value) = \frac{max(A) - value}{max(A) - min(A)}$$

$$SF_S(A < value) = \frac{value - min(A)}{max(A) - min(A)}$$

$$SF_S(p(A_i) \wedge p(A_j)) = SF_S(p(A_i)) * SF_S(p(A_j))$$

$$SF_S(p(A_i) \vee p(A_j)) = SF_S(p(A_i)) + SF_S(p(A_j)) - (SF_S(p(A_i)) * SF_S(p(A_j)))$$

$$SF_S(A \in \{values\}) = SF_S(A = value) * card(\{values\})$$

Projection.

As indicated in Section 2.1, projection can be with or without duplicate elimination. We consider projection with duplicate elimination. An arbitrary projection is difficult to evaluate precisely because the correlations between projected attributes are usually unknown [Gelenbe and Gardy, 1982]. However, there are two particularly useful cases where it is trivial. If the projection of relation R is based on a single attribute A, the cardinality is simply the number of tuples when the projection is performed. If one of the projected attributes is a key of R, then

$$card(\Pi_A(R)) = card(R)$$

Cartesian product.

The cardinality of the Cartesian product of R and S is simply

$$card(R \times S) = card(R) * card(S)$$

Join.

There is no general way to estimate the cardinality of a join without additional information. The upper bound of the join cardinality is the cardinality of the Cartesian product. It has been used in the earlier distributed DBMS (e.g. [Epstein et al., 1978]), but it is a quite pessimistic estimate. A more realistic solution is to divide this upper bound by a constant to reflect the fact that the join result is smaller than that of the Cartesian product [Selinger and Adiba, 1980]. However, there is a case, which occurs frequently, where the estimation is simple. If relation R is equijoined with S over attribute A from R, and B from S, where A is a key of relation R, and B is a foreign key of relation S, the cardinality of the result can be approximated as

$$card(R \bowtie_{A=B} S) = card(S)$$

because each tuple of S matches with at most one tuple of R. Obviously, the same thing is true if B is a key of S and A is a foreign key of R. However, this estimation is an upper bound since it assumes that each tuple of R participates in the join. For other important joins, it is worthwhile to maintain their join selectivity factor SF_J as part of statistical information. In that case the result cardinality is simply

$$card(R \bowtie S) = SF_J * card(R) * card(S)$$

Semijoin.

The selectivity factor of the semijoin of R by S gives the fraction (percentage) of tuples of R that join with tuples of S. An approximation for the semijoin selectivity factor is given by Hevner and Yao [1979] as

$$SF_{SJ}(R \ltimes_A S) = \frac{card(\Pi_A(S))}{card(dom[A])}$$

This formula depends only on attribute A of S. Thus it is often called the selectivity factor of attribute A of S, denoted $SF_{SJ}(S.A)$, and is the selectivity factor of $S.A$ on any other joinable attribute. Therefore, the cardinality of the semijoin is given by

$$card(R \ltimes_A S) = SF_{SJ}(S.A) * card(R)$$

This approximation can be verified on a very frequent case, that of $R.A$ being a foreign key of S ($S.A$ is a primary key). In this case, the semijoin selectivity factor is 1 since $\Pi_A(S)) = card(dom[A])$ yielding that the cardinality of the semijoin is $card(R)$.

Union.

It is quite difficult to estimate the cardinality of the union of R and S because the duplicates between R and S are removed by the union. We give only the simple formulas for the upper and lower bounds, which are, respectively,

$$card(R) + card(S)$$
$$max\{card(R), card(S)\}$$

Note that these formulas assume that R and S do not contain duplicate tuples.

Difference.

Like the union, we give only the upper and lower bounds. The upper bound of $card(R - S)$ is $card(R)$, whereas the lower bound is 0.

More complex predicates with conjunction and disjunction can also be handled by using the formulas given above.

8.1.3.4 Using Histograms for Selectivity Estimation

The formulae above for estimating the cardinalities of intermediate results of queries rely on the strong assumption that the distribution of attribute values in a relation is uniform. The advantage of this assumption is that the cost of managing the statistics is minimal since only the number of distinct attribute values is needed. However, this assumption is not practical. In case of skewed data distributions, it can result in fairly inaccurate estimations and QEPs which are far from the optimal.

An effective solution to accurately capture data distributions is to use histograms. Today, most commercial DBMS optimizers support histograms as part of their cost model. Various kinds of histograms have been proposed for estimating the selectivity of query predicates with different trade-offs between accuracy and maintenance cost [Poosala et al., 1996]. To illustrate the use of histograms, we use the basic definition by Bruno and Chaudhuri [2002]. A *histogram* on attribute A from R is a set of buckets. Each bucket b_i describes a range of values of A, denoted by $range_i$, with its associated frequency f_i and number of distinct values d_i. f_i gives the number of tuples of R where $R.A \in range_i$. d_i gives the number of distinct values of A where $R.A \in range_i$. This representation of a relation's attribute can capture non-uniform distributions of values, with the buckets adapted to the different ranges. However, within a bucket, the distribution of attribute values is assumed to be uniform.

Histograms can be used to accurately estimate the selectivity of selection operations. They can also be used for more complex queries including selection, projection and join. However, the precise estimation of join selectivity remains difficult and depends on the type of the histogram [Poosala et al., 1996]. We now illustrate the use of histograms with two important selection predicates: equality and range predicate.

Equality predicate.

With $value \in range_i$, we simply have: $SF_S(A = value) = 1/d_i$.

Range predicate.

Computing the selectivity of range predicates such as $A \leq value$, $A < value$ and $A > value$ requires identifying the relevant buckets and summing up their frequencies. Let us consider the range predicate $R.A \leq value$ with $value \in range_i$. To estimate the numbers of tuples of R that satisfy this predicate, we must sum up the frequencies of all buckets which precede bucket i and the estimated number of tuples that satisfy the predicate in bucket b_i. Assuming uniform distribution of attribute values in b_i, we have:

$$card(\sigma_{A \leq value}(R)) = \sum_{j=1}^{i-1} f_j + (\frac{value - min(range_i)}{min(range_i)} \cancel{- min(range_i}) * f_i)$$

The cardinality of other range predicates can be computed in a similar way.

Example 8.3. Figure 8.7 shows a possible 4-bucket histogram for attribute DUR of a relation ASG with 300 tuples. Let us consider the equality predicate ASG.DUR=18. Since the value "18" fits in bucket b_3, the selectivity factor is 1/12. Since the cardinalty of b_3 is 50, the cardinality of the selection is 50/12 which is approximately 5 tuples. Let us now consider the range predicate ASG.DUR $\leq$ 18. We have $min(range_3) = 12$ and $max(range_3) = 24$. The cardinality of the selection is: $100 + 75 + (((18 - 12)/(24 - 12)) * 50) = 200$ tuples. ♦

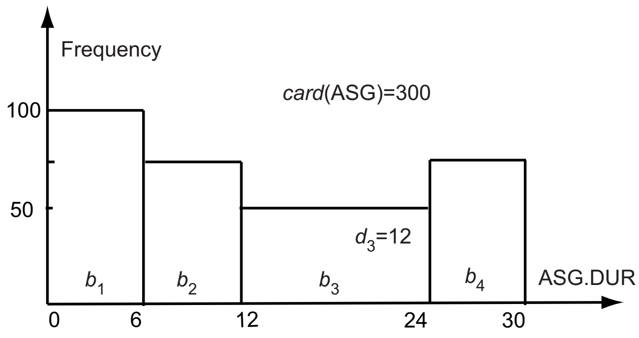


Fig. 8.7 Histogram of Attribute ASG.DUR

8.2 Centralized Query Optimization

In this section we present the main query optimization techniques for centralized systems. This presentation is a prerequisite to understanding distributed query optimization for three reasons. First, a distributed query is translated into local queries, each of which is processed in a centralized way. Second, distributed query optimization techniques are often extensions of the techniques for centralized systems. Finally, centralized query optimization is a simpler problem; the minimization of communication costs makes distributed query optimization more complex.

As discussed in Chapter 6, the optimization timing, which can be dynamic, static or hybrid, is a good basis for classifying query optimization techniques. Therefore, we present a representative technique of each class.

8.2.1 Dynamic Query Optimization

Dynamic query optimization combines the two phases of query decomposition and optimization with execution. The QEP is dynamically constructed by the query optimizer which makes calls to the DBMS execution engine for executing the query's operations. Thus, there is no need for a cost model.

The most popular dynamic query optimization algorithm is that of INGRES [Stonebraker et al., 1976], one of the first relational DBMS. In this section, we present this algorithm based on the detailed description by Wong and Youssefi [1976]. The algorithm recursively breaks up a query expressed in relational calculus (i.e., SQL) into smaller pieces which are executed along the way. The query is first decomposed into a sequence of queries having a unique relation in common. Then each monorelation query is processed by selecting, based on the predicate, the best access method to that relation (e.g., index, sequential scan). For example, if the predicate is of the form $A = value$, an index available on attribute A would be used if it exists. However, if the predicate is of the form $A \neq value$, an index on A would not help, and sequential scan should be used.

The algorithm executes first the unary (monorelation) operations and tries to minimize the sizes of intermediate results in ordering binary (multirelation) operations. Let us denote by $q_{i-1} \rightarrow q_i$ a query q decomposed into two subqueries, q_{i-1} and q_i, where q_{i-1} is executed first and its result is consumed by q_i. Given an n-relation query q, the optimizer decomposes q into n subqueries $q_1 \rightarrow q_2 \rightarrow \cdots \rightarrow q_n$. This decomposition uses two basic techniques: *detachment* and *substitution*. These techniques are presented and illustrated in the rest of this section.

Detachment is the first technique employed by the query processor. It breaks a query q into $q' \rightarrow q''$, based on a common relation that is the result of q'. If the query q expressed in SQL is of the form

```
SELECT $R_2.A_2, R_3.A_3, \ldots, R_n.A_n$
FROM   $R_1, R_2, \ldots, R_n$
WHERE  $P_1(R_1.A'_1)$
AND    $P_2(R_1.A_1, R_2.A_2, \ldots, R_n.A_n)$
```

where A_i and A'_i are lists of attributes of relation R_i, P_1 is a predicate involving attributes from relation R_1, and P_2 is a multirelation predicate involving attributes of relations $R_1, R_2, \ldots, R_n$. Such a query may be decomposed into two subqueries, q' followed by q'', by detachment of the common relation R_1:

```
$q'$: SELECT $R_1.A_1$ INTO $R'_1$
      FROM   $R_1$
      WHERE  $P_1(R_1.A'_1)$
```

where R'_1 is a temporary relation containing the information necessary for the continuation of the query:

```
$q''$: SELECT $R_2.A_2, \ldots, R_n.A_n$
       FROM   $R'_1, R_2, \ldots, R_n$
       WHERE  $P_2(R'_1.A_1, \ldots, R_n.A_n)$
```

This step has the effect of reducing the size of the relation on which the query q'' is defined. Furthermore, the created relation R'_1 may be stored in a particular structure to speed up the following subqueries. For example, the storage of R'_1 in a hashed file on the join attributes of q'' will make processing the join more efficient. Detachment extracts the select operations, which are usually the most selective ones. Therefore, detachment is systematically done whenever possible. Note that this can have adverse effects on performance if the selection has bad selectivity.

Example 8.4. To illustrate the detachment technique, we apply it to the following query:

"Names of employees working on the CAD/CAM project"

This query can be expressed in SQL by the following query q_1 on the engineering database of Chapter 2:

```
q1: SELECT EMP.ENAME
    FROM   EMP, ASG, PROJ
    WHERE  EMP.ENO=ASG.ENO
    AND    ASG.PNO=PROJ.PNO
    AND    PNAME="CAD/CAM"
```

After detachment of the selections, query q_1 is replaced by q_{11} followed by q', where JVAR is an intermediate relation.

```
q11: SELECT PROJ.PNO INTO JVAR
     FROM   PROJ
     WHERE  PNAME="CAD/CAM"
q':  SELECT EMP.ENAME
     FROM   EMP, ASG, JVAR
     WHERE  EMP.ENO=ASG.ENO
     AND    ASG.PNO=JVAR.PNO
```

The successive detachments of q' may generate

```
q12: SELECT ASG.ENO INTO GVAR
     FROM   ASG, JVAR
     WHERE  ASG.PNO=JVAR.PNO
q13: SELECT EMP.ENAME
     FROM   EMP, GVAR
     WHERE  EMP.ENO=GVAR.ENO
```

Note that other subqueries are also possible.

Thus query q_1 has been reduced to the subsequent queries $q_{11} \rightarrow q_{12} \rightarrow q_{13}$. Query q_{11} is monorelation and can be executed. However, q_{12} and q_{13} are not monorelation and cannot be reduced by detachment. ♦

Multirelation queries, which cannot be further detached (e.g., q_{12} and q_{13}), are *irreducible*. A query is irreducible if and only if its query graph is a chain with two nodes or a cycle with k nodes where $k > 2$. Irreducible queries are converted into monorelation queries by tuple substitution. Given an n-relation query q, the tuples of one relation are substituted by their values, thereby producing a set of $(n-1)$-relation queries. Tuple substitution proceeds as follows. First, one relation in q is chosen for tuple substitution. Let R_1 be that relation. Then for each tuple t_{1i} in R_1, the attributes referred to by in q are replaced by their actual values in t_{1i}, thereby generating a query q' with $n-1$ relations. Therefore, the total number of queries q' produced by tuple substitution is $card(R_1)$. Tuple substitution can be summarized as follows:

$$q(R_1, R_2, \ldots, R_n) \text{ is replaced by } \{q'(t_{1i}, R_2, R_3, \ldots, R_n), t_{1i} \in R_1\}$$

For each tuple thus obtained, the subquery is recursively processed by substitution if it is not yet irreducible.

Example 8.5. Let us consider the query q_{13}:

```
SELECT EMP.ENAME
FROM   EMP, GVAR
WHERE  EMP.ENO=GVAR.ENO
```

The relation GVAR is over a single attribute (ENO). Assume that it contains only two tuples: ⟨E1⟩ and ⟨E2⟩. The substitution of GVAR generates two one-relation subqueries:

```
q131: SELECT EMP.ENAME
      FROM   EMP
      WHERE  EMP.ENO="E1"
q132: SELECT EMP.ENAME
      FROM   EMP
      WHERE  EMP.ENO="E2"
```

These queries may then be executed. ♦

This dynamic query optimization algorithm (called Dynamic-QOA) is depicted in Algorithm 8.1. The algorithm works recursively until there remain no more monorelation queries to be processed. It consists of applying the selections and projections as soon as possible by detachment. The results of the monorelation queries are stored in data structures that are capable of optimizing the later queries (such as joins). The irreducible queries that remain after detachment must be processed by tuple substitution. For the irreducible query, denoted by MRQ', the smallest relation whose cardinality is known from the result of the preceding query is chosen for substitution. This simple method enables one to generate the smallest number of subqueries. Monorelation queries generated by the reduction algorithm are executed after choosing the best existing access path to the relation, according to the query qualification.

Algorithm 8.1: Dynamic-QOA

```
Input: $MRQ$: multirelation query with $n$ relations
Output: $output$: result of execution
begin
    $output \leftarrow \phi$ ;
    if $n = 1$ then
        $output \leftarrow run(MRQ)$                         {execute the one relation query}
    {detach $MRQ$ into $m$ one-relation queries (ORQ) and one multirelation
    query} $ORQ_1, \dots, ORQ_m, MRQ' \leftarrow MRQ$ ;
    for $i$ from 1 to $m$ do
        $output' \leftarrow run(ORQ_i)$ ;                     {execute $ORQ_i$}
        $output \leftarrow output \cup output'$              {merge all results}
    $R \leftarrow$ CHOOSE_RELATION($MRQ'$) ;      {$R$ chosen for tuple substitution}
    for each tuple $t \in R$ do
        $MRQ'' \leftarrow$ substitute values for $t$ in $MRQ'$ ;
        $output' \leftarrow$ Dynamic-QOA($MRQ''$) ;           {recursive call}
        $output \leftarrow output \cup output'$              {merge all results}
end
```

8.2.2 Static Query Optimization

With static query optimization, there is a clear separation between the generation of the QEP at compile-time and its execution by the DBMS execution engine. Thus, an accurate cost model is key to predict the costs of candidate QEPs.

The most popular static query optimization algorithm is that of System R [Astrahan et al., 1976], also one of the first relational DBMS. In this section, we present this algorithm based on the description by Selinger et al. [1979]. Most commercial relational DBMSs have implemented variants of this algorithm due to its efficiency and compatibility with query compilation.

The input to the optimizer is a relational algebra tree resulting from the decomposition of an SQL query. The output is a QEP that implements the "optimal" relational algebra tree.

The optimizer assigns a cost (in terms of time) to every candidate tree and retains the one with the smallest cost. The candidate trees are obtained by a permutation of the join orders of the n relations of the query using the commutativity and associativity rules. To limit the overhead of optimization, the number of alternative trees is reduced using dynamic programming. The set of alternative strategies is constructed dynamically so that, when two joins are equivalent by commutativity, only the cheapest one is kept. Furthermore, the strategies that include Cartesian products are eliminated whenever possible.

The cost of a candidate strategy is a weighted combination of I/O and CPU costs (times). The estimation of such costs (at compile time) is based on a cost model that provides a cost formula for each low-level operation (e.g., select using a B-tree index with a range predicate). For most operations (except exact match select), these cost formulas are based on the cardinalities of the operands. The cardinality information for the relations stored in the database is found in the database statistics. The cardinality of the intermediate results is estimated based on the operation selectivity factors discussed in Section 8.1.3.

The optimization algorithm consists of two major steps. First, the best access method to each individual relation based on a select predicate is predicted (this is the one with the least cost). Second, for each relation R, the best join ordering is estimated, where R is first accessed using its best single-relation access method. The cheapest ordering becomes the basis for the best execution plan.

In considering the joins, there are two basic algorithms available, with one of them being optimal in a given context. For the join of two relations, the relation whose tuples are read first is called the *external*, while the other, whose tuples are found according to the values obtained from the external relation, is called the *internal relation*. An important decision with either join method is to determine the cheapest access path to the internal relation.

The first method, called *nested-loop*, performs two loops over the relations. For each tuple of the external relation, the tuples of the internal relation that satisfy the join predicate are retrieved one by one to form the resulting relation. An index or a hashed table on the join attribute is a very efficient access path for the internal relation. In the absence of an index, for relations of n_1 and n_2 tuples, respectively, this algorithm has a cost proportional to $n_1 * n_2$, which may be prohibitive if n_1 and n_2 are high. Thus, an efficient variant is to build a hashed table on the join attribute for the internal relation (chosen as the smallest relation) before applying nested-loop. If the internal relation is itself the result of a previous operation, then the cost of building the hashed table can be shared with that of producing the previous result.

The second method, called *merge-join*, consists of merging two sorted relations on the join attribute. Indices on the join attribute may be used as access paths. If the join criterion is equality, the cost of joining two relations of n_1 and n_2 tuples, respectively, is proportional to $n_1 + n_2$. Therefore, this method is always chosen when there is an equijoin, and when the relations are previously sorted. If only one or neither of the relations are sorted, the cost of the nested-loop algorithm is to be compared with the combined cost of the merge join and of the sorting. The cost of sorting n pages is proportional to $n \log n$. In general, it is useful to sort and apply the merge join algorithm when large relations are considered.

The simplified version of the static optimization algorithm, for a select-project-join query, is shown in Algorithm 8.2. It consists of two loops, the first of which selects the best single-relation access method to each relation in the query, while the second examines all possible permutations of join orders (there are $n!$ permutations with n relations) and selects the best access strategy for the query. The permutations are produced by the dynamic construction of a tree of alternative strategies. First, the join of each relation with every other relation is considered, followed by joins of three relations. This continues until joins of n relations are optimized. Actually, the algorithm does not generate all possible permutations since some of them are useless.

As we discussed earlier, permutations involving Cartesian products are eliminated, as are the commutatively equivalent strategies with the highest cost. With these two heuristics, the number of strategies examined has an upper bound of 2^n rather than $n!$.

Algorithm 8.2: Static-QOA

```
Input: $QT$: query tree with $n$ relations
Output: $output$: best QEP
begin
    for each relation $R_i \in QT$ do
        for each access path $AP_{ij}$ to $R_i$ do
            compute cost($AP_{ij}$)
        $best\_AP_i \leftarrow AP_{ij}$ with minimum cost ;
        for each order $(R_{i1}, R_{i2}, \cdots, R_{in})$ with $i = 1, \cdots, n!$ do
            build QEP $(\ldots((\text{best } AP_{i1} \bowtie R_{i2}) \bowtie R_{i3}) \bowtie \ldots \bowtie R_{in})$ ;
            compute cost (QEP)
        $output \leftarrow$ QEP with minimum cost
end
```

Example 8.6. Let us illustrate this algorithm with the query q_1 (see Example 8.4) on the engineering database. The join graph of q_1 is given in Figure 8.8. For short, the label ENO on edge EMP–ASG stands for the predicate EMP.ENO=ASG.ENO and the label PNO on edge ASG–PROJ stands for the predicate ASG.PNO=PROJ.PNO. We assume the following indices:

EMP has an index on ENO
ASG has an index on PNO
PROJ has an index on PNO and an index on PNAME

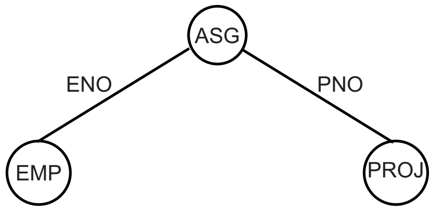


Fig. 8.8 Join Graph of Query q_1

We assume that the first loop of the algorithm selects the following best single-relation access paths:

EMP:	sequential scan (because there is no selection on EMP)
ASG:	sequential scan (because there is no selection on ASG)
PROJ:	index on PNAME (because there is a selection on PROJ based on PNAME)

The dynamic construction of the tree of alternative strategies is illustrated in Figure 8.9. Note that the maximum number of join orders is 3!; dynamic search considers fewer alternatives, as depicted in Figure 8.9. The operations marked "pruned" are dynamically eliminated. The first level of the tree indicates the best single-relation access method. The second level indicates, for each of these, the best join method with any other relation. Strategies (EMP $\times$ PROJ) and (PROJ $\times$ EMP) are pruned because they are Cartesian products that can be avoided (by other strategies). We assume that (EMP $\bowtie$ ASG) and (ASG $\bowtie$ PROJ) have a cost higher than (ASG $\bowtie$ EMP) and (PROJ $\bowtie$ ASG), respectively. Thus they can be pruned because there are better join orders equivalent by commutativity. The two remaining possibilities are given at the third level of the tree. The best total join order is the least costly of ((ASG $\bowtie$ EMP) $\bowtie$ PROJ) and ((PROJ $\bowtie$ ASG) $\bowtie$ EMP). The latter is the only one that has a useful index on the select attribute and direct access to the joining tuples of ASG and EMP. Therefore, it is chosen with the following access methods:

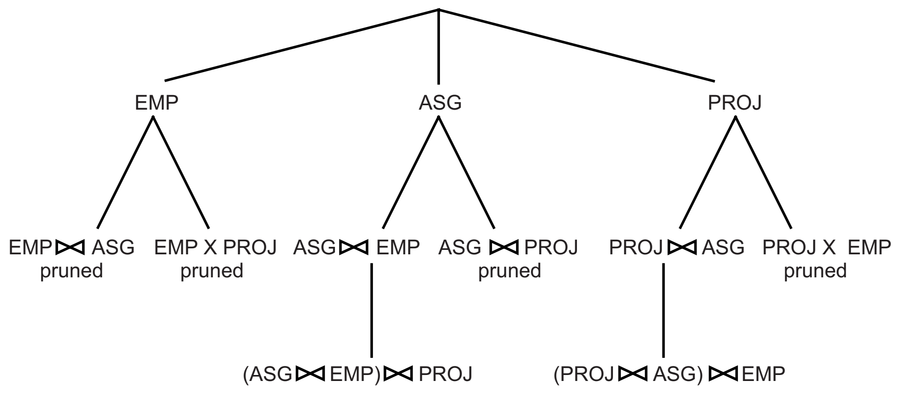


Fig. 8.9 Alternative Join Orders

Select PROJ using index on PNAME
Then join with ASG using index on PNO
Then join with EMP using index on ENO

♦

The performance measurements substantiate the important contribution of the CPU time to the total time of the query[Mackert and Lohman, 1986]. The accuracy of the optimizer's estimations is generally good when the relations can be contained in the main memory buffers, but degrades as the relations increase in size and are written to disk. An important performance parameter that should also be considered for better predictions is buffer utilization.

8.2.3 Hybrid Query Optimization

Dynamic and static query optimimization both have advantages and drawbacks. Dynamic query optimization mixes optimization and execution and thus can make accurate optimization choices at run-time. However, query optimization is repeated for each execution of the query. Therefore, this approach is best for ad-hoc queries. Static query optimization, done at compilation time, amortizes the cost of optimization over multiple query executions. The accuracy of the cost model is thus critical to predict the costs of candidate QEPs. This approach is best for queries embedded in stored procedures, and has been adopted by all commercial DBMSs.

However, even with a sophisticated cost model, there is an important problem that prevents accurate cost estimation and comparison of QEPs at compile-time. The problem is that the actual bindings of parameter values in embedded queries is not known until run-time. Consider for instance the selection predicate “WHERE $R.A = \$a$” where “$\$a$” is a parameter value. To estimate the cardinality of this selection, the optimizer must rely on the assumption of uniform distribution of A values in R and cannot make use of histograms. Since there is a runtime binding of the parameter a, the accurate selectivity of $\sigma_{A=\$a}(R)$ cannot be estimated until runtime.

Thus, it can make major estimation errors that can lead to the choice of suboptimal QEPs.

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 further optimization decisions may take place at run time. This approach was pionnered in System R by adding a conditional runtime reoptimization phase for execution plans statically optimized [Chamberlin et al., 1981]. Thus, plans that have become infeasible (e.g., because indices have been dropped) or suboptimal (e.g. because of changes in relation sizes) are reoptimized. However, detecting suboptimal plans is hard and this approach tends to perform much more reoptimization than necessary. A more general solution is to produce *dynamic QEPs* which include carefully selected optimization decisions to be made at runtime using “choose-plan” operators [Cole and Graefe, 1994]. The choose-plan operator links two or more equivalent subplans of a QEP that are incomparable at compile-time because important runtime information (e.g. parameter bindings) is missing to estimate costs. The execution of a choose-plan operator yields the comparison of the subplans based on actual costs and the selection of the best one. Choose-plan nodes can be inserted anywhere in a QEP.

Example 8.7. Consider the following query expressed in relational algebra:

$\sigma_{A\leq \$a}(R_1) \bowtie R_2 \bowtie R_3$

Figure 8.10 shows a dynamic execution plan for this query. We assume that each join is performed by nested-loop, with the left operand relation as external and the right operand relation as internal. The bottom choose-plan operator compares the cost of two alternative subplans for joining R_1 and R_2, the left subplan being better than the right one if the selection predicate has high selectivity. As stated above, since there is a runtime binding of the parameter \$a, the accurate selectivity of $\sigma_{A\leq \$a}(R_1)$ cannot be estimated until runtime. The top choose-plan operator compares the cost of two alternative subplans for joining the result of the bottom choose-plan operation with R_3. Depending on the estimated size of the join of R_1 and R_2, which indirectly depends on the selectivity of the selection on R_1 it may be better to use R_3 as external or internal relation. ♦

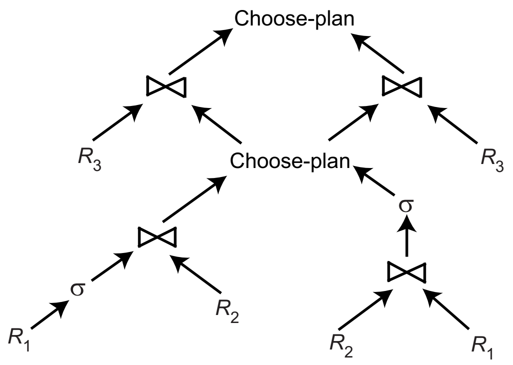


Fig. 8.10 A Dynamic Execution Plan

Dynamic QEPs are produced at compile-time using any static algorithm such as the one presented in Section 8.2.2. However, instead of producing a total order of operations, the optimizer must produce a partial order by introducing choose-node operators anywhere in the QEP. The main modification necessary to a static query optimizer to handle dynamic QEPs is that the cost model supports *incomparable* costs of plans in addition to the standard values “greater than”, “less than” and “equal to”. Costs may be incomparable because the costs of some subplans are unknown at compile-time. Another reason for cost incomparability is when cost is modeled as an interval of possible cost values rather than a single value [Cole and Graefe, 1994]. Therefore, if two plan costs have overlapping intervals, it is not possible to decide which one is better and they should be considered as incomparable.

Given a dynamic QEP, produced by a static query optimizer, the choose-plan decisions must be made at query startup time. The most effective solution is to simply evaluate the costs of the participating subplans and compare them. In Algorithm 8.3, we describe the startup procedure (called Hybrid-QOA) which makes the optimization decisions to produce the final QEP and run it. The algorithm executes the choose-plan operators in bottom-up order and propagates cost information upward in the QEP.

Algorithm 8.3: Hybrid-QOA

```
Input: $QEP$: dynamic QEP; $B$: Query parameter bindinds
Output: $output$: result of execution
begin
    $best\_QEP \leftarrow QEP$ ;
    for each choose-plan operator CP in bottom-up order do
        for each alternative subplan SP do
            compute cost($CP$) using $B$
        $best\_QEP \leftarrow best\_QEP$ without $CP$ and $SP$ of highest cost
    $output \leftarrow$ execute $best\_QEP$
end
```

Experimentation with the Volcano query optimizer [Graefe, 1994] has shown that this hybrid query optimization outperforms both dynamic and static query optimization. In particular, the overhead of dynamic QEP evaluation at startup time is significantly less than that of dynamic optimization, and the reduced execution time of dynamic QEPs relative to static QEPs more than offsets the startup time overhead.

8.3 Join Ordering in Distributed Queries

As we have seen in Section 8.2, ordering joins is an important aspect of centralized query optimization. Join ordering in a distributed context is even more important since joins between fragments may increase the communication time. Two basic approaches exist to order joins in distributed queries. One tries to optimize the ordering of joins directly, whereas the other replaces joins by combinations of semijoins in order to minimize communication costs.

8.3.1 Join Ordering

Some algorithms optimize the ordering of joins directly without using semijoins. The purpose of this section is to stress the difficulty that join ordering presents and to motivate the subsequent section, which deals with the use of semijoins to optimize join queries.

A number of assumptions are necessary to concentrate on the main issues. Since the query is localized and expressed on fragments, we do not need to distinguish between fragments of the same relation and fragments of different relations. To simplify notation, we use the term *relation* to designate a fragment stored at a particular site. Also, to concentrate on join ordering, we ignore local processing time, assuming that reducers (selection, projection) are executed locally either before or during the join (remember that doing selection first is not always efficient). Therefore, we consider only join queries whose operand relations are stored at different sites. We assume that relation transfers are done in a set-at-a-time mode rather than in a tuple-at-a-time mode. Finally, we ignore the transfer time for producing the data at a result site.

Let us first concentrate on the simpler problem of operand transfer in a single join. The query is $R \bowtie S$, where R and S are relations stored at different sites. The obvious choice of the relation to transfer is to send the smaller relation to the site of the larger one, which gives rise to two possibilities, as shown in Figure 8.11. To make this choice we need to evaluate the sizes of R and S. We now consider the case where there are more than two relations to join. As in the case of a single join, the objective of the join-ordering algorithm is to transmit smaller operands. The difficulty stems from the fact that the join operations may reduce or increase the size of the intermediate results. Thus, estimating the size of join results is mandatory, but also difficult. A solution is to estimate the communication costs of all alternative strategies and to choose the best one. However, as discussed earlier, the number of strategies grows rapidly with the number of relations. This approach makes optimization costly, although this overhead is amortized rapidly if the query is executed frequently.

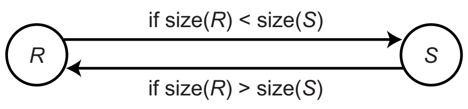


Fig. 8.11 Transfer of Operands in Binary Operation

Example 8.8. Consider the following query expressed in relational algebra:

PROJ $\bowtie_{\text{PNO}}$ ASG $\bowtie_{\text{ENO}}$ EMP

whose join graph is given in Figure 8.12. Note that we have made certain assumptions about the locations of the three relations. This query can be executed in at least five different ways. We describe these strategies by the following programs, where (R $\rightarrow$ site j) stands for "relation R is transferred to site j."

1. EMP $\rightarrow$ site 2; Site 2 computes EMP$' =$ EMP $\bowtie$ ASG; EMP$'$ $\rightarrow$ site 3; Site 3 computes EMP$'$ $\bowtie$ PROJ.
2. ASG $\rightarrow$ site 1; Site 1 computes EMP$' =$ EMP $\bowtie$ ASG; EMP$'$ $\rightarrow$ site 3; Site 3 computes EMP$'$ $\bowtie$ PROJ.

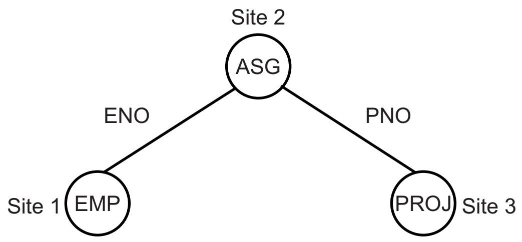


Fig. 8.12 Join Graph of Distributed Query

3. ASG $\rightarrow$ site 3; Site 3 computes ASG$'$ = ASG $\bowtie$ PROJ; ASG$'$ $\rightarrow$ site 1; Site 1 computes ASG$'$ $\bowtie$ EMP.
4. PROJ $\rightarrow$ site 2; Site 2 computes PROJ$'$ = PROJ $\bowtie$ ASG; PROJ$'$ $\rightarrow$ site 1; Site 1 computes PROJ$'$ $\bowtie$ EMP.
5. EMP $\rightarrow$ site 2; PROJ $\rightarrow$ site 2; Site 2 computes EMP $\bowtie$ PROJ $\bowtie$ ASG

To select one of these programs, the following sizes must be known or predicted: *size*(EMP), *size*(ASG), *size*(PROJ), *size*(EMP $\bowtie$ ASG), and *size*(ASG $\bowtie$ PROJ). Furthermore, if it is the response time that is being considered, the optimization must take into account the fact that transfers can be done in parallel with strategy 5. An alternative to enumerating all the solutions is to use heuristics that consider only the sizes of the operand relations by assuming, for example, that the cardinality of the resulting join is the product of operand cardinalities. In this case, relations are ordered by increasing sizes and the order of execution is given by this ordering and the join graph. For instance, the order (EMP, ASG, PROJ) could use strategy 1, while the order (PROJ, ASG, EMP) could use strategy 4. ♦

8.3.2 Semijoin Based Algorithms

In this section we show how the semijoin operation can be used to decrease the total time of join queries. The theory of semijoins was defined by Bernstein and Chiu [1981]. We are making the same assumptions as in Section 8.3.1. The main shortcoming of the join approach described in the preceding section is that entire operand relations must be transferred between sites. The semijoin acts as a size reducer for a relation much as a selection does.

The join of two relations R and S over attribute A, stored at sites 1 and 2, respectively, can be computed by replacing one or both operand relations by a semijoin with the other relation, using the following rules:

$$\begin{aligned} R \bowtie_A S &\Leftrightarrow (R \ltimes_A S) \bowtie_A S \\ &\Leftrightarrow R \bowtie_A (S \ltimes_A R) \end{aligned}$$

$$\Leftrightarrow (R \ltimes_A S) \bowtie_A (S \ltimes_A R)$$

The choice between one of the three semijoin strategies requires estimating their respective costs.

The use of the semijoin is beneficial if the cost to produce and send it to the other site is less than the cost of sending the whole operand relation and of doing the actual join. To illustrate the potential benefit of the semijoin, let us compare the costs of the two alternatives: $R \bowtie_A S$ versus $(R \ltimes_A S) \bowtie_A S$, assuming that $size(R) < size(S)$.

The following program, using the notation of Section 8.3.1, uses the semijoin operation:

1. $\Pi_A(S) \rightarrow$ site 1
2. Site 1 computes $R' = R \ltimes_A S$
3. $R' \rightarrow$ site 2
4. Site 2 computes $R' \bowtie_A S$

For the sake of simplicity, let us ignore the constant T_{MSG} in the communication time assuming that the term $T_{TR} * size(R)$ is much larger. We can then compare the two alternatives in terms of the amount of transmitted data. The cost of the join-based algorithm is that of transferring relation R to site 2. The cost of the semijoin-based algorithm is the cost of steps 1 and 3 above. Therefore, the semijoin approach is better if

$$size(\Pi_A(S)) + size(R \ltimes_A S) < size(R)$$

The semijoin approach is better if the semijoin acts as a sufficient reducer, that is, if a few tuples of R participate in the join. The join approach is better if almost all tuples of R participate in the join, because the semijoin approach requires an additional transfer of a projection on the join attribute. The cost of the projection step can be minimized by encoding the result of the projection in bit arrays [Valduriez, 1982], thereby reducing the cost of transferring the joined attribute values. It is important to note that neither approach is systematically the best; they should be considered as complementary.

More generally, the semijoin can be useful in reducing the size of the operand relations involved in multiple join queries. However, query optimization becomes more complex in these cases. Consider again the join graph of relations EMP, ASG, and PROJ given in Figure 8.12. We can apply the previous join algorithm using semijoins to each individual join. Thus an example of a program to compute EMP $\bowtie$ ASG $\bowtie$ PROJ is EMP$'$ $\bowtie$ ASG$'$ $\bowtie$ PROJ, where EMP$'$ = EMP $\ltimes$ ASG and ASG$'$ = ASG $\ltimes$ PROJ.

However, we may further reduce the size of an operand relation by using more than one semijoin. For example, EMP$'$ can be replaced in the preceding program by EMP$''$ derived as

$$\text{EMP}'' = \text{EMP} \ltimes (\text{ASG} \ltimes \text{PROJ})$$

since if $size(\text{ASG} \ltimes \text{PROJ}) \le size(\text{ASG})$, we have $size(\text{EMP}'') \le size(\text{EMP}')$. In this way, EMP can be reduced by the sequence of semijoins: EMP $\ltimes$ (ASG $\ltimes$ PROJ). Such a sequence of semijoins is called a *semijoin program* for EMP. Similarly, semijoin programs can be found for any relation in a query. For example, PROJ could be reduced by the semijoin program PROJ $\ltimes$ (ASG $\ltimes$ EMP). However, not all of the relations involved in a query need to be reduced; in particular, we can ignore those relations that are not involved in the final joins.

For a given relation, there exist several potential semijoin programs. The number of possibilities is in fact exponential in the number of relations. But there is one optimal semijoin program, called the *full reducer*, which for each relation R reduces R more than the others [Chiu and Ho, 1980]. The problem is to find the full reducer. A simple method is to evaluate the size reduction of all possible semijoin programs and to select the best one. The problems with the enumerative method are twofold:

1. There is a class of queries, called *cyclic queries*, that have cycles in their join graph and for which full reducers cannot be found.
2. For other queries, called *tree queries*, full reducers exist, but the number of candidate semijoin programs is exponential in the number of relations, which makes the enumerative approach NP-hard.

In what follows we discuss solutions to these problems.

Example 8.9. Consider the following relations, where attribute CITY has been added to relations EMP (renamed ET), PROJ (renamed PT) and ASG (renamed AT) of the engineering database. Attribute CITY of AT corresponds to the city where the employee identified by ENO lives.

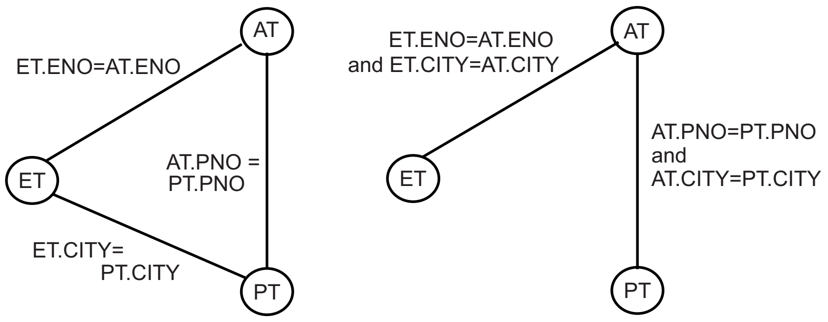
ET(ENO, ENAME, TITLE, CITY)
AT(ENO, PNO, RESP, DUR)
PT(PNO, PNAME, BUDGET, CITY)

The following SQL query retrieves the names of all employees living in the city in which their project is located together with the project name.

```
SELECT ENAME, PNAME
FROM   ET, AT, PT
WHERE  ET.ENO = AT.ENO
AND    AT.ENO = PT.ENO
AND    ET.CITY = PT.CITY
```

As illustrated in Figure 8.13a, this query is cyclic. ♦

No full reducer exists for the query in Example 8.9. In fact, it is possible to derive semijoin programs for reducing it, but the number of operations is multiplied by the number of tuples in each relation, making the approach inefficient. One solution consists of transforming the cyclic graph into a tree by removing one arc of the graph and by adding appropriate predicates to the other arcs such that the removed



(a) Cyclic query (b) Equivalent acyclic query

Fig. 8.13 Transformation of Cyclic Query

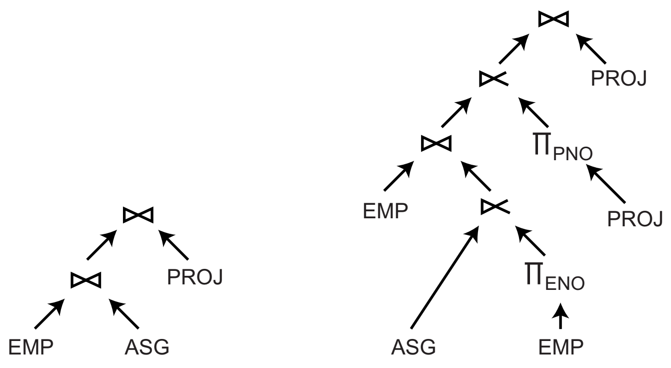
predicate is preserved by transitivity [Kambayashi et al., 1982]. In the example of Figure 8.13b, where the arc (ET, PT) is removed, the additional predicate ET.CITY = AT.CITY and AT.CITY = PT.CITY imply ET.CITY = PT.CITY by transitivity. Thus the acyclic query is equivalent to the cyclic query.

Although full reducers for tree queries exist, the problem of finding them is NP-hard. However, there is an important class of queries, called *chained queries*, for which a polynomial algorithm exists [Chiu and Ho, 1980; Ullman, 1982]). A chained query has a join graph where relations can be ordered, and each relation joins only with the next relation in the order. Furthermore, the result of the query is at the end of the chain. For instance, the query in Figure 8.12 is a chain query. Because of the difficulty of implementing an algorithm with full reducers, most systems use single semijoins to reduce the relation size.

8.3.3 *Join versus Semijoin*

Compared with the join, the semijoin induces more operations but possibly on smaller operands. Figure 8.14 illustrates these differences with an equivalent pair of join and semijoin strategies for the query whose join graph is given in Figure 8.12. The join of two relations, EMP $\bowtie$ ASG in Figure 8.12, is done by sending one relation, ASG, to the site of the other one, EMP, to complete the join locally. When a semijoin is used, however, the transfer of relation ASG is avoided. Instead, it is replaced by the transfer of the join attribute values of relation EMP to the site of relation ASG, followed by the transfer of the matching tuples of relation ASG to the site of relation EMP, where the join is completed. If the join attribute length is smaller than the length of an entire tuple and the semijoin has good selectivity, then the semijoin approach can result in significant savings in communication time. Using semijoins may well increase the local processing time, since one of the two joined relations must be accessed twice. For example, relations EMP and PROJ are accessed twice in Figure

8.14. Furthermore, the join of two intermediate relations produced by semijoins cannot exploit the indices that were available on the base relations. Therefore, using semijoins might not be a good idea if the communication time is not the dominant factor, as is the case with local area networks [Lu and Carey, 1985].



(a) Join approach (b) Semijoin approach

Fig. 8.14 Join versus Semijoin Approaches

Semijoins can still be beneficial with fast networks if they have very good selectivity and are implemented with bit arrays [Valduriez, 1982]. A bit array $BA[1:n]$ is useful in encoding the join attribute values present in one relation. Let us consider the semijoin $R \ltimes S$. Then $BA[i]$ is set to 1 if there exists a join attribute value $A = val$ in relation S such that $h(val) = i$, where h is a hash function. Otherwise, $BA[i]$ is set to 0. Such a bit array is much smaller than a list of join attribute values. Therefore, transferring the bit array instead of the join attribute values to the site of relation R saves communication time. The semijoin can be completed as follows. Each tuple of relation R, whose join attribute value is val, belongs to the semijoin if $BA[h(val)] = 1$.

8.4 Distributed Query Optimization

In this section we illustrate the use of the techniques presented in earlier sections within the context of four basic query optimization algorithms. First, we present the dynamic and static approaches which extend the centralized algorithms presented in Section 8.2. Then, we describe a popular semijoin-based optimization algorithm. Finally, we present a hybrid approach.

8.4.1 Dynamic Approach

We illustrate the dynamic approach with the algorithm of Distributed INGRES [Epstein et al., 1978] that is derived from the algorithm described in Section 8.2.1. The objective function of the algorithm is to minimize a combination of both the communication time and the response time. However, these two objectives may be conflicting. For instance, increasing communication time (by means of parallelism) may well decrease response time. Thus, the function can give a greater weight to one or the other. Note that this query optimization algorithm ignores the cost of transmitting the data to the result site. The algorithm also takes advantage of fragmentation, but only horizontal fragmentation is handled for simplicity.

Since both general and broadcast networks are considered, the optimizer takes into account the network topology. In broadcast networks, the same data unit can be transmitted from one site to all the other sites in a single transfer, and the algorithm explicitly takes advantage of this capability. For example, broadcasting is used to replicate fragments and then to maximize the degree of parallelism.

The input to the algorithm is a query expressed in tuple relational calculus (in conjunctive normal form) and schema information (the network type, as well as the location and size of each fragment). This algorithm is executed by the site, called the *master site*, where the query is initiated. The algorithm, which we call Dynamic*-QOA, is given in Algorithm 8.4.

Algorithm 8.4: Dynamic*-QOA

```
Input: MRQ: multirelation query
Output: result of the last multirelation query
begin
    for each detachable $ORQ_i$ in MRQ do          {ORQ is monorelation query}
        run($ORQ_i$)                                                     (1)
    MRQ'_list ← REDUCE(MRQ)   {MRQ repl. by n irreducible queries}    (2)
    while $n \neq 0$ do              {n is the number of irreducible queries}  (3)
        {choose next irreducible query involving the smallest fragments}
        MRQ' ← SELECT_QUERY(MRQ'_list);                                (3.1)
        {determine fragments to transfer and processing site for MRQ'}
        Fragment-site-list ← SELECT_STRATEGY(MRQ');                    (3.2)
        {move the selected fragments to the selected sites}
        for each pair $(F,S)$ in Fragment-site-list do
            move fragment F to site S                                  (3.3)
        execute MRQ';                                                  (3.4)
        $n \leftarrow n-1$
    {output is the result of the last MRQ'}
end
```

All monorelation queries (e.g., selection and projection) that can be detached are first processed locally [Step (1)]. Then the reduction algorithm [Wong and Youssefi, 1976] is applied to the original query [Step (2)]. Reduction is a technique that isolates all irreducible subqueries and monorelation subqueries by detachment (see Section 8.2.1). Monorelation subqueries are ignored because they have already been processed in step (1). Thus the REDUCE procedure produces a sequence of irreducible subqueries $q_1 \rightarrow q_2 \rightarrow \cdots \rightarrow q_n$, with at most one relation in common between two consecutive subqueries. Wong and Youssefi [1976] have shown that such a sequence is unique. Example 8.4 (in Section 8.2.1), which illustrated the detachment technique, also illustrates what the REDUCE procedure would produce.

Based on the list of irreducible queries isolated in step (2) and the size of each fragment, the next subquery, MRQ', which has at least two variables, is chosen at step (3.1) and steps (3.2), (3.3), and (3.4) are applied to it. Steps (3.1) and (3.2) are discussed below. Step (3.2) selects the best strategy to process the query MRQ'. This strategy is described by a list of pairs (F,S), in which F is a fragment to transfer to the processing site S. Step (3.3) transfers all the fragments to their processing sites. Finally, step (3.4) executes the query MRQ'. If there are remaining subqueries, the algorithm goes back to step (3) and performs the next iteration. Otherwise, it terminates.

Optimization occurs in steps (3.1) and (3.2). The algorithm has produced subqueries with several components and their dependency order (similar to the one given by a relational algebra tree). At step (3.1) a simple choice for the next subquery is to take the next one having no predecessor and involving the smaller fragments. This minimizes the size of the intermediate results. For example, if a query q has the subqueries q_1, q_2, and q_3, with dependencies $q_1 \rightarrow q_3, q_2 \rightarrow q_3$, and if the fragments referred to by q_1 are smaller than those referred to by q_2, then q_1 is selected. Depending on the network, this choice can also be affected by the number of sites having relevant fragments.

The subquery selected must then be executed. Since the relation involved in a subquery may be stored at different sites and even fragmented, the subquery may nevertheless be further subdivided.

Example 8.10. Assume that relations EMP, ASG, and PROJ of the query of Example 8.4 are stored as follows, where relation EMP is fragmented.

Site 1	Site 2
EMP_1	EMP_2
ASG	PROJ

There are several possible strategies, including the following:

1. Execute the entire query (EMP ⋈ ASG ⋈ PROJ) by moving EMP_1 and ASG to site 2.
2. Execute (EMP ⋈ ASG) ⋈ PROJ by moving (EMP_1 ⋈ ASG) and ASG to site 2, and so on.

The choice between the possible strategies requires an estimate of the size of the intermediate results. For example, if *size*($\text{EMP}_1 \bowtie \text{ASG}$) $>$ *size* (EMP_1), strategy 1 is preferred to strategy 2. Therefore, an estimate of the size of joins is required. ♦

At step (3.2), the next optimization problem is to determine how to execute the subquery by selecting the fragments that will be moved and the sites where the processing will take place. For an n-relation subquery, fragments from $n-1$ relations must be moved to the site(s) of fragments of the remaining relation, say R_p, and then replicated there. Also, the remaining relation may be further partitioned into k "equalized" fragments in order to increase parallelism. This method is called *fragment-and-replicate* and performs a substitution of fragments rather than of tuples. The selection of the remaining relation and of the number of processing sites k on which it should be partitioned is based on the objective function and the topology of the network. Remember that replication is cheaper in broadcast networks than in point-to-point networks. Furthermore, the choice of the number of processing sites involves a trade-off between response time and total time. A larger number of sites decreases response time (by parallel processing) but increases total time, in particular increasing communication costs.

Epstein et al. [1978] give formulas to minimize either communication time or processing time. These formulas use as input the location of fragments, their size, and the network type. They can minimize both costs but with a priority to one. To illustrate these formulas, we give the rules for minimizing communication time. The rule for minimizing response time is even more complex. We use the following assumptions. There are n relations $R_1, R_2, \ldots, R_n$ involved in the query. R_i^j denotes the fragment of R_i stored at site j. There are m sites in the network. Finally, $CT_k(\#bytes)$ denotes the communication time of transferring $\#bytes$ to k sites, with $1 \leq k \leq m$.

The rule for minimizing communication time considers the types of networks separately. Let us first concentrate on a broadcast network. In this case we have

$$CT_k(\#bytes) = CT_1(\#bytes)$$

The rule can be stated as

```
if $\max_{j=1,m}(\sum_{i=1}^{n} size(R_i^j)) > \max_{i=1,n}(size(R_i))$
then
    the processing site is the $j$ that has the largest amount of data
else
    $R_p$ is the largest relation and site of $R_p$ is the processing site
```

If the inequality predicate is satisfied, one site contains an amount of data useful to the query larger than the size of the largest relation. Therefore, this site should be the processing site. If the predicate is not satisfied, one relation is larger than the maximum useful amount of data at one site. Therefore, this relation should be the R_p, and the processing sites are those which have its fragments.

Let us now consider the case of the point-to-point networks. In this case we have

$$CT_k(\#bytes) = k * CT_1(\#bytes)$$

The choice of R_p that minimizes communication is obviously the largest relation. Assuming that the sites are arranged by decreasing order of amounts of useful data for the query, that is,

$$\sum_{i=1}^{n} size(R_i^j) > \sum_{i=1}^{n} size(R_i^{j+1})$$

the choice of k, the number of sites at which processing needs to be done, is given as

```
**if** $\sum_{i \neq p}(size(R_i) - size(R_i^1)) > size(R_p^1)$
**then**
    $k = 1$
**else**
    $k$ is the largest $j$ such that $\sum_{i \neq p}(size(R_i) - size(R_i^j)) \leq size(R_p^j)$
```

This rule chooses a site as the processing site only if the amount of data it must receive is smaller than the additional amount of data it would have to send if it were not a processing site. Obviously, the then-part of the rule assumes that site 1 stores a fragment of R_p.

Example 8.11. Let us consider the query PROJ $\bowtie$ ASG, where PROJ and ASG are fragmented. Assume that the allocation of fragments and their sizes are as follows (in kilobytes):

	Site 1	Site 2	Site 3	Site 4
PROJ	1000	1000	1000	1000
ASG			2000	

With a point–to–point network, the best strategy is to send each PROJ$_i$ to site 3, which requires a transfer of 3000 kbytes, versus 6000 kbytes if ASG is sent to sites 1, 2, and 4. However, with a broadcast network, the best strategy is to send ASG (in a single transfer) to sites 1, 2, and 4, which incurs a transfer of 2000 kbytes. The latter strategy is faster and maximizes response time because the joins can be done in parallel. ♦

This dynamic query optimization algorithm is characterized by a limited search of the solution space, where an optimization decision is taken for each step without concerning itself with the consequences of that decision on global optimization. However, the algorithm is able to correct a local decision that proves to be incorrect.

8.4.2 Static Approach

We illustrate the static approach with the algorithm of R* [Selinger and Adiba, 1980; Lohman et al., 1985] which is a substantial extension of the techniques we described in Section 8.2.2). This algorithm performs an exhaustive search of all alternative strategies in order to choose the one with the least cost. Although predicting and enumerating these strategies may be costly, the overhead of exhaustive search is rapidly amortized if the query is executed frequently. Query compilation is a distributed task, coordinated by a *master site*, where the query is initiated. The optimizer of the master site makes all intersite decisions, such as the selection of the execution sites and the fragments as well as the method for transferring data. The *apprentice sites*, which are the other sites that have relations involved in the query, make the remaining local decisions (such as the ordering of joins at a site) and generate local access plans for the query. The objective function of the optimizer is the general total time function, including local processing and communications costs (see Section 8.1.1).

We now summarize this query optimization algorithm. The input to the algorithm is a localized query expressed as a relational algebra tree (the query tree), the location of relations, and their statistics. The algorithm is described by the procedure Static*-QOA in Algorithm 8.5.

Algorithm 8.5: Static*-QOA

```
Input: $QT$: query tree
Output: $strat$: minimum cost strategy
begin
    for each relation $R_i \in QT$ do
        for each access path $AP_{ij}$ to $R_i$ do
            compute $cost(AP_{ij})$
        $best\_AP_i \leftarrow AP_{ij}$ with minimum cost
    for each order $(R_{i1}, R_{i2}, \cdots, R_{in})$ with $i = 1, \cdots, n!$ do
        build strategy $(\ldots((\text{best } AP_{i1} \bowtie R_{i2}) \bowtie R_{i3}) \bowtie \ldots \bowtie R_{in})$ ;
        compute the cost of strategy
    $strat \leftarrow$ strategy with minimum cost ;
    for each site $k$ storing a relation involved in $QT$ do
        $LS_k \leftarrow$ local strategy (strategy, $k$) ;
        send ($LS_k$, site $k$)          {each local strategy is optimized at site $k$}
end
```

As in the centralized case, the optimizer must select the join ordering, the join algorithm (nested-loop or merge-join), and the access path for each fragment (e.g., clustered index, sequential scan, etc.). These decisions are based on statistics and formulas used to estimate the size of intermediate results and access path information. In addition, the optimizer must select the sites of join results and the method of transferring data between sites. To join two relations, there are three candidate sites: the site of the first relation, the site of the second relation, or a third site (e.g., the site of a third relation to be joined with). Two methods are supported for intersite data transfers.

1. *Ship-whole*. The entire relation is shipped to the join site and stored in a temporary relation before being joined. If the join algorithm is merge join, the relation does not need to be stored, and the join site can process incoming tuples in a pipeline mode, as they arrive.
2. *Fetch-as-needed*. The external relation is sequentially scanned, and for each tuple the join value is sent to the site of the internal relation, which selects the internal tuples matching the value and sends the selected tuples to the site of the external relation. This method is equivalent to the semijoin of the internal relation with each external tuple.

The trade-off between these two methods is obvious. Ship-whole generates a larger data transfer but fewer messages than fetch-as-needed. It is intuitively better to ship whole relations when they are small. On the contrary, if the relation is large and the join has good selectivity (only a few matching tuples), the relevant tuples should be fetched as needed. The optimizer does not consider all possible combinations of join methods with transfer methods since some of them are not worthwhile. For example, it would be useless to transfer the external relation using fetch-as-needed in the nested-loop join algorithm, because all the outer tuples must be processed anyway and therefore should be transferred as a whole.

Given the join of an external relation R with an internal relation S on attribute A, there are four join strategies. In what follows we describe each strategy in detail and provide a simplified cost formula for each, where LT denotes local processing time (I/O + CPU time) and CT denotes communication time. For simplicity, we ignore the cost of producing the result. For convenience, we denote by s the average number of tuples of S that match one tuple of R:

$$s = \frac{card(S \ltimes_A R)}{card(R)}$$

Strategy 1.

Ship the entire external relation to the site of the internal relation. In this case the external tuples can be joined with S as they arrive. Thus we have

$$\begin{aligned} Total\_cost = {} & LT(\text{retrieve } card(R) \text{ tuples from } R) \\ & + CT(size(R)) \\ & + LT(\text{retrieve } s \text{ tuples from } S) * card(R) \end{aligned}$$

Strategy 2.

Ship the entire internal relation to the site of the external relation. In this case, the internal tuples cannot be joined as they arrive, and they need to be stored in a temporary relation T. Thus we have

$$\begin{aligned} Total\_cost = {} & LT(\text{retrieve } card(S) \text{ tuples from } S) \\ & + CT(size(S)) \\ & + LT(\text{store } card(S) \text{ tuples in } T) \\ & + LT(\text{retrieve } card(R) \text{ tuples from } R) \\ & + LT(\text{retrieve } s \text{ tuples from } T) * card(R) \end{aligned}$$

Strategy 3.

Fetch tuples of the internal relation as needed for each tuple of the external relation. In this case, for each tuple in R, the join attribute value is sent to the site of S. Then the s tuples of S which match that value are retrieved and sent to the site of R to be joined as they arrive. Thus we have

$$\begin{aligned} Total\_cost = {} & LT(\text{retrieve } card(R) \text{ tuples from } R) \\ & + CT(length(A)) * card(R) \\ & + LT(\text{retrieve } s \text{ tuples from } S) * card(R) \\ & + CT(s * length(S)) * card(R) \end{aligned}$$

Strategy 4.

Move both relations to a third site and compute the join there. In this case the internal relation is first moved to a third site and stored in a temporary relation T. Then the external relation is moved to the third site and its tuples are joined with T as they arrive. Thus we have

$$\begin{aligned} Total\_cost = {} & LT(\text{retrieve } card(S) \text{ tuples from } S) \\ & + CT(size(S)) \\ & + LT(\text{store } card(S) \text{ tuples in } T) \\ & + LT(\text{retrieve } card(R) \text{ tuples from } R) \\ & + CT(size(R)) \\ & + LT(\text{retrieve } s \text{ tuples from } T) * card(R) \end{aligned}$$

Example 8.12. Let us consider a query that consists of the join of relations PROJ, the external relation, and ASG, the internal relation, on attribute PNO. We assume that PROJ and ASG are stored at two different sites and that there is an index on attribute PNO for relation ASG. The possible execution strategies for the query are as follows:

1. Ship whole PROJ to site of ASG.
2. Ship whole ASG to site of PROJ.
3. Fetch ASG tuples as needed for each tuple of PROJ.

4. Move ASG and PROJ to a third site.

The optimization algorithm predicts the total time of each strategy and selects the cheapest. Given that there is no operation following the join PROJ ⋈ ASG, strategy 4 obviously incurs the highest cost since both relations must be transferred. If *size*(PROJ) is much larger than *size*(ASG), strategy 2 minimizes the communication time and is likely to be the best if local processing time is not too high compared to strategies 1 and 3. Note that the local processing time of strategies 1 and 3 is probably much better than that of strategy 2 since they exploit the index on the join attribute.

If strategy 2 is not the best, the choice is between strategies 1 and 3. Local processing costs in both of these alternatives are identical. If PROJ is large and only a few tuples of ASG match, strategy 3 probably incurs the least communication time and is the best. Otherwise, that is, if PROJ is small or many tuples of ASG match, strategy 1 should be the best. ♦

Conceptually, the algorithm can be viewed as an exhaustive search among all alternatives that are defined by the permutation of the relation join order, join methods (including the selection of the join algorithm), result site, access path to the internal relation, and intersite transfer mode. Such an algorithm has a combinatorial complexity in the number of relations involved. Actually, the algorithm significantly reduces the number of alternatives by using dynamic programming and the heuristics, as does the System R's optimizer (see Section 8.2.2). With dynamic programming, the tree of alternatives is dynamically constructed and pruned by eliminating the inefficient choices.

Performance evaluation of the algorithm in the context of both high-speed networks (similar to local networks) and medium-speed wide area networks confirm the significant contribution of local processing costs, even for wide area networks[Lohman and Mackert, 1986; Mackert and Lohman, 1986]. It is shown in particular that for the distributed join, transferring the entire internal relation outperforms the fetch-as-needed method.

8.4.3 Semijoin-based Approach

We illustrate the semijoin-based approach with the algorithm of SDD-1 [Bernstein et al., 1981] which takes full advantage of the semijoin to minimize communication cost. The query optimization algorithm is derived from an earlier method called the "hill-climbing" algorithm [Wong, 1977], which has the distinction of being the first distributed query processing algorithm. In the hill-climbing algorithm, refinements of an initial feasible solution are recursively computed until no more cost improvements can be made. The algorithm does not use semijoins, nor does it assume data replication and fragmentation. It is devised for wide area point-to-point networks. The cost of transferring the result to the final site is ignored. This algorithm is quite general in that it can minimize an arbitrary objective function, including the total time and response time.

The hill-climbing algorithm proceeds as follows. The input to the algorithm includes the query graph, location of relations, and relation statistics. Following the completion of initial local processing, an initial feasible solution is selected which is a global execution schedule that includes all intersite communication. It is obtained by computing the cost of all the execution strategies that transfer all the required relations to a single candidate result site, and then choosing the least costly strategy. Let us denote this initial strategy as ES_0. Then the optimizer splits ES_0 into two strategies, ES_1 followed by ES_2, where ES_1 consists of sending one of the relations involved in the join to the site of the other relation. The two relations are joined locally and the resulting relation is transmitted to the chosen result site (specified as schedule ES_2). If the cost of executing strategies ES_1 and ES_2, plus the cost of local join processing, is less than that of ES_0, then ES_0 is replaced in the schedule by ES_1 and ES_2. The process is then applied recursively to ES_1 and ES_2 until no more benefit can be gained. Notice that if n-way joins are involved, ES_0 will be divided into n subschedules instead of just two.

The hill-climbing algorithm is in the class of greedy algorithms, which start with an initial feasible solution and iteratively improve it. The main problem is that strategies with higher initial cost, which could nevertheless produce better overall benefits, are ignored. Furthermore, the algorithm may get stuck at a local minimum cost solution and fail to reach the global minimum.

Example 8.13. Let us illustrate the hill-climbing algorithm using the following query involving relations EMP, PAY, PROJ, and ASG of the engineering database:

"Find the salaries of engineers who work on the CAD/CAM project"

The query in relational algebra is

$$\Pi_{\text{SAL}}\ (\text{PAY} \bowtie_{\text{TITLE}} (\text{EMP} \bowtie_{\text{ENO}} (\text{ASG} \bowtie_{\text{PNO}} (\ \sigma_{\text{PNAME} = \text{“CAD/CAM”}}(\text{PROJ})))))$$

We assume that $T_{MSG} = 0$ and $T_{TR} = 1$. Furthermore, we ignore the local processing, following which the database is

Relation	Size	Site
EMP	8	1
PAY	4	2
PROJ	1	3
ASG	10	4

To simplify this example, we assume that the length of a tuple (of every relation) is 1, which means that the size of a relation is equal to its cardinality. Furthermore, the placement of the relation is arbitrary. Based on join selectivities, we know that $size(\text{EMP} \bowtie \text{PAY}) = size(\text{EMP})$, $size(\text{PROJ} \bowtie \text{ASG}) = 2 * size(\text{PROJ})$, and $size(\text{ASG} \bowtie \text{EMP}) = size(\text{ASG})$.

Considering only data transfers, the initial feasible solution is to choose site 4 as the result site, producing the schedule

ES_0 : EMP $\rightarrow$ site 4
PAY $\rightarrow$ site 4
PROJ $\rightarrow$ site 4
$Total\_cost(ES_0) = 4+8+1 = 13$

This is true because the cost of any other solution is greater than the foregoing alternative. For example, if one chooses site 2 as the result site and transmits all the relations to that site, the total cost will be

$$\begin{aligned} Total\_cost &= cost(\text{EMP} \rightarrow \text{site 2}) + cost(\text{ASG} \rightarrow \text{site 2}) \\ &\quad + cost(\text{PROJ} \rightarrow \text{site 2}) \\ &= 19 \end{aligned}$$

Similarly, the total cost of choosing either site 1 or site 3 as the result site is 15 and 22, respectively.

One way of splitting this schedule (call it ES') is the following:

ES_1 : EMP $\rightarrow$ site 2
ES_2 : (EMP $\bowtie$ PAY) $\rightarrow$ site 4
ES_3 : PROJ $\rightarrow$ site 4
$Total\_cost(ES') = 8+8+1 = 17$

A second splitting alternative (ES'') is as follows:

ES_1 : PAY $\rightarrow$ site 1
ES_2 : (PAY $\bowtie$ EMP) $\rightarrow$ site 4
ES_3 : PROJ $\rightarrow$ site 4
$Total\_cost(ES'') = 4+8+1 = 13$

Since the cost of either of the alternatives is greater than or equal to the cost of ES_0, ES_0 is kept as the final solution. A better solution (ignored by the algorithm) is

B : PROJ $\rightarrow$ site 4
ASG$'$ = (PROJ $\bowtie$ ASG) $\rightarrow$ site 1
(ASG$'$ $\bowtie$ EMP) $\rightarrow$ site 2
$Total\_cost(B) = 1+2+2 = 5$

♦

The semijoin-based algorithm extends the hill-climbing algorithm in a number of ways [Bernstein et al., 1981]. In addition to the extensive use of semijoins, the objective function is expressed in terms of total communication time (local time and response time are not considered). Furthermore, the algorithm uses statistics on the database, called *database profiles*, where a profile is associated with a relation. The algorithm also selects an initial feasible solution that is iteratively refined. Finally, a postoptimization step is added to improve the total time of the solution selected. The main step of the algorithm consists of determining and ordering beneficial semijoins, that is semijoins whose cost is less than their benefit.

The cost of a semijoin is that of transferring the semijoin attributes A,

$$Cost(R \ltimes_A S) = T_{MSG} + T_{TR} * size(\Pi_A(S))$$

while its benefit is the cost of transferring irrelevant tuples of R (which is avoided by the semijoin):

$$Benefit(R \ltimes_A S) = (1 - SF_{SJ}(S.A)) * size(R) * T_{TR}$$

The semijoin-based algorithm proceeds in four phases: initialization, selection of beneficial semijoins, assembly site selection, and postoptimization. The output of the algorithm is a global strategy for executing the query (Algorithm 8.6).

Algorithm 8.6: Semijoin-based-QOA

```
Input: $QG$: query graph with $n$ relations; statistics for each relation
Output: $ES$: execution strategy
begin
    ES $\leftarrow$ local-operations ($QG$) ;
    modify statistics to reflect the effect of local processing ;
    BS $\leftarrow \phi$;                                  {set of beneficial semijoins}
    for each semijoin $SJ$ in $QG$ do
        if $cost(SJ) < benefit(SJ)$ then
            $BS \leftarrow BS \cup SJ$
    while $BS \neq \phi$ do
                                                    {selection of beneficial semijoins}
        $SJ \leftarrow most\_beneficial(BS)$;   {$SJ$: semijoin with $max(benefit - cost)$}
        $BS \leftarrow BS - SJ$;                              {remove $SJ$ from $BS$}
        $ES \leftarrow ES + SJ$;                     {append $SJ$ to execution strategy}
        modify statistics to reflect the effect of incorporating $SJ$ ;
        $BS \leftarrow BS -$ non-beneficial semijoins ;
        $BS \leftarrow BS \cup$ new beneficial semijoins ;
    {assembly site selection}
    $AS(ES) \leftarrow$ select site $i$ such that $i$ stores the largest amount of data after all
    local operations ;
    $ES \leftarrow ES \cup$ transfers of intermediate relations to $AS(ES)$ ;
    {postoptimization}
    for each relation $R_i$ at $AS(ES)$ do
        for each semijoin $SJ$ of $R_i$ by $R_j$ do
            if $cost(ES) > cost(ES - SJ)$ then
                $ES \leftarrow ES - SJ$
end
```

The initialization phase generates a set of beneficial semijoins, $BS = \{SJ_1, SJ_2, \ldots, SJ_k\}$, and an execution strategy ES that includes only local processing. The next phase selects the beneficial semijoins from BS by iteratively choosing the most beneficial semijoin, SJ_i, and modifying the database statistics and BS accordingly. The modification affects the statistics of relation R involved in SJ_i and the remaining semijoins in BS that use relation R. The iterative phase terminates when all semijoins in BS have been appended to the execution strategy. The order in which semijoins are appended to ES will be the execution order of the semijoins.

The next phase selects the assembly site by evaluating, for each candidate site, the cost of transferring to it all the required data and taking the one with the least cost. Finally, a postoptimization phase permits the removal from the execution strategy of those semijoins that affect only relations stored at the assembly site. This phase is necessary because the assembly site is chosen after all the semijoins have been ordered. The SDD-1 optimizer is based on the assumption that relations can be transmitted to another site. This is true for all relations except those stored at the assembly site, which is selected after beneficial semijoins are considered. Therefore, some semijoins may incorrectly be considered beneficial. It is the role of postoptimization to remove them from the execution strategy.

Example 8.14. Let us consider the following query:

```
SELECT R3.C
FROM   R1,R2,R3
WHERE  R1.A = R2.A
AND    R2.B = R3.B
```

Figure 8.15 gives the join graph of the query and of relation statistics. We assume that $T_{MSG} = 0$ and $T_{TR} = 1$. The initial set of beneficial semijoins will contain the following two:

SJ_1: $R_2 \ltimes R_1$, whose benefit is $2100 = (1-0.3)*3000$ and cost is 36
SJ_2: $R_2 \ltimes R_3$, whose benefit is $1800 = (1-0.4)*3000$ and cost is 80

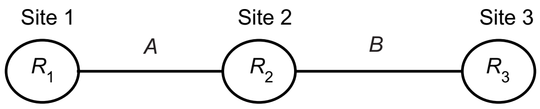
Furthermore there are two non-beneficial semijoins:

SJ_3: $R_1 \ltimes R_2$, whose benefit is $300 = (1-0.8)*1500$ and cost is 320
SJ_4: $R_3 \ltimes R_2$, whose benefit is 0 and cost is 400.

At the first iteration of the selection of beneficial semijoins, SJ_1 is appended to the execution strategy ES. One effect on the statistics is to change the size of R_2 to $900 = 3000 * 0.3$. Furthermore, the semijoin selectivity factor of attribute $R_2.A$ is reduced because $card(\Pi_A(R_2))$ is reduced. We approximate $SF_{SJ}(R_2.A)$ by $0.8*0.3 = 0.24$. Finally, size of $\Pi_{R_2.A}$ is also reduced to $96 = 320*0.3$. Similarly, the semijoin selectivity factor of attribute $R_2.B$ and $\Pi_{R_2.B}$ should also be reduced (but they not needed in the rest of the example).

At the second iteration, there are two beneficial semijoins:

SJ_2 : $R_2' \ltimes R_3$, whose benefit is $540 = 900*(1-0.4)$ and cost is 80
(here $R_2' = R_2 \ltimes R_1$, which is obtained by SJ_1
SJ_3: $R_1 \ltimes R_2'$, whose benefit is $1140 = (1-0.24)*1500$ and cost is 96



relation	card	tuple size	relation size
R_1	30	50	1500
R_2	100	30	3000
R_3	50	40	2000

attribute	SF_{SJ}	$size(\Pi_{attribute})$
$R_1.A$	0.3	36
$R_2.A$	0.8	320
$R_2.B$	1.0	400
$R_3.B$	0.4	80

Fig. 8.15 Example Query and Statistics

The most beneficial semijoin is SJ_3 and is appended to ES. One effect on the statistics of relation R_1 is to change the size of R_1 to $360 (= 1500 * 0.24)$. Another effect is to change the selectivity of R_1 and size of $\Pi_{R_1.A}$.

At the third iteration, the only remaining beneficial semijoin, SJ_2, is appended to ES. Its effect is to reduce the size of relation R_2 to $360 (= 900 * 0.4)$. Again, the statistics of relation R_2 may also change.

After reduction, the amount of data stored is 360 at site 1, 360 at site 2, and 2000 at site 3. Site 3 is therefore chosen as the assembly site. The postoptimization does not remove any semijoin since they all remain beneficial. The strategy selected is to send $(R_2 \ltimes R_1) \ltimes R_3$ and $R_1 \ltimes R_2$ to site 3, where the final result is computed. ♦

Like its predecessor hill-climbing algorithm, the semijoin-based algorithm selects locally optimal strategies. Therefore, it ignores the higher-cost semijoins which would result in increasing the benefits and decreasing the costs of other semijoins. Thus this algorithm may not be able to select the global minimum cost solution.

8.4.4 Hybrid Approach

The static and dynamic distributed optimization approaches have the same advantages and disadvantages as in centralized systems (see Section 8.2.3). However, the problems of accurate cost estimation and comparison of QEPs at compile-time are much more severe in distributed systems. In addition to unknown bindings of parameter values in embedded queries, sites may become unvailable or overloaded at runtime. In addition, relations (or relation fragments) may be replicated at several sites. Thus, site and copy selection should be done at runtime to increase availability and load balancing of the system.

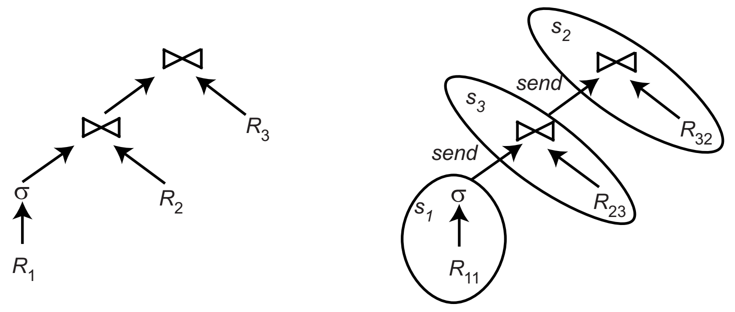
The hybrid query optimization technique using dynamic QEPs (see Section 8.2.3) is general enough to incorporate site and copy selection decisions. However, the search space of alternative subplans linked by choose-plan operators becomes much larger and may result in heavy static plans and much higher startup time. Therefore, several hybrid techniques have been proposed to optimize queries in distributed systems [Carey and Lu, 1986; Du et al., 1995; Evrendilek et al., 1997]. They essentially rely on the following two-step approach:

1. At compile time, generate a static plan that specifies the ordering of operations and the access methods, without considering where relations are stored.
2. At startup time, generate an execution plan by carrying out site and copy selection and allocating the operations to the sites.

Example 8.15. Consider the following query expressed in relational algebra:

$$\sigma(R_1) \bowtie R_2 \bowtie R_3$$

Figure 8.16 shows a 2-step plan for this query. The static plan shows the relational operation ordering as produced by a centralized query optimizer. The run-time plan extends the static plan with site and copy selection and communication between sites. For instance, the first selection is allocated at site s_1 on copy R_{11} of relation R_1 and sends its result to site s_3 to be joined with R_{23} and so on. ♦



(a) Static plan (b) Run-time plan

Fig. 8.16 A 2-Step Plan

The first step can be done by a centralized query optimizer. It may also include choose-plan operators so that runtime bindings can be used at startup time to make accurate cost estimations. The second step carries out site and copy selection, possibly in addition to choose-plan operator execution. Furthermore, it can optimize the load balancing of the system. In the rest of this section, we illustrate this second step based on the seminal paper by Carey and Lu [1986] on two-step query optimization.

We consider a distributed database system with a set of sites $S = \{s_1,..,s_n\}$. A query Q is represented as an ordered sequence of subqueries $Q = \{q_1,..,q_m\}$. Each subquery q_i is the maximum processing unit that accesses a single base relation and communicates with its neighboring subqueries. For instance, in Figure 8.16, there are three subqueries, one for R_1, one for R_2, and one for R_3. Each site s_i has a load, denoted by $load(s_i)$, which reflects the number of queries currently submitted. The load can be expressed in different ways, e.g. as the number of I/O bound and CPU bound queries at the site [Carey and Lu, 1986]. The average load of the system is defined as:

$$Avg\_load(S) = \frac{\sum_{i=1}^{n} load(s_i)}{n}$$

The balance of the system for a given allocation of subqueries to sites can be measured as the variance of the site loads using the following *unbalance factor* [Carey and Lu, 1986]:

$$UF(S) = \frac{1}{n}\sum_{i=1}^{n}(load(s_i) - Avg\_load(S))^2$$

As the system gets balanced, its unbalance factor approaches 0 (perfect balance). For example, with $load(s_1)$=10 and $load(s_1)$=30, the unbalance factor of s_1, s_2 is 100 while with $load(s_1)$=20 and $load(s_1)$=20, it is 0.

The problem addressed by the second step of two-step query optimization can be formalized as the following subquery allocation problem. Given

1. a set of sites $S = \{s_1,..,s_n\}$ with the load of each site;
2. a query $Q = \{q_1,..,q_m\}$; and
3. for each subquery q_i in Q, a feasible allocation set of sites $S_q = \{s_1,...,s_k\}$ where each site stores a copy of the relation involved in q_i;

the objective is to find an optimal allocation on Q to S such that

1. $UF(S)$ is minimized, and
2. the total communication cost is minimized.

Carey and Lu [1986] propose an algorithm that finds near-optimal solutions in a reasonable amount of time. The algorithm, which we describe in Algorithm 8.7 for linear join trees, uses several heuristics. The first heuristic (step 1) is to start by allocating subqueries with least allocation flexibility, i.e. with the smaller feasible allocation sets of sites. Thus, subqueries with a few candidate sites are allocated earlier. Another heuristic (step 2) is to consider the sites with least load and best benefit. The benefit of a site is defined as the number of subqueries already allocated to the site and measures the communication cost savings from allocating the subquery to the site. Finally, in step 3 of the algorithm, the load information of any unallocated subquery that has a selected site in its feasible allocation set is recomputed.

Algorithm 8.7: SQAllocation

```
Input: $Q$: $q_1, \ldots, q_m$ ;
  Feasible allocation sets: $S_{q_1}, \ldots, S_{q_m}$ ;
  Loads: $load(S_1), \ldots, load(S_m)$;
Output: an allocation of $Q$ to $S$
begin
    for each $q$ in $Q$ do
        compute($load(S_q)$)
    while $Q$ not empty do
        $a \leftarrow q \in Q$ with least allocation flexibility; {select subquery $a$ for
        allocation}                                                          (1)
        $b \leftarrow s \in S_a$ with least load and best benefit; {select best site $b$ for $a$} (2)
        $Q \leftarrow Q - a$ ;
        {recompute loads of remaining feasible allocation sets if necessary} (3)
        for each $q \in Q$ where $b \in S_q$ do
            compute($load(S_q)$)
end
```

Example 8.16. Consider the following query Q expressed in relational algebra:

$\sigma(R_1) \bowtie R_2 \bowtie R_3 \bowtie R_4$

Figure 8.17 shows the placement of the copies of the 4 relations at the 4 sites, and the site loads. We assume that Q is decomposed as $Q = \{q_1, q_2, q_3, q_4\}$ where q_1 is associated with R_1, q_2 with R_2 joined with the result of q_1, q_3 with R_3 joined with the result of q_2, and q_4 with R_4 joined with the result of q_3. The SQAllocation algorithm performs 4 iterations. At the first one, it selects q_4 which has the least allocation flexibility, allocates it to s_1 and updates the load of s_1 to 2. At the second iteration, the next set of subqueries to be selected are either q_2 or q_3 since they have the same allocation flexibility. Let us choose q_2 and assume it gets allocated to s_2 (it could be allocated to s_4 which has the same load as s_2). The load of s_2 is increased to 3. At the third iteration, the next subquery selected is q_3 and it is allocated to s_1 which has the same load as s_3 but a benefit of 1 (versus 0 for s_3) as a result of the allocation of q_4. The load of s_1 is increased to 3. Finally, at the last iteration, q_1 gets allocated to either s_3 or s_4 which have the least loads. If in the second iteration q_2 were allocated to s_4 instead of to s_2, then the fourth iteration would have allocated q_1 to s_4 because of a benefit of 1. This would have produced a better execution plan with less communication. This illustrates that two-step optimization can still miss optimal plans. ♦

sites	load	R_1	R_2	R_3	R_4
s_1	1	R_{11}		R_{31}	R_{41}
s_2	2		R_{22}		
s_3	2	R_{13}		R_{33}	
s_4	2	R_{14}	R_{24}		

Fig. 8.17 Example Data Placement and Load

This algorithm has reasonable complexity. It considers each subquery in turn, considering each potential site, selects a current one for allocation, and sorts the list of remaining subqueries. Thus, its complexity can be expressed as $O(max(m*n, m^2 * log_2 m))$.

Finally, the algorithm includes a refining phase to further optimize join processing and decide whether or not to use semijoins. Although it minimizes communication given a static plan, two-step query optimization may generate runtime plans that have higher communication cost than the optimal plan. This is because the first step is carried out ignoring data location and its impact on communication cost. For instance, consider the runtime plan in 8.16 and assume that the third subquery on R_3 is allocated to site s_1 (instead of site s_2). In this case, the plan that does the join (or Cartesian product) of the result of the selection of R_1 with R_3 first at site s_1 may be better since it minimizes communication. A solution to this problem is to perform plan reorganization using operation tree transformations at startup time [Du et al., 1995].

8.5 Conclusion

In this chapter we have presented the basic concepts and techniques for distributed query optimization. We first introduced the main components of query optimization, including the search space, the cost model and the search strategy. The details of the environment (centralized versus distributed) are captured by the search space and the cost model. The search space describes the equivalent execution plans for the input query. These plans differ on the execution order of operations and their implementation, and therefore on performance. The search space is obtained by applying transformation rules, such as those described in Section 7.1.4.

The cost model is key to estimating the cost of a given execution plan. To be accurate, the cost model must have good knowledge about the distributed execution environment. Important inputs are the database statistics and the formulas used to estimate the size of intermediate results. For simplicity, earlier cost models relied on the strong assumption that the distribution of attribute values in a relation is uniform. However, in case of skewed data distributions, this can result in fairly inaccurate estimations and execution plans which are far from the optimal. An effective solution to accurately capture data distributions is to use histograms. Today, most commercial DBMS optimizers support histograms as part of their cost model. A difficulty remains to estimate the selectivity of the join operation when it is not on foreign key. In this case, maintaining join selectivity factors is of great benefit [Mackert and Lohman, 1986]. Earlier distributed DBMSs considered transmission costs only. With the availability of faster communication networks, it is important to consider local processing costs as well.

The search strategy explores the search space and selects the best plan, using the cost model. It defines which plans are examined and in which order. The most popular search strategy is dynamic programming which enumerates all equivalent execution plans with some pruning. However, it may incur a high optimization cost for queries involving large number of relations. Thus, it is best suited when optimization is static (done at compile time) and amortized over multiple executions. Randomized strategies, such as Iterative Improvement and Simulated Annealing, have received much attention. They do not guarantee that the best solution is obtained, but avoid the high cost of optimization. Thus, they are appropriate for ad-hoc queries which are not repetitive.

As a prerequisite to understanding distributed query optimization, we have introduced centralized query optimization with the three basic techniques: dynamic, static and hybrid. Dynamic and static query optimimization both have advantages and drawbacks. Dynamic query optimization can make accurate optimization choices at run-time. but optimization is repeated for each query execution. Therefore, this approach is best for ad-hoc queries. Static query optimization, done at compilation time, is best for queries embedded in stored procedures, and has been adopted by all commercial DBMSs. However, it can make major estimation errors, in particular, in the case of parameter values not known until runtime, which can lead to the choice of suboptimal execution plans. 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 further optimization decisions may take place at run time.

Next, we have seen two approaches to solve distributed join queries, which are the most important type of queries. The first one considers join ordering. The second one computes joins with semijoins. Semijoins are beneficial only when a join has good selectivity, in which case the semijoins act as powerful size reducers. The first systems that make extensive use of semijoins assumed a slow network and therefore concentrated on minimizing only the communication time at the expense of local processing time. However, with faster networks, the local processing time is as important as the communication time and sometimes even more important. Therefore, semijoins should be employed carefully since they tend to increase the local processing time. Join and semijoin techniques should be considered complementary, not alternative [Valduriez and Gardarin, 1984], because each technique may be better under certain database-dependent parameters. For instance, if a relation has very large tuples, as is the case with multimedia data, semijoin is useful to minimize data transfers. Finally, semijoins implemented by hashed bit arrays [Valduriez, 1982] can be made very efficient [Mackert and Lohman, 1986].

We illustrated the use of the join and semijoin techniques in four basic distributed query optimization algorithms: dynamic, static, semijoin-based and hybrid. The static and dynamic distributed optimization approaches have the same advantages and disadvantages as in centralized systems. The semijoin-based approach is best for slow networks. The hybrid approach is best in today's dynamic environments as it delays important decisions such as copy selection and allocation of subqueries to sites at query startup time. Thus, it can better increase availability and load balancing of the system. We illustrated the hybrid approach with two-step query optimization which first generates a static plan that specifies the operations ordering as in a centralized system and then generates an execution plan at startup time, by carrying out site and copy selection and allocating the operations to the sites.

In this chapter we focused mostly on join queries for two reasons: join queries are the most frequent queries in the relational framework and they have been studied extensively. Furthermore, the number of joins involved in queries expressed in languages of higher expressive power than relational calculus (e.g., Horn clause logic) can be extremely large, making the join ordering more crucial [Krishnamurthy et al., 1986]. However, the optimization of general queries containing joins, unions, and aggregate functions is a harder problem [Selinger and Adiba, 1980]. Distributing unions over joins is a simple and good approach since the query can be reduced as a union of join subqueries, which are optimized individually. Note also that the unions are more frequent in distributed DBMSs because they permit the localization of horizontally fragmented relations.

8.6 Bibliographic Notes

Good surveys of query optimization are provided in [Graefe, 1993], [Ioannidis, 1996] and [Chaudhuri, 1998]. Distributed query optimization is surveyed in [Kossmann, 2000].

The three basic algorithms for query optimization in centralized systems are: the dynamic algorithm of INGRES [Wong and Youssefi, 1976] which performs query reduction, the static algorithm of System R [Selinger et al., 1979] which uses dynamic programming and a cost model and the hybrid algorithm of Volcano [Cole and Graefe, 1994] which uses choose-plan operators.

The theory of semijoins and their value for distributed query processing has been covered in [Bernstein and Chiu, 1981], [Chiu and Ho, 1980], and [Kambayashi et al., 1982]. Algorithms for improving the processing of semijoins in distributed systems are proposed in [Valduriez, 1982]. The value of semijoins for multiprocessor database machines having fast communication networks is also shown in [Valduriez and Gardarin, 1984]. Parallel execution strategies for horizontally fragmented databases is treated in [Ceri and Pelagatti, 1983] and [Khoshafian and Valduriez, 1987]. The solutions in [Shasha and Wang, 1991] are also applicable to parallel systems.

The dynamic approach to distributed query optimization was was first proposed for Distributed INGRES in [Epstein et al., 1978]. It extends the dynamic algorithm of INGRES, with a heuristic approach. The algorithm takes advantage of the network topology (general or broadcast networks). Improvements on this method based on the enumeration of all possible solutions are given and analyzed in [Epstein and Stonebraker, 1980].

The static approach to distributed query optimization was first proposed for R* in [Selinger and Adiba, 1980] as an extension of the static algorithm of System R. It is one of the first papers to recognize the significance of local processing on the performance of distributed queries. Experimental validation in [Lohman and Mackert, 1986] have confirmed this important statement.

The semijoin-based approach to distributed query optimization was proposed in [Bernstein et al., 1981] for SDD-1 [Wong, 1977]. It is one of the most complete algorithms which make full use of semijoins.

Several hybrid approaches based on two-step query optimization have been proposed for distributed systems [Carey and Lu, 1986; Du et al., 1995; Evrendilek et al., 1997]. The content of Section 8.4.4 is based on [Carey and Lu, 1986] which is the first paper on two-step query optimization. In [Du et al., 1995], efficient operations to transform linear join trees (produced by the first step) into bushy trees which exhibit more parallelism are proposed. In [Evrendilek et al., 1997], a solution to maximize intersite join parallelism in the second step is proposed.

Exercises

Problem 8.1 (*). Apply the dynamic query optimization algorithm in Section 8.2.1 to the query of Exercise 7.3, and illustrate the successive detachments and substitutions by giving the monorelation subqueries generated.

Problem 8.2. Consider the join graph of Figure 8.12 and the following information: $size(\text{EMP}) = 100$, $size(\text{ASG}) = 200$, $size(\text{PROJ}) = 300$, $size(\text{EMP} \bowtie \text{ASG}) = 300$, and $size(\text{ASG} \bowtie \text{PROJ}) = 200$. Describe an optimal join program based on the objective function of total transmission time.

Problem 8.3. Consider the join graph of Figure 8.12 and make the same assumptions as in Problem 8.2. Describe an optimal join program that minimizes response time (consider only communication).

Problem 8.4. Consider the join graph of Figure 8.12, and give a program (possibly not optimal) that reduces each relation fully by semijoins.

Problem 8.5 (*). Consider the join graph of Figure 8.12 and the fragmentation depicted in Figure 8.18. Also assume that $size(\text{EMP} \bowtie \text{ASG}) = 2000$ and $size(\text{ASG} \bowtie \text{PROJ}) = 1000$. Apply the dynamic distributed query optimization algorithm in Section 8.4.1 in two cases, general network and broadcast network, so that communication time is minimized.

Rel.	Site 1	Site 2	Site 3
EMP	1000	1000	1000
ASG		2000	
PROJ	1000		

Fig. 8.18 Fragmentation

Problem 8.6. Consider the join graph of Figure 8.19 and the statistics given in Figure 8.20. Apply the semijoin-based distributed query optimization algorithm in Section 8.4.3 with $T_{MSG} = 20$ and $T_{TR} = 1$.

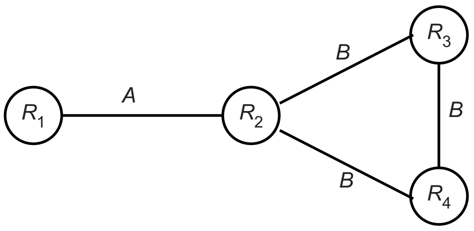


Fig. 8.19 Join Graph

(a)

relation	size
R_1	1000
R_2	1000
R_3	2000
R_3	1000

(b)

attribute	size	SF_{SJ}
$R_1.A$	200	0.5
$R_2.A$	100	0.1
$R_2.A$	100	0.2
$R_3.B$	300	0.9
$R_4.B$	150	0.4

Fig. 8.20 Relation Statistics

Problem 8.7 (**). Consider the query in Problem 7.5. Assume that relations EMP, ASG, PROJ and PAY have been stored at sites 1, 2, and 3 according to the table in Figure 8.21. Assume also that the transfer rate between any two sites is equal and that data transfer is 100 times slower than data processing performed by any site. Finally, assume that $size(R \bowtie S) = max(size(R), size(S))$ for any two relations R and S, and the selectivity factor of the disjunctive selection of the query in Exercise 7.5 is

0.5. Compose a distributed program which computes the answer to the query and minimizes total time.

Rel.	Site 1	Site 2	Site 3
EMP	2000		
ASG		3000	
PROJ			1000
PAY			500

Fig. 8.21 Fragmentation Statistics

Problem 8.8 (**). In Section 8.4.4, we described Algorithm 8.7 for linear join trees. Extend this algorithm to support bushy join trees. Apply it to the bushy join tree in Figure 8.3 using the data placement and site loads shown in Figure 8.17.