Chapter 9

OPTIMIZATION OF DISTRIBUTED QUERIES

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 unions. We also discuss how semijoin operations can help to process a join efficiently.

This chapter is organized as follows. In Section 9.1 we introduce the main components of query optimization, including tyhe search space, the search strategy and the cost model. Centralized query optimization is described in Section 9.2 as a prerequisite to understand distributed query optimization, which is more complex. In Section 9.3 we discuss the major optimization issue, which deals with the join ordering in fragment queries. We also examine alternative join strategies based on semijoin. In Section 9.4 we illustrate the use of the techniques and concepts in three basic distributed query optimization algorithms.

9.1 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. The selected plan minimizes an objective cost function. A query optimizer, the software module that

performs query optimization, is usually seen as three components: a search space, a cost model, and a search strategy (see Figure 9.1). The *search space* is the set of alternative execution plans to represent the input query. These plans are equivalent, in the sense that they yield the same result but they differ on the execution order of operations and the way these operations are implemented, and therefore on performance. The search space is obtained by applying transformation rules, such as those for relational algebra described in Section 8.1.4. 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.

9.1.1 Search Space

Query execution plans are typically abstracted by means of operator trees (see Section 8.1.4), 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*, operator trees whose operators are join or Cartesian product. This is because permutations of the join order have the most important effect on performance of relational queries.

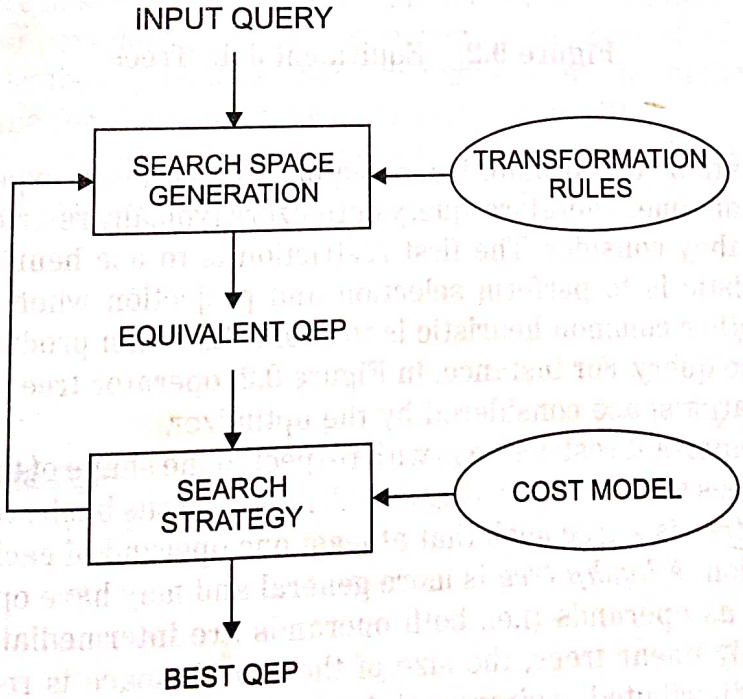


Figure 9.1. Query Optimization Process

Consider the following query:

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

Figure 9.2 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 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

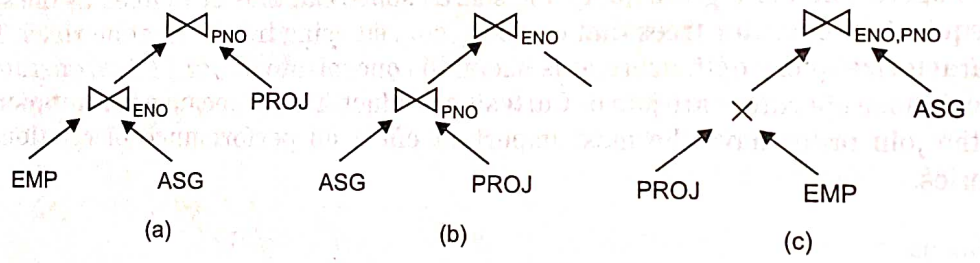


Figure 9.2. Equivalent Join Trees

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 9.2, 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 9.3). 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.

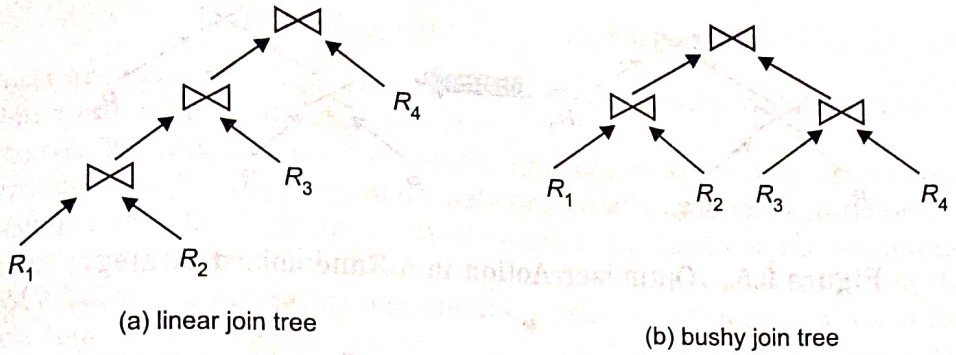


Figure 9.3. The Two Major Shapes of Join Trees

9.1.2 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 9.4. 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) 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 this reason, there has been recent interest in *randomized* strategies, 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 tune for execution time [Lanzelotte et al., 1993].

Randomized strategies, such as Iterative Improvement [Swami, 1989] and Simulated Annealing [Ioannidis and Wong, 1987] concentrate on searching the

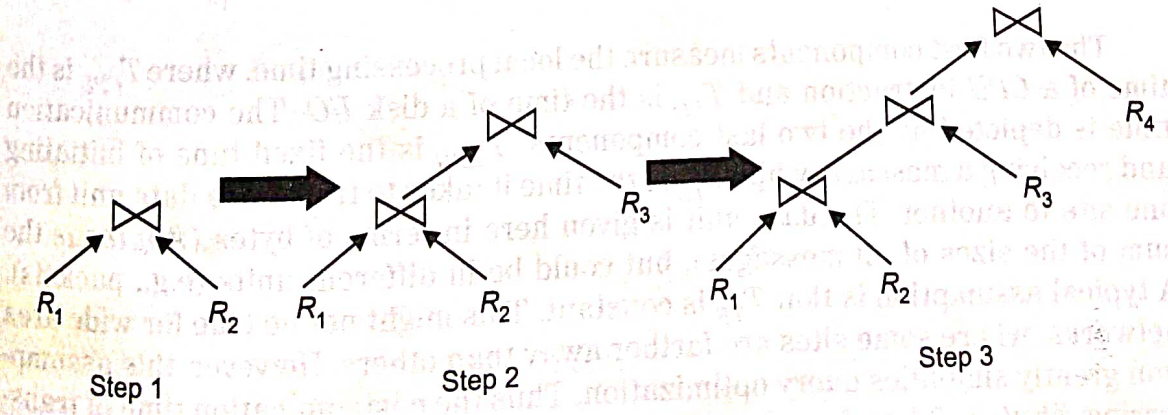


Figure 9.4. Optimizer Actions in a Deterministic Strategy

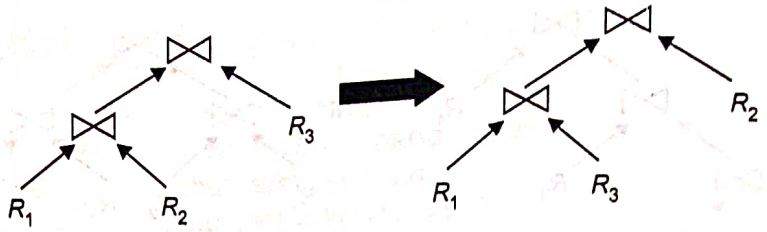


Figure 9.5. Optimizer Action in a Randomized Strategy

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 9.5. In [Lanzelotte et al., 1993], it is shown experimentally that randomized strategies provide better performance than deterministic strategies as soon as the query involves more than several relations.

9.1.3 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.

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]:

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

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 tune 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 (10Mbps) Ethernet [Page and Popek, 1985]. Thus, most early distributed DBMSs designed for wide area networks have ignored the local processing cost and concentrate 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 is 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

$$Response\_time = T_{CPU} * seq\_\#insts + T_{I/O} * seq\_\#I/O_s + T_{MSG} * seq\_\#msgs + T_{TR} * seq\_\#bytes$$

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 9.2

Let us illustrate the difference between total cost and response time using the example of Figure 9.6, which computes the answer to a query at site 3

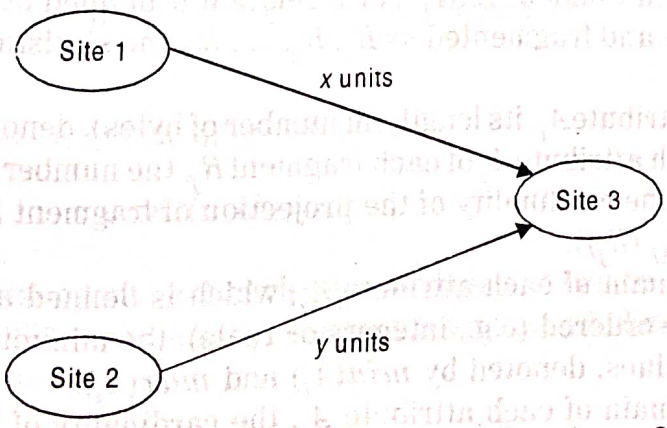


Figure 9.6. Example of Data Transfers for a Query

with data from sites 1 and 2. For simplicity, we assume that only communication cost is considered.

Assume that T_{MSG} and T_{TR} are expressed in time units. The total cost 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 9.4 we present algorithms that can optimize a combination of total time and response time, with more weight on one of them.

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 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 $card(\Pi_{Ai}(R_j))$.
2. For the domain of each attribute A_i, which is denned 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 $card(R_j)$.

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 selectivity in the former case and good selectivity in the latter case.

These statistics are useful to predict the size of intermediate relations. Remember that in Chapter 5 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.

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 $SFs(F)$ 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 Chapter 2, 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. Some systems, such as Distributed INGRES [Epstein et al., 1978], use this upper bound, which is quite pessimistic. R* [Selinger and Adiba, 1980] uses this upper bound divided by a constant to reflect the fact that the join result is smaller than that of the Cartesian product. 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 SFj 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 in [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.

9.2 CENTRALIZED QUERY OPTIMIZATION

In this section we present two of the most popular 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. Since we discussed in Chapter 8 some common techniques for query decomposition, we will concentrate on the optimization aspects used by two popular relational database systems: INGRES [Stonebraker et al., 1976] and System

R [Astrahan et al., 1979]. Furthermore, both systems have distributed versions (see Section 9.4) whose optimization algorithms are extensions of the centralized version.

The optimization techniques of these systems differ significantly (see [Gardarin and Valduriez, 1989] for more details). INGRES employs a dynamic optimization algorithm and System R uses a static optimization algorithm based on exhaustive search using statistics about the database. We note that most commercial relational DBMSs (see [Valduriez and Gardarin, 1989] for a survey) implement variants of the exhaustive search approach for its efficiency and compatibility with query compilation.

9.2.1 INGRES Algorithm

INGRES uses a dynamic query optimization algorithm [Wong and Youssefi, 1976] that recursively breaks up a calculus query into smaller pieces. It combines the two phases of calculus-algebra decomposition and optimization. A query is first decomposed into a sequence of queries having a unique relation (more precisely a unique tuple variable) in common. Then each monorelation query is processed by a "one-variable query processor" (OVQP). The OVQP optimizes the access to a single relation 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. However, if the predicate is of the form $< A \neq value >$, an index on A would not help, and sequential scan should be used.

We concentrate our presentation on the main query type, which is the "retrieve" command of the QUEL language [Stonebraker et al., 1976], which is used by INGRES and is similar to the "select" command of SQL. However, to maintain uniformity throughout the book, we use SQL to express our examples. 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 INGRES query processor 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   R2.A2, R3.A3, ... ,Rn.An
FROM     R1, R2, ... , Rn
WHERE    P1(R1.A'1)
AND      P2(R1, A1, R2.A2, ..., Rn.An)
```

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 $P_1.A_1$INTRO 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 $P_2(V_1.A_1, \ldots, V_n.A_n)$

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 9.3

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.END
      AND     ASG.PNO=JVAR.PNO
```

The successive detachments of q' may generate

```
q12 : SELECT  ASG.END  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_i has been reduced to the subsequent queries $q_{11} \rightarrow q_{12} \rightarrow q_{13}$. Query q_{11} is monorelation and can be performed by the OVQP. 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_{11} 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 variable are substituted by their values, thereby producing a set of $(n-1)$-variable 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 $t_1 i$, 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 9.4

Let us consider the query qi_3:

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

The relation defined by the variable 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 processed by the OVQP.

9.2.2 System R Algorithm

System R performs static query optimization based on the exhaustive search of the solution space [Selinger et al., 1979]. The input to the optimizer of System R is a relational algebra tree resulting from the decomposition of an SQL query. The output is an execution plan that implements the "optimal" relational algebra tree.

Instead of systematically executing the select operations before the joins as in INGRES, System R does so only if this leads to a better strategy. 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, automatically managed by System R. The cardinality of the intermediate results is estimated based on the operation selectivity factors (see Section 9.1.3).

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 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 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 loops*, composes the product of the two 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 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 pages, respectively, this algorithm has a cost proportional to $n_1 * n_2$, which may be prohibitive if n_1 and n_2 are high.

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 pages, 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 System R optimization algorithm, for a select-project-join query, is shown in Algorithm 9.2. 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 9.2 *R-QOA*

input: QT: query tree with n relations
output: *output: the result of execution*
begin
  for each relation $Ri \in QT$ **do**
  begin
    for each access path AP_{ij} to R_i **do**
      determine cost(AP_{ij})
    end-for
    $best\_AP_i \leftarrow AP_{ij}$ with minimum cost
  end-for
  for each order ($R_{i1}, R_{i2}, ..., R_{in}$) with i=1, ..., $n!$ **do**
  begin
    build strategy (...((best $AP_{i1} \bowtie R_{i2}) \bowtie R_{i3}) \bowtie ... \bowtie R_{in}$)
    compute the cost of strategy
  end-for
  output $\leftarrow$ strategy with minimum cost
end. { R-QOA }

Example 9.5

Let us illustrate this algorithm with the query q_1 (see Example 9.3) on the engineering database. The join graph of q_1 is given in Figure 9.7. For short,

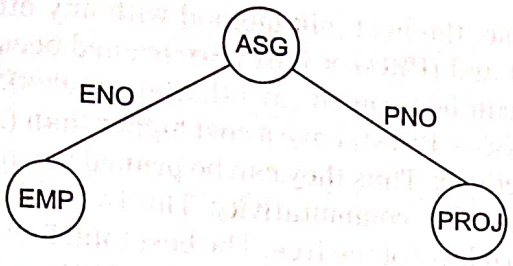


Figure 9.7. Join Graph of Query q_1

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

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 9.8. Note that the maximum number of join orders is 3!; dynamic search considers fewer alternatives, as depicted in Figure 9.8. The operations marked "pruned" are dynamically eliminated. The first level of the tree indicates the best single-relation access method. The second level indicates,

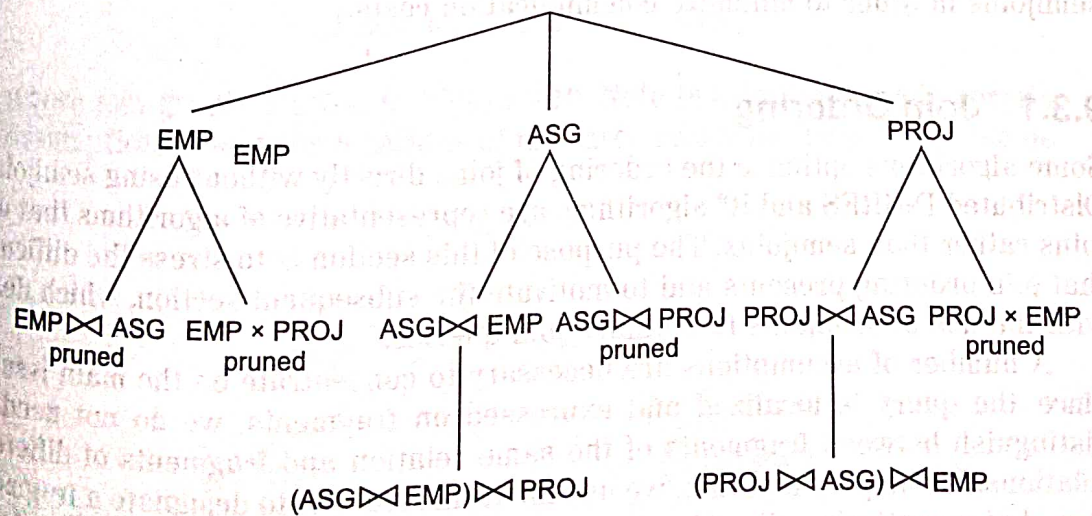


Figure 9.8. Alternative Join Orders

for each of these, the best join method with any other relation. Strategies (EMP × PROJ) and (PROJ × EMP) are pruned because they are Cartesian products that can be avoided (by other strategies). We assume that (EMP ⋈ ASG) and (ASG ⋈ PROJ) have a cost higher than (ASG ⋈ EMP) and (PROJ ⋈ 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 ⋈ EMP) ⋈ PROJ) and ((PROJ ⋈ ASG) ⋈ 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:

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

The performance measurement of System R [Mackert and Lohman, 1986] substantiates the important contribution of the CPU time to the total time of the query. 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.

9.3 JOIN ORDERING IN FRAGMENT QUERIES

As we have seen in Section 9.2, 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 fragment 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.

9.3.1 Join Ordering

Some algorithms optimize the ordering of joins directly without using semijoins. Distributed INGRES and R* algorithms are representative of algorithms that use joins rather than 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

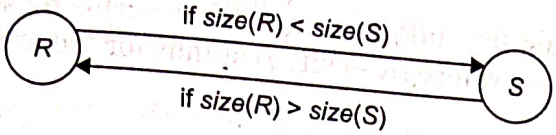


Figure 9.9. Transfer of Operands in Binary Operation

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 9.9. To make this choice we need to evaluate the size of R and of 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, the number of strategies grows rapidly with the number of relations. This approach, used by System R*, makes optimization costly, although this overhead is amortized rapidly if the query is executed frequently.

Example 9.6

Consider the following query expressed in relational algebra:

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

whose join graph is given in Figure 9.10. Note that we have made certain assumptions about the locations of the three relations. This query can be

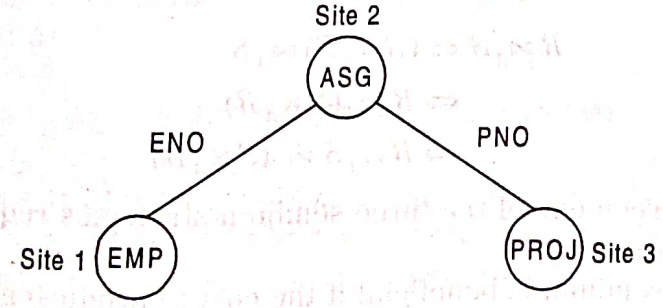


Figure 9.10. Join Graph of Distributed Query

executed in at least five different ways. We describe these strategies by the following programs, where (R → site j) stands for "relation R is transferred to site j."

1. EMP → site 2 Site 2 computes EMP′ = EMP ⋈ ASG EMP′ → site 3 Site 3 computes BMP′ ⋈ PROJ
2. ASG → site 1 Site 1 computes EMP′ = EMP ⋈ ASG EMP′ → site 3 Site 3 computes EMP′ ⋈ PROJ
3. ASG → site 3 Site 3 computes ASG′ = ASG ⋈ PROJ ASG′ → site 1 Site 1 computes ASG′ ⋈ BMP
4. PROJ → site 2 Site 2 computes PROJ′ = PROJ ⋈ ASG PROJ′ → site 1 Site 1 computes PROJ′ ⋈ EMP
5. EMP → site 2 PROJ → site 2 Site 2 computes EMP ⋈ PROJ ⋈ ASG

To select one of these programs, the following sizes must be known or predicted: *size*(EMP), *size*(ASG), *size*(PROJ), *size*(EMP ⋈ ASG), and *size*(ASG ⋈ 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 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.

9.3.2 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 is defined in [Bernstein and Chiu, 1981]. We are making the same assumptions as in Section 9.3.1. 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:

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

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 9.3.1, 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 9.10. 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 $size(\text{ASG} \ltimes \text{PROJ}) \leq size(\text{ASG})$, we have $size(\text{EMP}'') \leq size(\text{EMP}')$. 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 9.7

Consider the following relations, where attribute CITY has been added to relations EMP (renamed ET) and PROJ (renamed PT) of the engineering database:

ET(ENO, ENAME, TITLE, CITY)
ASG(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.

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

As illustrated in Figure 9.11a, this query is cyclic.

No full reducer exists for the query in Example 9.7. 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 predicate is preserved by transitivity [Kambayashi et al., 1982].

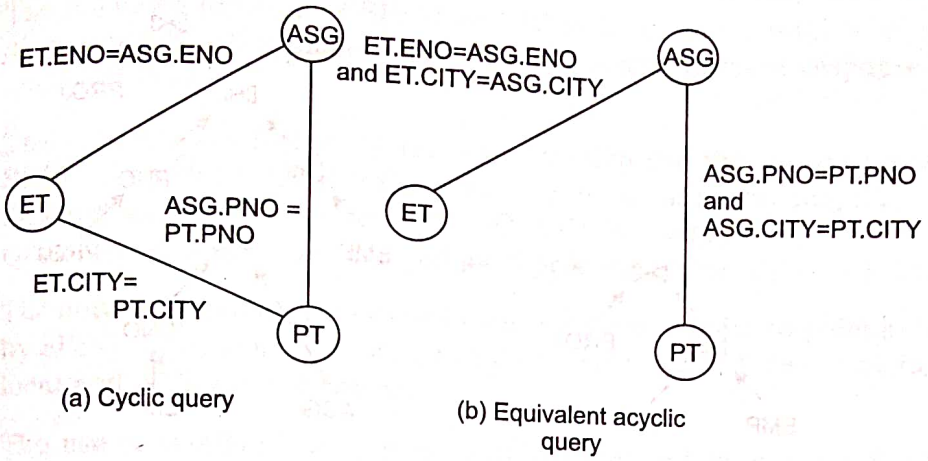


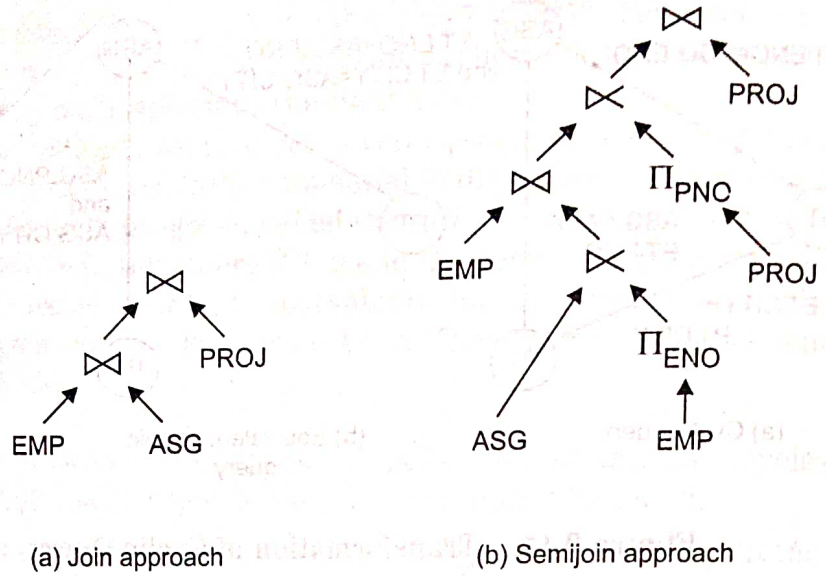
Figure 9.11. Transformation of Cyclic Query

In the example of Figure 9.11b, where the arc (ET, PT) is removed, the additional predicate ET.CITY = ASG.CITY and ASG.CITY = PT.CITY imply ET.CITY = PT.CITY by transitivity. Thus the acyclic query is equivalent to the cyclic query. The addition of these predicates implies the addition of attribute CITY in relation ASG. Hence, the values for attribute CITY must be sent from either ET or ASG.

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] and [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 9.10 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.

9.3.3 Join versus Semijoin

Compared with the join, the semijoin induces more operations but possibly on smaller operands. Figure 9.12 illustrates these differences with an equivalent pair of join and semijoin strategies for the query whose join graph is given in Figure 9.10. The join of two relations, EMP ⋈ ASG in Figure 9.10, 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 tune. 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



(a) Join approach (b) Semijoin approach

Figure 9.12. Join versus Semijoin Approaches

are accessed twice in Figure 9.12. 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].

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 $BA[h(val)\} = 1$.

9.4 DISTRIBUTED QUERY OPTIMIZATION ALGORITHMS

In this section we illustrate the use of the techniques presented previously in three basic query optimization algorithms: the reduction algorithm of Distributed INGRES [Epstein et al., 1978], System R* algorithm [Selinger et al., 1979], and SDD-1 algorithm [Bernstein et al., 1981]. We describe them because they are representative of different classes of algorithms and are therefore often used as paradigms. The differences among them can be specified in terms of the features introduced in Chapter 7 (specifically, Section 7.4):

1. The optimization timing is dynamic for distributed INGRES, while it is static for the others.

2. The objective function of SDD-1 and R* is to minimize total time, while distributed INGRES aims at decreasing a combination of response time and total time.

3. The optimization factors of the cost function are the message size for SDD-1. System R*, which takes into account local processing time, uses message size, number of messages, I/O, and CPU costs. Distributed INGRES considers both message size and local processing time (I/O + CPU time).

4. The network topology is assumed to be a wide area point-to-point network by SDD-1. The algorithms of distributed INGRES and R* can work in both local and wide area networks.

5. The use of semijoins as a query optimization technique is employed by SDD-1. Distributed INGRES and R* perform joins in a fashion similar to that of the centralized query optimization algorithms of their counterparts: INGRES and System R.

6. Each algorithm assumes statistical information about the data. As shown in Figure 9.13, semijoin algorithms typically use more information.

7. INGRES can handle fragments.

The differences between these three algorithms are summarized in Figure 9.13. In the rest of this section we detail each of these algorithms.

9.4.1 Distributed INGRES Algorithm

The query optimization algorithm of Distributed INGRES [Epstein et al., 1978] is derived from the algorithm used in centralized INGRES (see Section 9.2.1). Therefore, it consists of dynamically optimizing the processing strategy of a given query. The objective function of the algorithm is to minimize a combination of both the

Algorithms	Optm. Timing	Objective Function	Optm. Factors	Network Topology	Semi Joins	Stats†	Fragments
Dist. INGRES	Dynamic	Response time or total cost	Msg size, proc. cost	General or broadcast	No	1	Horizontal
R*	Static	Total cost	#Msg, Msg size, IO, CPU	General or local	No	1,2	No
SDD-1	Static	Total Cost	Msg size,	General	Yes	1,3,4,5	No

Figure 9.13. Comparison of Query Optimization Algorithms† 1 = relation cardinality, 2 = number of unique values per attribute, 3 = join selectivity factor, 4 = size of projection on each join attribute, 5 = attribute size and tuple size.

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 query processing algorithm 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 query processing 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). As in the centralized version, we describe the distributed query optimization algorithm for the case of a retrieval query. This algorithm is executed by the site, called the *master site*, where the query is initiated. The algorithm, which we call D-INGRES-QOA, is given in Algorithm 9.3.

Algorithm 9.3 *D-INGRES-QOA*

input: *MRQ:* multirelation query
output: result of the last multirelation query
begin
  for each detachable OVQ_i in *MRQ* **do** {run all one-relation queries}
    run(OVQ,);
  end-for (1)
  MVQ'_list ← REDUCE(MRQ) {replace MRQ by a list of n irreducible queries} (2)
  while $n \neq 0$ **do** {n is the number of irreducible queries} (3)
  begin
    {choose next irreducible query involving the smallest fragments}
    M RQ' ← SELECT_QUERY(*M RQ'_list*) (3.1)
    {determine fragments to transfer and processing site for *M RQ'*}
    Fragment-site-list ← SELECT_STRATEGY(M RQ') (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)
    end-for
    run(*M RQ'*)
    $n \leftarrow n - 1$ (3.4)
  end-while {output of the algorithm is the result of the last *MRQ'*}
end.{ D-INGRES-QOA}

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 9.2.1). Monorelation subqueries are ignored because they have already been processed in step (1). Thus the REDUCE procedure produces a sequence of irreducible subqueries $q_1 \rightarrow q_2 \rightarrow \ldots \rightarrow q_n$, with at most one relation in common between two consecutive subqueries. In [Wong and Youssefi, 1976], it is shown that such a sequence is unique. Example 9.3 (in Section 9.2.1), 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, MVQ', 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, the algorithm 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 $q_1 \rightarrow q_3$, $q_2 \rightarrow \theta_3$, 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. [Epstein et al., 1978] provides more details about this choice.

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 9.8

Assume that relations EMP, ASG, and PROJ of the query of Example 9.3 are stored as follows, where relation EMP is fragmented.

Site 1	Site 2
EMP_1 ASG	EMP_2 PROJ

There are several possible strategies, including the following:

1. Execute the entire query (EMP $\bowtie$ ASG $\bowtie$ 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 $size(EMP_1 \bowtie ASG) > size(EMP_1)$, 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 as in centralized INGRES. 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.

In [Epstein et al., 1978], formulas are given 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 $\max_{j=1,m}(\Sigma_{i=1}^{n} size(R_i^j)) > \max_{i=1,n}(size(R_i))$
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

$$CT_k \text{ (\#bytes)} = k * CT_1 \text{ (\#bytes)}$$

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 $\Sigma_{i \neq p}(size(R_i) - size(R_i^1)) > size\,(R_p^1)$

then

$k = 1$

else

k is the largest j such that $\Sigma_{i \neq p}(size(R_i) - size(R_i^j)) \leq size\,(R_p^j)$

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 Rp.

Example 9.9

Let us consider the query PROJ ⋈ 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.

The algorithm of Distributed INGRES 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. An alternative to the limited search is the exhaustive search approach (used by R*), where all possible strategies are evaluated to find the best one. In [Epstein and Stonebraker, 1980], the two approaches are simulated and compared on the basis of the size of data transfers. An important conclusion of this study is that exhaustive search significantly outperforms limited search as soon as the query accesses more than three relations. Another conclusion is that dynamic optimization is beneficial because the exact sizes of the intermediate results are known.

9.4.2 R* Algorithm

The distributed query optimization algorithm of R* ([Selinger and Adiba, 1980], [Lohman et al., 1985]) is a substantial extension of the techniques developed for System R's optimizer (see Section 9.2.2). Therefore, it uses a compilation approach where an exhaustive search of all alternative strategies is performed in order to choose the one with the least cost. Although predicting and enumerating these strategies is costly, the overhead of exhaustive search is rapidly amortized if the query is executed frequently. Although the algorithm described in [Selinger and Adiba, 1980] deals with fragmentation, the implemented version of R* supports neither fragmentation nor replication. Therefore, the R* query processing algorithm deals only with relations as basic units. Query compilation is a distributed task in R*, 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 System R*'s optimizer is the general total time function, including local processing and communications costs (see Section 9.1.1).

We now summarize the query optimization algorithm of R*. 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 R*-QOA in Algorithm 9.4.

Algorithm 9.4 *R*-QOA*

```
input: QT: query tree
output: strat: minimum cost strategy
begin
    for each relation R_i ∈ QT do
    begin
        for each access path AP_ij to R_i do
            determine cost(AP_ij)
        end-for
        best-AP_i ← AP_ij with minimum cost
```

```
    end
    for each order (R_i1, R_i2, ... , R_in) with i = 1, ... , n! do
    begin
        build strategy (...(( best AP_i1, ... , ⋈ R_i2) ⋈ R_i3) ⋈ ... ⋈ R_in)
        compute the cost of strategy
    end-for
    strat ← strategy with minimum cost
    for each site k storing a relation involved in QT do
    begin
        LS_k ← local strategy (strategy, k)
        send (LS_k, site k) {each local strategy is optimized at site k}
    end-for
end. { R*-QOA }
```

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). In R*, 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. R* 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 5. Then the s tuples of 5 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 9.10

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 R* algorithm predicts the total time of each strategy and selects the cheapest. Given that there is no operation following the join PROJ $\bowtie$ 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 R* algorithm significantly reduces the number of alternatives by using dynamic programming and the heuristics, as does the System R's optimizer (see Section 9.2.2). With dynamic programming, the tree of alternatives is dynamically constructed and pruned by eliminating the inefficient choices.

In [Lohman and Mackert, 1986] and [Mackert and Lohman, 1986], an instructive performance evaluation of the R* optimizer is described in the context of both high-speed networks (similar to local networks) and medium-speed wide area networks. The tests confirm the significant contribution of local processing costs, even for wide area networks. It is shown in particular that for the distributed join, transferring the entire internal relation outperforms the fetch-as-needed method.

9.4.3 SDD-1 Algorithm

The query optimization algorithm of SDD-1 [Bernstein et al., 1981] 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 this 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 ESo 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 hiil-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 9.11

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

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

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 $size(EMP \bowtie PAY) = size(EMP)$, $size(PROJ \bowtie ASG) = 2 * size(PROJ)$, and $size(ASG \bowtie EMP) = size(ASG)$.

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

$$\begin{aligned} ES_0: \quad & EMP \rightarrow \text{site } 4 \\ & PAY \rightarrow \text{site } 4 \\ & PROJ \rightarrow \text{site } 4 \\ & Total\_cost(ES_0) = 4 + 8 + 1 = 13 \end{aligned}$$

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

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

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:

$$\begin{aligned} ES_1: \quad & EMP \rightarrow \text{site } 2 \\ ES_2: \quad & (EMP \bowtie PAY) \rightarrow \text{site } 4 \\ ES_3: \quad & PROJ \rightarrow \text{site } 4 \\ & Total\_cost(ES') = 8 + 8 + 1 = 17 \end{aligned}$$

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 hill-climbing algorithm has been substantially improved in SDD-1 [Bernstein et al. 1981] in a number of ways. The improved version "makes extensive use of semijoins. The objective function is expressed in terms of total communication time (local time and response time are not considered). Finally, the algorithm uses statistics on the database, called *database profiles*, where a profile is associated with a relation. The improved version also selects an initial feasible solution that is iteratively refined. Furthermore, 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 SDD-1 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. The algorithm is detailed in Algorithm 9.5 by the procedure SDD-1-QOA.

Algorithm 9.5 *SDD-1-QOA*

input: *QG*: query graph with n relations; statistics for each relation
output: *ES*: execution strategy
begin
ES ← local-operations (*QG*);
modify statistics to reflect the effect of local processing
BS ← ϕ
for each semijoin $S\ J$ in *QG* **do** {set of beneficial semijoins}
if cost(SJ) < *benefit*(S J) **then**

```
            BS ← BS ∪ SJ
        end-if
    end-for
    while BS ≠ ϕ do                                  {selection of beneficial semijoins}
    begin
    SJ ← most_beneficial(BS)          {SJ: semijoin with max(benefit_cost)}
    BS ← BS – SJ                                          {remove SJ from BS}
    ES ← ES + SJ                                  {append SJ to execution strategy}
    modify statistics to reflect the effect of incorporating SJ
    BS ← BS – nonbeneficial semijoins
    BS ← BS∪ new beneficial semijoins
end-while
    {assembly site selection}
    AS(ES) ← select site i such that i stores the largest amount
        of data after all local operations
    ES ← ES ∪ 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 Rj do
            if cost(ES) > cost(ES – SJ) then
                ES ← ES – SJ
            end-if
        end-for
    end-for
end. { SDD-1-QOA }
```

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.

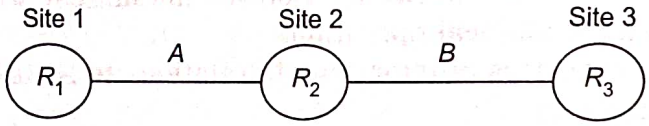
The next phase selects the assembly site by evaluating, for each candidate site, the cost of transferring to it all the required data and taking the one with the least cost. Finally, a postoptimization phase permits the removal from the execution strategy of those semijoins that affect only relations 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 9.12

Let us consider the following query:

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

Figure 9.14 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:



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
$R_2.B$	1.0	400
$R_3.B$	0.4	80

Figure 9.14. Example Query and Statistics

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 nonbeneficial 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 P is also reduced to $96 = 320 * 0.3$.

At the second iteration, there are two beneficial semijoins:

SJ_2: $Rz \ltimes R_3$, whose benefit is $540 = 900 * (1 - 0.4)$ and cost is 200 (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.

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 $\Pi_{R1:A}$.
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 SDD-1 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.

REVIEW QUESTIONS

9.1 What is query optimization?
9.2 Explain distributed cost model with an example.
9.3 What do you mean by cardinality of selection?
9.4 Explain centralized query optimization.
9.5 Give an algorithm for strategy with minimum cost.
9.6 Explain in detail join ordering in fragment queries.
9.7 Give an example for join graph of distributed query.
9.8 Explain semijoin-based algorithms.
9.9 Explain distributed query optimization algorithms.