Chapter 8

QUERY DECOMPOSITION AND DATA LOCALIZATION

This chapter is organized as follows. In Section 8.1 we present the four successive phases of query decomposition: normalization, semantic analysis, simplification, and restructuring of the query. In Section 8.2 we describe data localization, with emphasis on reduction and simplification techniques for the four following types of fragmentation: horizontal, vertical, derived, and hybrid.

8.1 QUERY DECOMPOSITION

Query decomposition (see Figure 7.3) is the first phase of query processing that transforms a relational calculus query into a relational algebra query. Both input and output queries refer to global relations, without knowledge of the distribution of data. Therefore, query decomposition is the same for centralized and distributed systems [Gardarin and Valduriez, 1989]. In this section the input query is assumed to be syntactically correct. When this phase is completed successfully the output query is semantically correct and good in the sense that redundant work is avoided. The successive steps of query decomposition are (1) normalization, (2) analysis, (3) elimination of redundancy, and (4) rewriting. Steps 1, 3, and 4 rely on the fact that various transformations are equivalent for a given query, and some can have better performance than others. We present the first three steps in the context of tuple relational calculus (e.g., SQL). Only the last step rewrites the query into relational algebra.

8.1.1 Normalization

The input query may be arbitrarily complex, depending on the facilities provided by the language. It is the goal of normalization to transform the query to a normalized form to facilitate further processing. With relational languages such as SQL, the most important transformation is that of the query qualification (the WHERE clause), which may be an arbitrarily complex, quantifier-free predicate, preceded by all necessary quantifiers ($\forall$ or $\exists$). There are two possible normal forms for the predicate, one giving precedence to the AND ($\wedge$) and the other to the OR

($\vee$). The conjunctive normal form is a conjunction ($\wedge$ predicate) of disjunctions ($\vee$ predicates) as follows:

$$(p_{11} \vee p_{12} \vee \ldots p_{1n}) \wedge \ldots \wedge (p_{m1} \vee p_{m2} \vee \ldots \vee p_{mn})$$

where p_{ij} is a simple predicate. A qualification in *disjunctive normal form*, on the other hand, is as follows:

$$(p_{11} \wedge p_{12} \wedge \ldots p_{1n}) \vee \ldots \vee (p_{m1} \wedge p_{m2} \wedge \ldots \wedge p_{mn})$$

The transformation of the quantifier-free predicate is straightforward using the well-known equivalence rules for logical operations ($\wedge$, $\vee$, and $\neg$):

1. $p_1 \wedge p_2 \Leftrightarrow p_2 \wedge p_1$
2. $p_1 \vee p_2 \Leftrightarrow p_2 \vee p_1$
3. $p_1 \wedge (p_2 \wedge p_3) \Leftrightarrow (p_1 \wedge p_2) \wedge p_3$
4. $p_1 \vee (p_2 \vee p_3) \Leftrightarrow (p_1 \vee p_2) \vee p_3$
5. $p_1 \wedge (p_2 \vee p_3) \Leftrightarrow (p_1 \wedge p_2) \vee (p_1 \wedge p_3)$
6. $p_1 \vee (p_2 \wedge p_3) \Leftrightarrow (p_1 \vee p_2) \wedge (p_1 \vee p_3)$
7. $\neg(p_1 \wedge p_2) \Leftrightarrow \neg p_1 \vee \neg p_2$
8. $\neg(p_1 \vee p_2) \Leftrightarrow \neg p_1 \wedge \neg p_2$
9. $\neg(\neg p_1) \Leftrightarrow p$

In the disjunctive normal form, the query can be processed as independent conjunctive subqueries linked by unions (corresponding to the disjunctions). However, this form may lead to replicated join and select predicates, as shown in the following example. The reason is that predicates are very often linked with the other predicates by AND. The use of rule 5 mentioned above, with p_1 as a join or select predicate, would result in replicating p_1. The conjunctive normal form is more practical since query qualifications typically include more AND than OR predicates. However, it leads to predicate replication for queries involving many disjunctions and few conjunctions, a rare case.

Example 8.1

Let us consider the following query on the engineering database that we have been referring to:

> "Find the names of employees who have been working on project P_1 for 12 or 24 months"

The query expressed in SQL is

```
SELECT  ENAME
FROM    EMP, ASG
WHERE   EMP.ENO = ASG.ENO
AND     ASG.PNO = "P1"
AND     DUR = 12 OR DUR = 24
```

The qualification in conjunctive normal form is

$$\text{EMP.ENO} = \text{ASG.ENO} \wedge \text{ASG.PNO} = \text{“}P_1\text{”} \wedge (\text{DUR} = 12 \vee \text{DUR} = 24)$$

while the qualification in disjunctive normal form is

$$(\text{EMP.ENO} = \text{ASG.ENO} \wedge \text{ASG.PNO} = \text{“}P_1\text{”} \wedge \text{DUR} = 12) \vee$$
$$(\text{EMP.ENO} = \text{ASG.ENO} \wedge \text{ASG.PNO} = \text{“}P_1\text{”} \wedge \text{DUR} = 24)$$

In the latter form, treating the two conjunctions independently may lead to redundant work if common subexpressions are not eliminated.

8.1.2 Analysis

Query analysis enables rejection of normalized queries for which further processing is either impossible or unnecessary. The main reasons for rejection are that the query is *type incorrect* or *semantically incorrect*. When one of these cases is detected, the query is simply returned to the user with an explanation. Otherwise, query processing is continued. Below we present techniques to detect these incorrect queries.

A query is type incorrect if any of its attribute or relation names are not defined in the global schema, or if operations are being applied to attributes of the wrong type. The technique used to detect type incorrect queries is similar to type checking for programming languages. However, the type declarations are part of the global schema rather than of the query, since a relational query does not produce new types.

Example 8.2

The following SQL query on the engineering database

```
SELECT  E#
FROM    EMP
WHERE   ENAME > 200
```

is type incorrect for two reasons. First, attribute E# is not declared in the schema. Second, the operation ">200" is incompatible with the type string of ENAME.

A query is semantically incorrect if components of it do not contribute in any way to the generation of the result. In the context of relational calculus, it is not

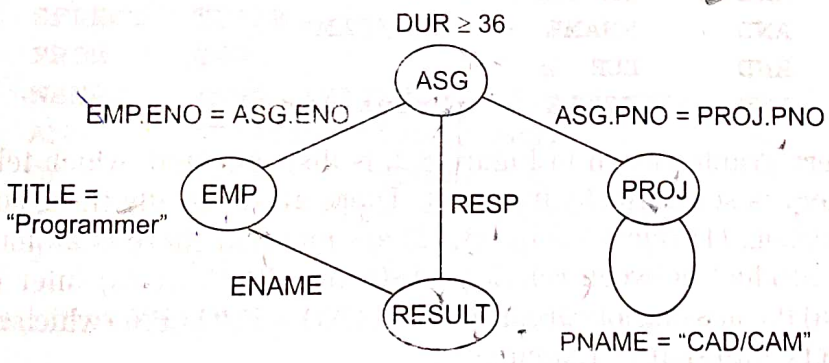
possible to determine the semantic correctness of general queries. However, it is possible to do so for a large class of relational queries, those which do not contain disjunction and negation [Rosenkrantz and Hunt, 1980]. This is based on the representation of the query as a graph, called a *query graph* or *connection graph* [Ullman, 1982]. We define this graph for the most useful kinds of queries involving select, project, and join operators. In a query graph, one node indicates the result relation, and any other node indicates an operand relation. An edge between two nodes that are not results represents a join, whereas an edge whose destination node is the result represents a project. Furthermore, a nonresult node may be labeled by a select or a self-join (join of the relation with itself) predicate. An important subgraph of the relation connection graph is the *join graph*, in which only the joins are considered. The join graph is particularly useful in the query optimization phase.

Example 8.3

Let us consider the following query:

> "Find the names and responsibilities of programmers who have been working on the CAD/CAM project for more than 3 years, and their manager's name"

The query expressed in SQL is



(a) Query graph

(b) Corresponding join graph

Figure 8.1. Relation Graphs

```
SELECT  ENAME, RESP
FROM    EMP, ASG, PROJ
WHERE   EMP.ENO = ASG.END
AND     ASG.PNO = PROJ.PNO
AND     PNAME = "CAD/CAM"
AND     DOR ≥ 36
AND     TITLE = "Programmer"
```

The query graph for the query above is shown in Figure 8.1a. Figure 8.1b shows the join graph for the graph in Figure 8.1a.

The query graph is useful to determine the semantic correctness of a conjunctive multivariable query without negation. Such a query is semantically incorrect if its query graph is not connected. In this case one or more subgraphs (corresponding to subqueries) are disconnected from the graph that contains the result relation. The query could be considered correct (which some systems do) by considering the missing connection as a Cartesian product. But, in general, the problem is that join predicates are missing and the query should be rejected.

Example 8.4

Let us consider the following SQL query:

```
SELECT  ENAME, RESP
FROM    EMP, ASG, PROJ
WHERE   EMP.ENO = ASG.END
AND     PNAME = "CAD/CAM"
AND     DUR ≥ 36
AND     TITLE = "Programmer"
```

Its query graph, shown in Figure 8.2, is disconnected, which tells us that the query is semantically incorrect. There are basically three solutions to the problem: (1) reject the query, (2) assume that there is an implicit Cartesian product between relations ASG and PROJ, or (3) infer (using the schema) the missing join predicate ASG.PNO = PROJ.PNO which transforms the query into that of Example 8.3.

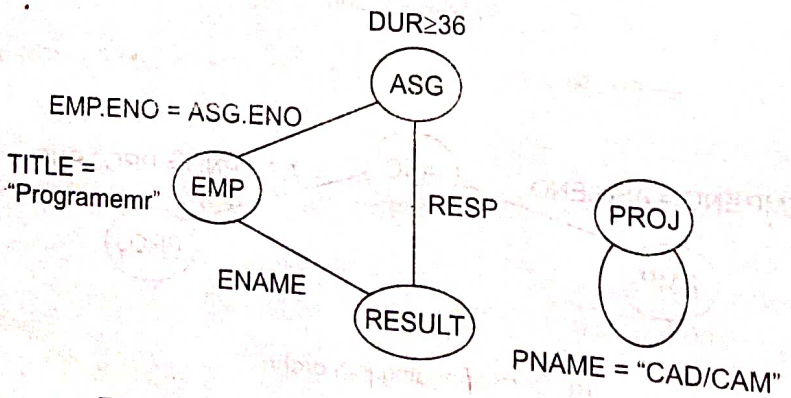


Figure 8.2. Disconnected Query Graph

8.1.3 Elimination of Redundancy

As we saw in Chapter 6, relational languages can be used uniformly for semantic data control. In particular, a user query typically expressed on a view may be enriched with several predicates to achieve view-relation correspondence, and ensure semantic integrity and security. The enriched query qualification may then contain redundant predicates. A naive evaluation of a qualification with redundancy can well lead to duplicated work. Such redundancy and thus redundant work may be eliminated by simplifying the qualification with the following well-known idempotency rules:

1. $p \wedge p \Leftrightarrow p$
2. $p \vee p \Leftrightarrow p$
3. $p \wedge true \Leftrightarrow p$
4. $p \wedge false \Leftrightarrow p$
5. $p \wedge false \Leftrightarrow$ false
6. $p \vee true \Leftrightarrow$ true
7. $p \wedge p \Leftrightarrow$ false
8. $p \vee p \Leftrightarrow$ true
9. $p_1 \wedge (p_1 \vee p_2) \Leftrightarrow p_1$
10. $p_1 \vee (p_1 \wedge p_2) \Leftrightarrow p_1$

Example 8.5

The SQL query

```
SELECT   TITLE
FROM     EMP
WHERE    (NOT  (TITLE  =  "Programmer")
AND      (TITLE=  "Programmer"
OR       TITLE  =  "Elect.  Eng.")
AND      NOT  (TITLE  =  Elect.  Eng."))
OR       ENAME  =  "J.  Doe"
```

can be simplified using the previous rules to become

```
SELECT   TITLE
FROM     EMP
WHERE    ENAME  =  "J.  Doe"
```

The simplification proceeds as follows. Let p_1 be <TITLE = "Programmer">, p_2 be <TITLE = "Elect. Eng.">, and p_3 be <ENAME = "J. Doe">. The query qualification is

$$(\neg p_1 \wedge (p_1 \vee p_2) \wedge \neg p_2) \vee p_3$$

The disjunctive normal form for this qualification is obtained by applying rule 5 defined in Section 8.1.1, which yields

$$(\neg p_1 \wedge ((p_1 \wedge \neg p_2) \vee (p_2 \wedge \neg p_2))) \vee p_3$$

and then rule 3 defined in Section 8.1.1, which yields

$$(\neg p_1 \wedge p_1 \wedge \neg p_2) \vee (\neg p_1 \wedge p_2 \wedge \neg p_2) \vee p_3$$

By applying rule 7 defined above, we obtain

$$(false \wedge \neg p_2) \vee (\neg p_1 \wedge false) \vee p_3$$

By applying the same rule, we get

$$false \vee false) \vee p_3$$

which is equivalent to p_3 by rule 4 above.

8.1.4 Rewriting

The last step of query decomposition rewrites the query in relational algebra. This is typically divided into the following two substeps: (1) straightforward transformation of the query from relational calculus into relational algebra, and (2) restructuring of the relational algebra query to improve performance. For the sake of clarity it is customary to represent the relational algebra query graphically by an operator tree. An operator tree is a tree in which a leaf node is a relation stored in the database, and a nonleaf node is an intermediate relation produced by a relational algebra operator. The sequence of operations is directed from the leaves to the root, which represents the answer to the query.

The transformation of a tuple relational calculus query into an operator tree can easily be achieved as follows. First, a different leaf is created for each different tuple variable (corresponding to a relation). In SQL, the leaves are immediately available in the FROM clause. Second, the root node is created as a project operation involving the result attributes. These are found in the SELECT clause in SQL. Third, the qualification (SQL WHERE clause) is translated into the appropriate sequence of relational operations (select, join, union, etc.) going from the leaves to the root. The sequence can be given directly by the order of appearance of the predicates and operators.

Example 8.6

The query

"Find the names of employees other than J. Doe who worked on the CAD/CAM project for either one or two years" whose SQL expression is

```
SELECT  ENAME
FROM    PROJ, ASG, EMP
WHERE   ASG.END = EMP.END)
AND     ASG.PNO = PROJ.PNO
AND     ENAME ≠ "J.Doe"
AND     PROJ.PNAME = "CAD/CAM"
AND     (DUR = 12 OR DUR =24)
```

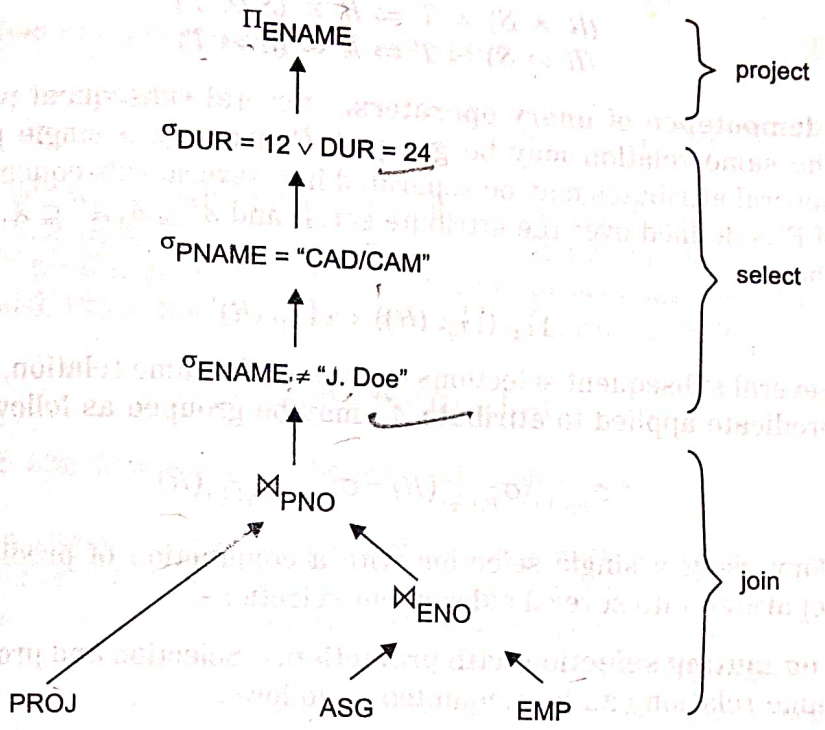


Figure 8.3. Example of Operator Tree

can be mapped in a straightforward way in the tree in Figure 8.3. The predicates have been transformed in order of appearance as join and then select operations.

By applying *transformation rules*, many different trees may be found equivalent to the one produced by the method described above [Smith and Chang, 1975]. The six most useful equivalence rules, which concern the basic relational algebra operators, are now presented. The correctness of these rules is given in [Ullman, 1982].

In the remainder of this section, R, S, and T are relations where R is defined over attributes $A = \{A_1, A_2, \ldots, A_n\}$ and S is defined over $B = \{B_1, B_2, \ldots, B_n\}$.

1. **Commutativity of binary operators.** The Cartesian product of two relations R and S is commutative:

$$R \times S \Leftrightarrow S \times R$$

Similarly, the join of two relations is commutative:

$$R \bowtie S \Leftrightarrow S \bowtie R$$

This rule also applies to union but not to set difference or semijoin.

2. **Associativity of binary operators.** The Cartesian product and the join are associative operators:

$$(R \times S) \times T \Leftrightarrow R \times (S \times T)$$
$$(R \bowtie S) \bowtie T \Leftrightarrow R \bowtie (S \bowtie T)$$

3. **Idempotence of unary operators.** Several subsequent projections on the same relation may be grouped. Conversely, a single projection on several attributes may be separated into several subsequent projections. If R is defined over the attribute set A, and $A' \subseteq A$, $A'' \subseteq A$, and $A' \subseteq A''$, then

$$\Pi_{A'}(\Pi_{A''}(R)) \Leftrightarrow \Pi_{A'}(R)$$

Several subsequent selections $\sigma_{pi}(A_i)$ on the same relation, where p_1 is a predicate applied to attribute A_i, may be grouped as follows:

$$\sigma_{p_1(A_1)}(\sigma_{p_2(A_2)}(R) = \sigma_{p_1(A_1) \wedge p_2(A_2)}(R)$$

Conversely, a single selection with a conjunction of predicates may be separated into several subsequent selections.

4. **Commuting selection with projection.** Selection and projection on the same relation can be commuted as follows:

$$\Pi_{A_1,\ldots,A_n}(\sigma_{p_1(A_p)}(R)) \Leftrightarrow \Pi_{A_1,\ldots,A_n}(\sigma_{p}(A_p)(\Pi_{A_1,\ldots,A_n,A_p}(R)))$$

Note that if A_p is already a member of $\{A_1, \ldots, A_n\}$, the last projection on $[A_1, \ldots, A_n]$ on the right-hand side of the equality is useless.

5. **Commuting selection with binary operators.** Selection and Cartesian product can be commuted using the following rule (remember that attribute A_i belongs to relation R):

$$\sigma_{p(A_i)}(R \times S) \Leftrightarrow (\sigma_{p(A_i)}(R)) \times S$$

Selection and join can be commuted:

$$\sigma_{p(A_i)}(R \bowtie_{p(A_j,B_k)} S) \Leftrightarrow \sigma_{p(A_i)}(R) \bowtie_{p(A_j,B_k)} S$$

Selection and union can be commuted if R and T are union compatible (have the same schema):

$$\sigma_{p(A_i)}(R \cup T) \Leftrightarrow \sigma_{p(A_i)}(R) \cup \sigma_{p(A_i)}(T)$$

Selection and difference can be commuted in a similar fashion.

6. **Commuting projection with binary operators.** Projection and Cartesian product can be commuted. If $C = A' \cup B'$, where $A' \subseteq A$, $B' \subseteq B$, and A and B are the sets of attributes over which relations R and S, respectively, are defined, we have

$$\Pi_C(R \times S) \Leftrightarrow \Pi_{A'}(R) \times \Pi_{B'}(S)$$

Projection and join can also be commuted.

$$\Pi_C(R \bowtie_{p(A_i,B_j)} S) \Leftrightarrow \Pi_{A'}(R) \bowtie_{p(A_i,B_j)} \Pi_{B'}(S)$$

For the join on the right-hand side of the implication to hold we need tc have $A_i \in A'$ and $B_j \in B'$. Since $C = A' \cup B'$, A_i and B_j are in C and therefore we don't need a projection over C once the projections over A and B' are performed. Projection and union can be commuted as follows

$$\Pi_C(R \cup S) \Leftrightarrow \Pi_C(R) \cup \Pi_C(S)$$

Projection and difference can be commuted similarly.

The application of these six rules enables the generation of many equivalent trees. For instance, the tree in Figure 8.4 is equivalent to the one in Figure 8.3. However, the one in Figure 8.4 requires a Cartesian product of relations EMP and PROJ, and may lead to a higher execution cost than the original tree. In the optimization phase, one can imagine comparing all possible trees based on their predicted cost. However, the excessively large number of possible trees makes this approach unrealistic. The rules presented above can be used to restructure the tree in a systematic way so that the "bad" operator trees are eliminated. These rules can be used in four different ways. First, they allow the separation of the unary operations, simplifying the query expression. Second, unary operations on the same relation may be grouped so that access to a relation for performing unary operations can be done only once. Third, unary operations can be commuted with binary operations so that some operations (e.g., selection) may be done first. Fourth, the binary operations can be ordered. This last rule is used extensively in query optimization. A simple restructuring algorithm, presented in [Ullman, 1982], uses a single heuristic that consists of applying unary operations (select/project) as soon as possible to reduce the size of intermediate relations.

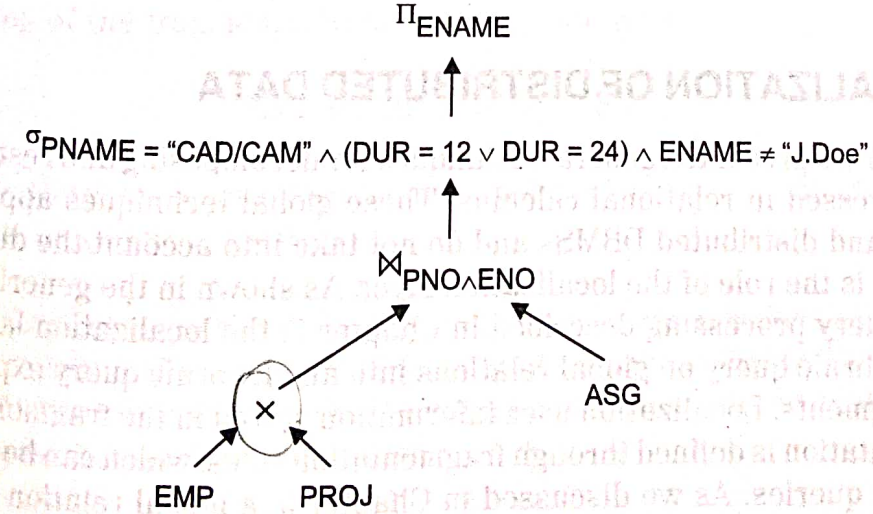


Figure 8.4. Equivalent Operator Tree

Example 8.7

The restructuring of the tree in Figure 8.3 leads to the tree in Figure 8.5. The resulting tree is good in the sense that repeated access to the same relation (as in Figure 8.3) is avoided and that the most selective operations are done first. However, this tree is far from optimal. For example, the select operation on EMP is not very useful before the join because it does not greatly reduce the size of the operand relation.

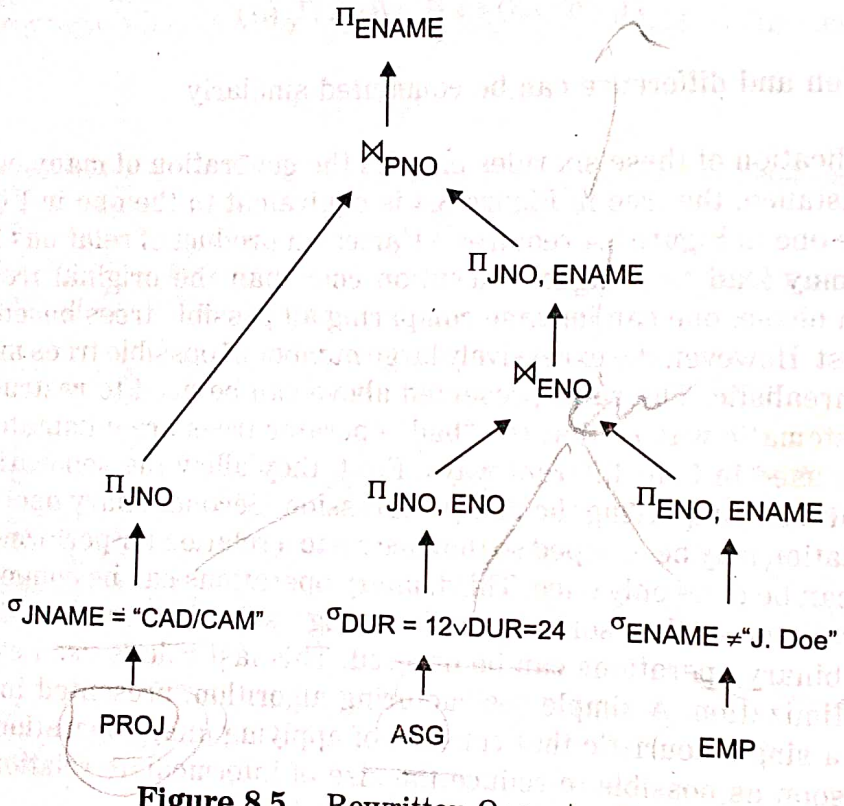


Figure 8.5. Rewritten Operator Tree

8.2 LOCALIZATION OF DISTRIBUTED DATA

In Section 8.1 we presented general techniques for decomposing and restructuring queries expressed in relational calculus. These global techniques apply to both centralized and distributed DBMSs and do not take into account the distribution of data. This is the role of the localization layer. As shown in the generic layering scheme of query processing described in Chapter 7, the localization layer translates an algebraic query on global relations into an algebraic query expressed on physical fragments. Localization uses information stored in the fragment schema.

Fragmentation is defined through fragmentation rules, which can be expressed as relational queries. As we discussed in Chapter 5, a global relation can be reconstructed by applying the reconstruction (or reverse fragmentation) rules and deriving a relational algebra program whose operands are the fragments. We call

this a *localization program*. To simplify this section, we do not consider the fact that data fragments may be replicated, although this can improve performance. Replication is considered in Chapter 9.

A naive way to localize a distributed query is to generate a query where each global relation is substituted by its localization program. This can be viewed as replacing the leaves of the operator tree of the distributed query with subtrees corresponding to the localization programs. We call the query obtained this way the generic query. In general, this approach is inefficient because important restructurings and simplifications of the generic query can still be made ([Ceri and Pelagatti, 1983], [Ceri et al., 1986]). In the remainder of this section, for each type of fragmentation we present reduction techniques that generate simpler and optimized queries. We use the transformation rules and the heuristics, such as pushing unary operations down the tree, that were introduced in Section 8.1.4.

8.2.1 Reduction for Primary Horizontal Fragmentation

The horizontal fragmentation function distributes a relation based on selection predicates. The following example is used in subsequent discussions.

Example 8.8

Relation EMP(ENO, ENAME, TITLE) of Figure 2.4 can be split into three horizontal fragments EMP_1, EMP_2, and EMP_3, defined as follows:

$$EMP_1 = \sigma_{ENO \leq \text{“E3”}}(EMP)$$
$$EMP_2 = \sigma_{\text{“E3”} < ENO \geq \text{“E6”}}(EMP)$$
$$EMP_3 = \sigma_{ENO > \text{“E6”}}(EMP)$$

Note that this fragmentation of the EMP relation is different from the one discussed in Example 5.12.

The localization program for an horizontally fragmented relation is the union of the fragments. In our example we have

$$EMP = EMP_1 \cup EMP_2 \cup EMP_3$$

Thus the generic form of any query specified on EMP is obtained by replacing it by $(EMP_1 \cup EMP_2 \cup EMP_3)$.

The reduction of queries on horizontally fragmented relations consists primarily of determining, after restructuring the subtrees, those that will produce empty relations, and removing them. Horizontal fragmentation can be exploited to simplify both selection and join operations.

Reduction with Selection

Selections on fragments that have a qualification contradicting the qualification of the fragmentation rule generate empty relations. Given a relation R that has been horizontally fragmented as $R_1, R_2, \ldots, R_w$, where $R_j = \sigma_{Pj}(R)$, the rule can be stated formally as follows:

Rule 1:

$$\sigma_{pi}(R_j) = \phi \text{ if } \forall x \text{ in } R : \neg(p_i(x) \wedge p_j(x))$$

where p_i and p_j are selection predicates, x denotes a tuple, and $p(x)$ denotes "predicate p holds for x."

For example, the selection predicate ENO="E1" conflicts with the predicates of fragments EMP_2 and EMP_3 of Example 8.8 (i.e., no tuple in EMP_2 and EMP_3 can satisfy this predicate). Determining the contradicting predicates requires theorem-proving techniques if the predicates are quite general [Hunt and Rosenkrantz, 1979]. However, DBMSs generally simplify predicate comparison by supporting only simple predicates for defining fragmentation rules (by the database administrator).

Example 8.9

We now illustrate reduction by horizontal fragmentation using the following example query:

```
SELECT  *
FROM    EMP
WHERE   ENO  =  "E5"
```

Applying the naive approach to localize EMP from EMP_1, EMP_2, and EMP_3 gives the generic query of Figure 8.6a. By commuting the selection with the union operation, it is easy to detect that the selection predicate contradicts the predicates of EMP_1 and EMP_3, thereby producing empty relations. The reduced query is simply applied to EMP_2 as shown in Figure 8.6b.

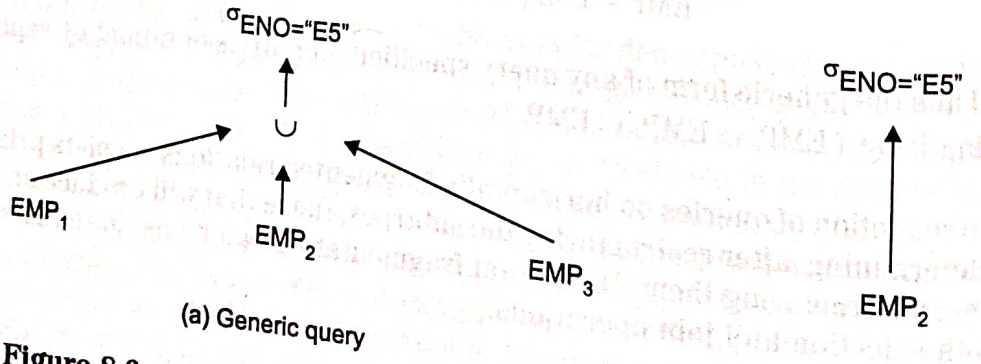


Figure 8.6. Reduction for Horizontal Fragmentation (with Selection)

Reduction with Join

Joins on horizontally fragmented relations can be simplified when the joined relations are fragmented according to the join attribute. The simplification consists of distributing joins over unions and eliminating useless joins. The distribution of join over union can be stated as

$$(R_1 \cup R_2) \bowtie S = (R_1 \bowtie S) \cup (R_2 \bowtie S)$$

where R_i are fragments of R and S is a relation.

With this transformation, unions can be moved up in the operator tree so that all possible joins of fragments are exhibited. Useless joins of fragments can be determined when the qualifications of the joined fragments are contradicting. Assuming that fragments R_i and R_j are defined, respectively, according to predicates p_i and p_j on the same attribute, the simplification rule can be stated as follows:

Rule 2:

$$R_i \bowtie R_j = \phi \text{ if } \forall x \text{ in } R_i, \forall_y \text{ in } R_j : \neg(p_i(x) \wedge p_i(y))$$

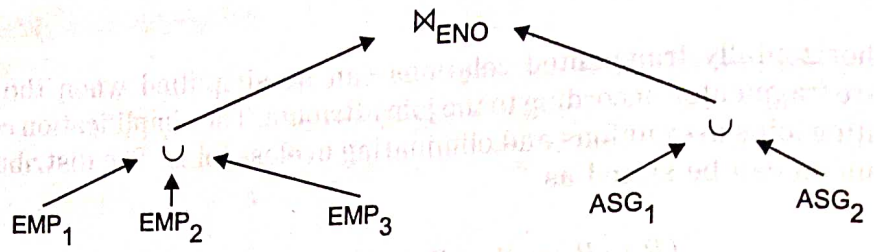
The determination of useless joins can thus be performed by looking only at the fragment predicates. The application of this rule permits the join of two relations to be implemented as parallel partial joins of fragments [Ceri et al., 1986]. It is not always the case that the reduced query is better (i.e., simpler) than the generic query. The generic query is better when there are a large number of partial joins in the reduced query. This case arises when there are few contradicting fragmentation predicates. The worst case occurs when each fragment of one relation must be joined with each fragment of the other relation. This is tantamount to the Cartesian product of the two sets of fragments, with each set corresponding to one relation. The reduced query is better when the number of partial joins is small. For example, if both relations are fragmented using the same predicates, the number of partial joins is equal to the number of fragments of each relation. One advantage of the reduced query is that the partial joins can be done in parallel, and thus increase response time.

Example 8.10

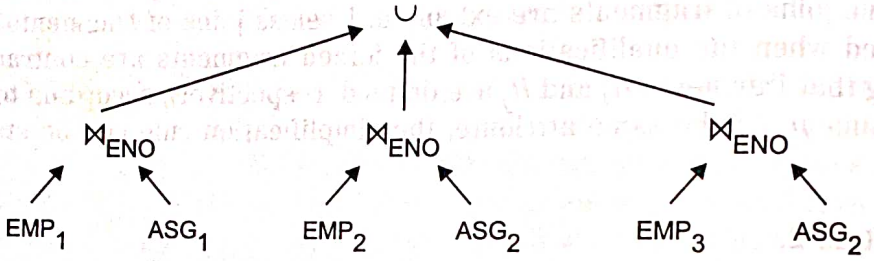
Assume that relation EMP is fragmented between EMP_1, EMP_2, and EMP_3, as above, and that relation ASG is fragmented as

$$ASG_1 = \sigma_{ENO \leq \text{"E3"}}(ASG)$$
$$ASG_2 = \sigma_{ENO > \text{"E3"}}(ASG)$$

EMP_1 and ASG_1 are defined by the same predicate. Furthermore, the predicate defining ASG_2 is the union of the predicates defining EMP_2 and EMP_3. Now consider the join query



(a) Generic query



(b) Reduced query

Figure 8.7. Reduction by Horizontal Fragmentation (with Join)

```
SELECT   *
FROM     EMP, ASG
WHERE    EMP.ENO = ASG.ENO
```

The equivalent generic query is given in Figure 8.7a. The query reduced by distributing joins over unions and applying rule 2 can be implemented as a union of three partial joins that can be done in parallel (Figure 8.7b).

8.2.2 Reduction for Vertical Fragmentation

The vertical fragmentation function distributes a relation based on projection attributes. Since the reconstruction operator for vertical fragmentation is the join, the localization program for a vertically fragmented relation consists of the join of the fragments on the common attribute. For vertical fragmentation, we use the following example.

Example 8.11

Relation EMP can be divided into two vertical fragments where the key attribute ENO is duplicated:

$$EMP_1 = \Pi_{ENO, ENAME}(EMP)$$
$$EMP_2 = \Pi_{ENO, TITLE}(EMP)$$

The localization program is

$$\text{EMP} = \text{EMP}_1 \mid_{\text{ENO}} \text{EMP}_2$$

Similar to horizontal fragmentation, queries on vertical fragments can be reduced by determining the useless intermediate relations and removing the subtrees that produce them. Projections on a vertical fragment that has no attributes in common with the projection attributes (except the key of the relation) produce useless, though not empty relations. Given a relation R, denned over attributes $A = \{A_1, \ldots, A_n\}$, which is vertically fragmented as $R_i = \Pi_{A'}(R)$, where $A' \subseteq A$, the rule can be formally stated as follows:

Rule 3: $\Pi_{D,K}(R_i)$ is useless if the set of projection attributes D is not in A'.

Example 8.12

Let us illustrate the application of this rule using the following example query in SQL:

```
SELECT   ENAME
FROM     EMP
```

The equivalent generic query on EMP_1 and EMP_2 (as obtained in Example 8.10) is given in Figure 8.8a. By commuting the projection with the join (i.e., projecting on ENO, ENAME), we can see that the projection on EMP_2 is useless because ENAME is not in EMP_2. Therefore, the projection needs to apply only to EMP_1, as shown in Figure 8.8b.

8.2.3 Reduction for Derived Fragmentation

As we saw in previous sections, the join operation, which is probably the most important operation because it is both frequent and expensive, can be optimized using primary horizontal fragmentation when the joined relations are fragmented

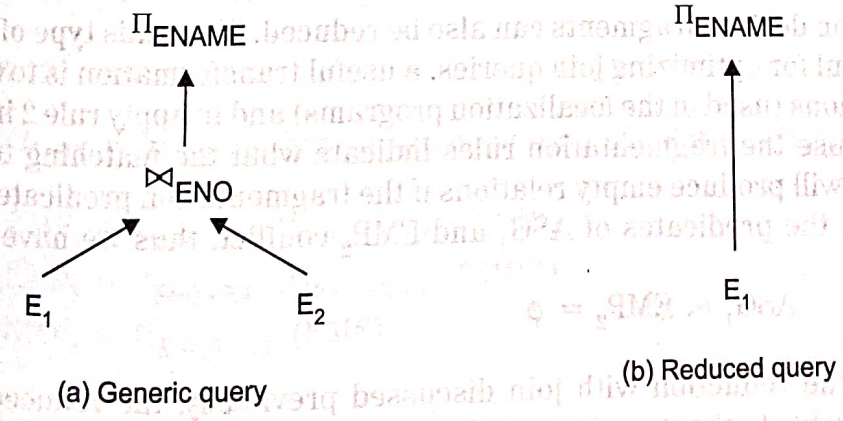


Figure 8.8. Reduction for Vertical Fragmentation

according to the join attributes. In this case the join of two relations is implemented as a union of partial joins. However, this method precludes one of the relations from being fragmented on a different attribute used for selection. Derived horizontal fragmentation is another way of distributing two relations so that the joint processing of select and join is improved. Typically, if relation R is subject to derived horizontal fragmentation due to relation S, the fragments of R and S that have the same join attribute values are located at the same site. In addition, S can be fragmented according to a selection predicate.

Since tuples of R are placed according to the tuples of S, derived fragmentation should be used only for one-to-many (hierarchical) relationships of the form $S \rightarrow R$, where a tuple of S can match with n tuples of R, but a tuple of R matches with exactly one tuple of S. Note that derived fragmentation could be used for many-to-many relationships provided that tuples of S (that match with n tuples of R) are replicated. Such replication is difficult to maintain consistently. For simplicity, we assume and advise that derived fragmentation be used only for hierarchical relationships.

Example 8.13

Given a one-to-many relationship from EMP to ASG, relation ASG(ENO, PNO, RESP, DUR) can be indirectly fragmented according to the following rules:

$$\text{ASG}_1 = \text{ASG} \ltimes_{\text{ENO}} \text{EMP}_1$$
$$\text{ASG}_2 = \text{ASG} \ltimes_{\text{ENO}} \text{EMP}_2$$

Recall from Chapter 5 that the predicate on

$$\text{EMP}_1 = \sigma_{\text{TITLE}=\text{"Programmer"}} (\text{EMP})$$
$$\text{EMP}_2 = \sigma_{\text{TITLE}\neq\text{"Programmer"}} (\text{EMP})$$

The localization program for a horizontally fragmented relation is the union of the fragments. In our example, we have

$$\text{ASG} = \text{ASG}_1 \cup \text{ASG}_2$$

Queries on derived fragments can also be reduced. Since this type of fragmentation is useful for optimizing join queries, a useful transformation is to distribute joins over unions (used in the localization programs) and to apply rule 2 introduced earlier. Because the fragmentation rules indicate what the matching tuples are, certain joins will produce empty relations if the fragmentation predicates conflict. For example, the predicates of ASG_1 and EMP_2 conflict; thus we have

$$\text{ASG}_1 \bowtie \text{EMP}_2 = \phi$$

Contrary to the reduction with join discussed previously, the reduced query is always preferable to the generic query because the number of partial joins usually equals the number of fragments of R.

Example 8.14

The reduction by derived fragmentation is illustrated by applying it to the following SQL query, which retrieves all attributes of tuples from EMP and ASG that have the same value of ENO and the title "Mech. Eng."

```
SELECT   *
FROM     EMP, ASG
WHERE    ASG.ENO = EMP.ENO
AND      TITLE = "Mech. Eng."
```

The generic query on fragments EMP_1, EMP_2, ASG_1, and ASG_2, defined previously is given in Figure 8.9a. By pushing selection down to fragments EMP_1 and EMP_2, the query reduces to that of Figure 8.9b. This is because the selection predicate conflicts with that of EMP_1, and thus EMP_1 can be removed. In order to discover conflicting join predicates, we distribute joins over unions. This produces the tree of Figure 8.9c. The left subtree joins two fragments, ASG_1 and EMP_2, whose qualifications conflict because of predicates TITLE = "Programmer" in ASG_1, and TITLE $\neq$ "Programmer" in EMP_2. Therefore the left subtree which produces an empty relation can be removed, and the reduced query of Figure 8.9d is obtained. This example illustrates the value of fragmentation in improving the execution performance of distributed queries.

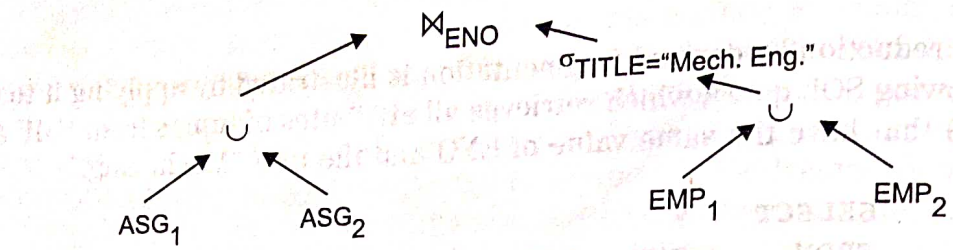
8.2.4 Reduction for Hybrid Fragmentation

Hybrid fragmentation is obtained by combining the fragmentation functions discussed above. The goal of hybrid fragmentation is to support, efficiently, queries involving projection, selection, and join. Note that the optimization of an operation or of a combination of operations is always done at the expense of other operations. For example, hybrid fragmentation based on selection–projection will make selection only, or projection only, less efficient than with horizontal fragmentation (or vertical fragmentation). The localization program for a hybrid fragmented relation uses unions and joins of fragments.

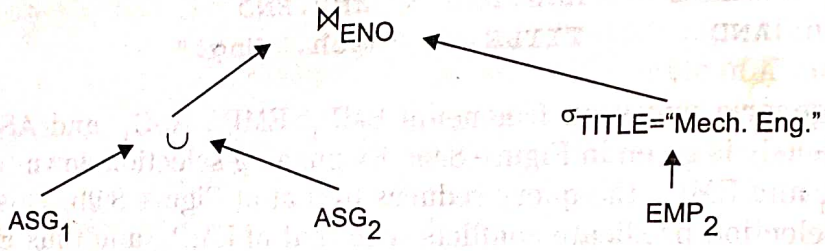
Example 8.15

Here is an example of hybrid fragmentation of relation EMP:

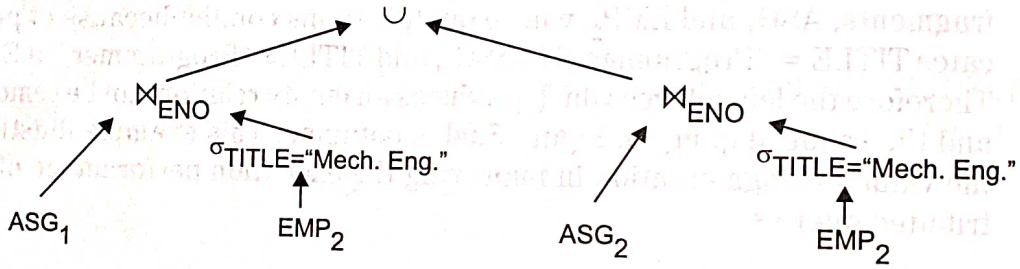
$$EMP_1 = \sigma_{ENO \geq \text{"E4"}} (\Pi_{ENO,ENAME} (EMP))$$
$$EMP_2 = \sigma_{ENO > \text{"E4"}} (\Pi_{ENO,ENAME} (EMP))$$
$$EMP_3 = \Pi_{ENO,TITLE} (EMP)$$



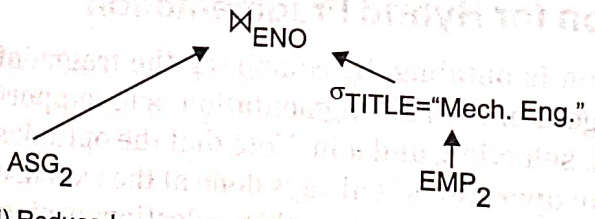
(a) Generic query



(b) Query after pushing selection down



(c) Query after moving unions up



(d) Reduced query after eliminating the left subtree

Figure 8.9. Reduction for Indirect Fragmentation

In our example, the localization program is

$$EMP = (EMP_1 \cup EMP_2) \mid_{ENO} EMP_3$$

Queries on hybrid fragments can be reduced by combining the rules used, respectively, in primary horizontal, vertical, and derived horizontal fragmentation. These rules can be summarized as follows:

1. Remove empty relations generated by contradicting selections on horizontal fragments.
2. Remove useless relations generated by projections on vertical fragments.
3. Distribute joins over unions in order to isolate and remove useless joins.

REVIEW QUESTIONS

8.1 What do you mean by normalization? Explain with examples.
8.2 Explain query analysis with examples.
8.3 What is elimination of redundancy?
8.4 Give an example of operation tree.
8.5 How do you generate equivalent operator tree?
8.6 What do you mean by rewritten operator tree?
8.7 Give an example of reduction for primary horizontal fragmentation.
8.8 Give an example of reduction for vertical fragmentation.
8.9 Give an example of reduction for derived fragmentation.
8.10 Explain reduction for hybrid fragmentation.