Example 5.9. Consider the view SYSAN in Example 5.1. Let us write the view definition as SYSAN=q(EMP) meaning that the view is defined by a query q on EMP. We can compute the differential views using only the differential relations, i.e., $\text{SYSAN}^+ = q(\text{EMP}^+)$ and $\text{SYSAN}^- = q(\text{EMP}^-)$. Thus, the view SYSAN is self-maintainable. ♦

Self-maintainability depends on the views' expressiveness and can be defined with respect to the kind of updates (insertion, deletion or modification) [Gupta et al., 1996]. Most SPJ views are not self-maintainable with respect to insertion but are often self-maintainable with respect to deletion and modification. For instance, an SPJ view is self-maintainable with respect to deletion of relation R if the key attributes of R are included in the view.

Example 5.10. Consider the view EG of Example 5.5. Let us add attribute ENO (which is key of EMP) in the view definition. This view is not self-maintainable with respect to insertion. For instance, after an insertion of an ASG tuple, we need to perform the join with EMP to get the corresponding ENAME to insert in the view. However, this view is self-maintainable with respect to deletion on EMP. For instance, if one EMP tuple is deleted, the view tuples having same ENO can be deleted. ♦

5.2 Data Security

Data security is an important function of a database system that protects data against unauthorized access. Data security includes two aspects: *data protection* and *access control*.

Data protection is required to prevent unauthorized users from understanding the physical content of data. This function is typically provided by file systems in the context of centralized and distributed operating systems. The main data protection approach is data encryption [Fernandez et al., 1981], which is useful both for information stored on disk and for information exchanged on a network. Encrypted (encoded) data can be decrypted (decoded) only by authorized users who "know" the code. The two main schemes are the Data Encryption Standard [NBS, 1977] and the public-key encryption schemes ([Diffie and Hellman, 1976] and [Rivest et al., 1978]). In this section we concentrate on the second aspect of data security, which is more specific to database systems. A complete presentation of database security techniques can be found in [Castano et al., 1995].

Access control must guarantee that only authorized users perform operations they are allowed to perform on the database. Many different users may have access to a large collection of data under the control of a single centralized or distributed system. The centralized or distributed DBMS must thus be able to restrict the access of a subset of the database to a subset of the users. Access control has long been provided by operating systems, and more recently, by distributed operating systems [Tanenbaum, 1995] as services of the file system. In this context, a centralized control is offered. Indeed, the central controller creates objects, and this person may

allow particular users to perform particular operations (read, write, execute) on these objects. Also, objects are identified by their external names.

Access control in database systems differs in several aspects from that in traditional file systems. Authorizations must be refined so that different users have different rights on the same database objects. This requirement implies the ability to specify subsets of objects more precisely than by name and to distinguish between groups of users. In addition, the decentralized control of authorizations is of particular importance in a distributed context. In relational systems, authorizations can be uniformly controlled by database administrators using high-level constructs. For example, controlled objects can be specified by predicates in the same way as is a query qualification.

There are two main approaches to database access control [Lunt and Fernández, 1990]. The first approach is called *discretionary* and has long been provided by DBMS. Discretionary access control (or *authorization control*) defines access rights based on the users, the type of access (e.g., SELECT, UPDATE) and the objects to be accessed. The second approach, called *mandatory* or *multilevel* [Lunt and Fernández, 1990; Jajodia and Sandhu, 1991] further increases security by restricting access to classified data to cleared users. Support of multilevel access control by major DBMSs is more recent and stems from increased security threats coming from the Internet.

From solutions to access control in centralized systems, we derive those for distributed DBMSs. However, there is the additional complexity which stems from the fact that objects and users can be distributed. In what follows we first present discretionary and multilevel access control in centralized systems and then the additional problems and their solutions in distributed systems.

5.2.1 Discretionary Access Control

Three main actors are involved in discretionary access control control: the *subject* (e.g., users, groups of users) who trigger the execution of application programs; the *operations*, which are embedded in application programs; and the *database objects*, on which the operations are performed [Hoffman, 1977]. Authorization control consists of checking whether a given triple (subject, operation, object) can be allowed to proceed (i.e., the user can execute the operation on the object). An authorization can be viewed as a triple (subject, operation type, object definition) which specifies that the subjects has the right to perform an operation of operation type on an object. To control authorizations properly, the DBMS requires the definition of subjects, objects, and access rights.

The introduction of a subject in the system is typically done by a pair (user name, password). The user name uniquely *identifies* the users of that name in the system, while the password, known only to the users of that name, *authenticates* the users. Both user name and password must be supplied in order to log in the system. This prevents people who do not know the password from entering the system with only the user name.

The objects to protect are subsets of the database. Relational systems provide finer and more general protection granularity than do earlier systems. In a file system, the protection granule is the file, while in an object-oriented DBMS, it is the object type. In a relational system, objects can be defined by their type (view, relation, tuple, attribute) as well as by their content using selection predicates. Furthermore, the view mechanism as introduced in Section 5.1 permits the protection of objects simply by hiding subsets of relations (attributes or tuples) from unauthorized users.

A right expresses a relationship between a subject and an object for a particular set of operations. In an SQL-based relational DBMS, an operation is a high-level statement such as SELECT, INSERT, UPDATE, or DELETE, and rights are defined (granted or revoked) using the following statements:

GRANT ⟨operation type(s)⟩ ON ⟨object⟩ TO ⟨subject(s)⟩
REVOKE ⟨operation type(s)⟩ FROM ⟨object⟩ TO ⟨subject(s)⟩

The keyword *public* can be used to mean all users. Authorization control can be characterized based on who (the grantors) can grant the rights. In its simplest form, the control is centralized: a single user or user class, the database administrators, has all privileges on the database objects and is the only one allowed to use the GRANT and REVOKE statements.

A more flexible but complex form of control is decentralized [Griffiths and Wade, 1976]: the creator of an object becomes its owner and is granted all privileges on it. In particular, there is the additional operation type GRANT, which transfers all the rights of the grantor performing the statement to the specified subjects. Therefore, the person receiving the right (the grantee) may subsequently grant privileges on that object. The main difficulty with this approach is that the revoking process must be recursive. For example, if A, who granted B who granted C the GRANT privilege on object O, wants to revoke all the privileges of B on O, all the privileges of C on O must also be revoked. To perform revocation, the system must maintain a hierarchy of grants per object where the creator of the object is the root.

The privileges of the subjects over objects are recorded in the catalog (directory) as authorization rules. There are several ways to store the authorizations. The most convenient approach is to consider all the privileges as an *authorization matrix*, in which a row defines a subject, a column an object, and a matrix entry (for a pair ⟨subject, object⟩), the authorized operations. The authorized operations are specified by their operation type (e.g., SELECT, UPDATE). It is also customary to associate with the operation type a predicate that further restricts the access to the object. The latter option is provided when the objects must be base relations and cannot be views. For example, one authorized operation for the pair ⟨Jones, relation EMP⟩ could be

```
SELECT WHERE TITLE = "Syst.Anal."
```

which authorizes Jones to access only the employee tuples for system analysts. Figure 5.5 gives an example of an authorization matrix where objects are either relations (EMP and ASG) or attributes (ENAME).

	EMP	ENAME	ASG
Casey	UPDATE	UPDATE	UPDATE
Jones	SELECT	SELECT	SELECT WHERE RESP ≠ "Manager"
Smith	NONE	SELECT	NONE

Fig. 5.5 Example of Authorization Matrix

The authorization matrix can be stored in three ways: by row, by column, or by element. When the matrix is stored by *row*, each subject is associated with the list of objects that may be accessed together with the related access rights. This approach makes the enforcement of authorizations efficient, since all the rights of the logged-on user are together (in the user profile). However, the manipulation of access rights per object (e.g., making an object public) is not efficient since all subject profiles must be accessed. When the matrix is stored by *column*, each object is associated with the list of subjects who may access it with the corresponding access rights. The advantages and disadvantages of this approach are the reverse of the previous approach.

The respective advantages of the two approaches can be combined in the third approach, in which the matrix is stored by *element*, that is, by relation (subject, object, right). This relation can have indices on both subject and object, thereby providing fast-access right manipulation per subject and per object.

5.2.2 Multilevel Access Control

Discretionary access control has some limitations. One problem is that a malicious user can access unauthorized data through an authorized user. For instance, consider user A who has authorized access to relations R and S and user B who has authorized access to relation S only. If B somehow manages to modify an application program used by A so it writes R data into S, then B can read unauthorized data without violating authorization rules.

Multilevel access control answers this problem and further improves security by defining different security levels for both subjects and data objects. Multilevel access control in databases is based on the well-known Bell and Lapaduda model designed for operating system security [Bell and Lapuda, 1976]. In this model, subjects are processes acting on a user's behalf; a process has a security level also called *clearance* derived from that of the user. In its simplest form, the security levels are Top Secret (TS), Secret (S), Confidential (C) and Unclassified (U), and ordered as $TS > S > C > U$, where ">" means "more secure". Access in read and write modes by subjects is restricted by two simple rules:

1. A subject S is allowed to read an object of security level l only if $level(S) \geq l$.

2. A subject S is allowed to write an object of security level l only if $class(S) \leq l$.

Rule 1 (called "no read up") protects data from unauthorized disclosure, i.e., a subject at a given security level can only read objects at the same or lower security levels. For instance, a subject with secret clearance cannot read top-secret data. Rule 2 (called "no write down") protects data from unauthorized change, i.e., a subject at a given security level can only write objects at the same or higher security levels. For instance, a subject with top-secret clearance can only write top-secret data but cannot write secret data (which could then contain top-secret data).

In the relational model, data objects can be relations, tuples or attributes. Thus, a relation can be classified at different levels: relation (i.e., all tuples in the relation have the same security level), tuple (i.e., every tuple has a security level), or attribute (i.e., every distinct attribute value has a security level). A classified relation is thus called *multilevel relation* to reflect that it will appear differently (with different data) to subjects with different clearances. For instance, a multilevel relation classified at the tuple level can be represented by adding a security level attribute to each tuple. Similarly, a multilevel relation classified at attribute level can be represented by adding a corresponding security level to each attribute. Figure 5.6 illustrates a multilevel relation PROJ* based on relation PROJ which is classified at the attribute level. Note that the additional security level attributes may increase significantly the size of the relation.

PROJ*

PNO	SL1	PNAME	SL2	BUDGET	SL3	LOC	SL4
P1	C	Instrumentation	C	150000	C	Montreal	C
P2	C	Database Develop.	C	135000	S	New York	S
P3	S	CAD/CAM	S	250000	S	New York	S

Fig. 5.6 Multilevel relation PROJ* classified at the attribute level

The entire relation also has a security level which is the lowest security level of any data it contains. For instance, relation PROJ* has security level C. A relation can then be accessed by any subject having a security level which is the same or higher. However, a subject can only access data for which it has clearance. Thus, attributes for which a subject has no clearance will appear to the subject as null values with an associated security level which is the same as the subject. Figure 5.7 shows an instance of relation PROJ* as accessed by a subject at a confidential security level.

Multilevel access control has strong impact on the data model because users do not see the same data and have to deal with unexpected side-effects. One major side-effect is called *polyinstantiation* [Lunt et al., 1990] which allows the same object to have different attribute values depending on the users' security level. Figure 5.8 illustrates a multirelation with polyinstantiated tuples. Tuple of primary key P3 has two instantiations, each one with a different security level. This may result from a subject S with security level C inserting a tuple with key="P3" in relation PROJ* in

PROJ*C

PNO	SL1	PNAME	SL2	BUDGET	SL3	LOC	SL4
P1	C	Instrumentation	C	150000	C	Montreal	C
P2	C	Database Develop.	C	Null	C	Null	C

Fig. 5.7 Confidential relation PROJ*C

Figure 5.6. Because *S* (with confidential clearance level) should ignore the existence of tuple with key="P3" (classified as secret), the only practical solution is to add a second tuple with same key and different classification. However, a user with secret clearance would see both tuples with key="E3" and should interpret this unexpected effect.

PROJ**

PNO	SL1	PNAME	SL2	BUDGET	SL3	LOC	SL4
P1	C	Instrumentation	C	150000	C	Montreal	C
P2	C	Database Develop.	C	135000	S	New York	S
P3	S	CAD/CAM	S	250000	S	New York	S
P3	C	Web Develop.	C	200000	C	Paris	C

Fig. 5.8 Multilevel relation with polyinstantiation

5.2.3 Distributed Access Control

The additional problems of access control in a distributed environment stem from the fact that objects and subjects are distributed and that messages with sensitive data can be read by unauthorized users. These problems are: remote user authentication, management of discretionary access rules, handling of views and of user groups, and enforcing multilevel access control.

Remote user authentication is necessary since any site of a distributed DBMS may accept programs initiated, and authorized, at remote sites. To prevent remote access by unauthorized users or applications (e.g., from a site that is not part of the distributed DBMS), users must also be identified and authenticated at the accessed site. Furthermore, instead of using passwords that could be obtained from sniffing messages, encrypted certificates could be used.

Three solutions are possible for managing authentication:

1. Authentication information is maintained at a central site for *global users* which can then be authenticated only once and then accessed from multiple sites.

2. The information for authenticating users (user name and password) is replicated at all sites in the catalog. Local programs, initiated at a remote site, must also indicate the user name and password.
3. All sites of the distributed DBMS identify and authenticate themselves similar to the way users do. Intersite communication is thus protected by the use of the site password. Once the initiating site has been authenticated, there is no need for authenticating their remote users.

The first solution simplifies password administration significantly and enables single authentication (also called single sign on). However, the central authentication site can be a single point of failure and a bottleneck. The second solution is more costly in terms of directory management given that the introduction of a new user is a distributed operation. However, users can access the distributed database from any site. The third solution is necessary if user information is not replicated. Nevertheless, it can also be used if there is replication of the user information. In this case it makes remote authentication more efficient. If user names and passwords are not replicated, they should be stored at the sites where the users access the system (i.e., the home site). The latter solution is based on the realistic assumption that users are more static, or at least they always access the distributed database from the same site.

Distributed authorization rules are expressed in the same way as centralized ones. Like view definitions, they must be stored in the catalog. They can be either fully replicated at each site or stored at the sites of the referenced objects. In the latter case the rules are duplicated only at the sites where the referenced objects are distributed. The main advantage of the fully replicated approach is that authorization can be processed by query modification [Stonebraker, 1975] at compile time. However, directory management is more costly because of data duplication. The second solution is better if locality of reference is very high. However, distributed authorization cannot be controlled at compile time.

Views may be considered to be objects by the authorization mechanism. Views are composite objects, that is, composed of other underlying objects. Therefore, granting access to a view translates into granting access to underlying objects. If view definition and authorization rules for all objects are fully replicated (as in many systems), this translation is rather simple and can be done locally. The translation is harder when the view definition and its underlying objects are all stored separately [Wilms and Lindsay, 1981], as is the case with site autonomy assumption. In this situation, the translation is a totally distributed operation. The authorizations granted on views depend on the access rights of the view creator on the underlying objects. A solution is to record the association information at the site of each underlying object.

Handling user groups for the purpose of authorization simplifies distributed database administration. In a centralized DBMS, "all users" can be referred to as *public*. In a distributed DBMS, the same notion is useful, the public denoting all the users of the system. However an intermediate level is often introduced to specify the public at a particular site, denoted by public@site_s [Wilms and Lindsay, 1981]. The public is a particular user group. More precise groups can be defined by the command

```
DEFINE GROUP ⟨group_id⟩ AS ⟨list of subject ids⟩
```

The management of groups in a distributed environment poses some problems since the subjects of a group can be located at various sites and access to an object may be granted to several groups, which are themselves distributed. If group information as well as access rules are fully replicated at all sites, the enforcement of access rights is similar to that of a centralized system. However, maintaining this replication may be expensive. The problem is more difficult if site autonomy (with decentralized control) must be maintained. Several solutions to this problem have been identified [Wilms and Lindsay, 1981]. One solution enforces access rights by performing a remote query to the nodes holding the group definition. Another solution replicates a group definition at each node containing an object that may be accessed by subjects of that group. These solutions tend to decrease the degree of site autonomy.

Enforcing multilevel access control in a distributed environment is made difficult by the possibility of indirect means, called *covert channels*, to access unauthorized data [Rjaibi, 2004]. For instance, consider a simple distributed DBMS architecture with two sites, each managing its database at a single security level, e.g., one site is confidential while the other is secret. According to the "no write down" rule, an update operation from a subject with secret clearance could only be sent to the secret site. However, according to the "no read up" rule, a read query from the same secret subject could be sent to both the secret and the confidential sites. Since the query sent to the confidential site may contain secret information (e.g., in a select predicate), it is potentially a covert channel. To avoid such covert channels, a solution is to replicate part of the database [Thuraisingham, 2001] so that a site at security level l contains all data that a subject at level l can access. For instance, the secret site would replicate confidential data so that it can entirely process secret queries. One problem with this architecture is the overhead of maintaining the consistency of replicas (see Chapter 13 on replication). Furthermore, although there are no covert channels for queries, there may still be covert channels for update operations because the delays involved in synchronizing transactions may be exploited [Jajodia et al., 2001]. The complete support for multilevel access control in distributed database systems, therefore, requires significant extensions to transaction management techniques [Ray et al., 2000] and to distributed query processing techniques [Agrawal et al., 2003].

5.3 Semantic Integrity Control

Another important and difficult problem for a database system is how to guarantee *database consistency*. A database state is said to be consistent if the database satisfies a set of constraints, called *semantic integrity constraints*. Maintaining a consistent database requires various mechanisms such as concurrency control, reliability, protection, and semantic integrity control, which are provided as part of transaction management. Semantic integrity control ensures database consistency by rejecting update transactions that lead to inconsistent database states, or by activat-

ing specific actions on the database state, which compensate for the effects of the update transactions. Note that the updated database must satisfy the set of integrity constraints.

In general, semantic integrity constraints are rules that represent the *knowledge* about the properties of an application. They define static or dynamic application properties that cannot be directly captured by the object and operation concepts of a data model. Thus the concept of an integrity rule is strongly connected with that of a data model in the sense that more semantic information about the application can be captured by means of these rules.

Two main types of integrity constraints can be distinguished: structural constraints and behavioral constraints. *Structural constraints* express basic semantic properties inherent to a model. Examples of such constraints are unique key constraints in the relational model, or one-to-many associations between objects in the object-oriented model. *Behavioral constraints*, on the other hand, regulate the application behavior. Thus they are essential in the database design process. They can express associations between objects, such as inclusion dependency in the relational model, or describe object properties and structures. The increasing variety of database applications and the development of database design aid tools call for powerful integrity constraints that can enrich the data model.

Integrity control appeared with data processing and evolved from procedural methods (in which the controls were embedded in application programs) to declarative methods. Declarative methods have emerged with the relational model to alleviate the problems of program/data dependency, code redundancy, and poor performance of the procedural methods. The idea is to express integrity constraints using assertions of predicate calculus [Florentin, 1974]. Thus a set of semantic integrity assertions defines database consistency. This approach allows one to easily declare and modify complex integrity constraints.

The main problem in supporting automatic semantic integrity control is that the cost of checking for constraint violation can be prohibitive. Enforcing integrity constraints is costly because it generally requires access to a large amount of data that are not directly involved in the database updates. The problem is more difficult when constraints are defined over a distributed database.

Various solutions have been investigated to design an integrity manager by combining optimization strategies. Their purpose is to (1) limit the number of constraints that need to be enforced, (2) decrease the number of data accesses to enforce a given constraint in the presence of an update transaction, (3) define a preventive strategy that detects inconsistencies in a way that avoids undoing updates, (4) perform as much integrity control as possible at compile time. A few of these solutions have been implemented, but they suffer from a lack of generality. Either they are restricted to a small set of assertions (more general constraints would have a prohibitive checking cost) or they only support restricted programs (e.g., single-tuple updates).

In this section we present the solutions for semantic integrity control first in centralized systems and then in distributed systems. Since our context is the relational model, we consider only declarative methods.

5.3.1 Centralized Semantic Integrity Control

A semantic integrity manager has two main components: a language for expressing and manipulating integrity assertions, and an enforcement mechanism that performs specific actions to enforce database integrity upon update transactions.

5.3.1.1 Specification of Integrity Constraints

Integrity constraints should be manipulated by the database administrator using a high-level language. In this section we illustrate a declarative language for specifying integrity constraints [Simon and Valduriez, 1987]. This language is much in the spirit of the standard SQL language, but with more generality. It allows one to specify, read, or drop integrity constraints. These constraints can be defined either at relation creation time, or at any time, even if the relation already contains tuples. In both cases, however, the syntax is almost the same. For simplicity and without lack of generality, we assume that the effect of integrity constraint violation is to abort the violating transactions. However, the SQL standard provides means to express the propagation of update actions to correct inconsistencies, with the CASCADING clause within the constraint declaration. More generally, *triggers* (event-condition-action rules) [Ramakrishnan and Gehrke, 2003] can be used to automatically propagate updates, and thus to maintain semantic integrity. However, triggers are quite powerful and thus more difficult to support efficiently than specific integrity constraints.

In relational database systems, integrity constraints are defined as assertions. An assertion is a particular expression of tuple relational calculus (see Chapter 2), in which each variable is either universally ($\forall$) or existentially ($\exists$) quantified. Thus an assertion can be seen as a query qualification that is either true or false for each tuple in the Cartesian product of the relations determined by the tuple variables. We can distinguish between three types of integrity constraints: predefined, precondition, or general constraints.

Examples of integrity constraints will be given on the following database:

EMP(ENO, ENAME, TITLE)

PROJ(PNO, PNAME, BUDGET)

ASG(ENO, PNO, RESP, DUR)

Predefined constraints are based on simple keywords. Through them, it is possible to express concisely the more common constraints of the relational model, such as non-null attribute, unique key, foreign key, or functional dependency [Fagin and Vardi, 1984]. Examples 5.11 through 5.14 demonstrate predefined constraints.

Example 5.11. Employee number in relation EMP cannot be null.

```
ENO NOT NULL IN EMP
```

♦

Example 5.12. The pair (ENO, PNO) is the unique key in relation ASG.

```
(ENO, PNO) UNIQUE IN ASG
```

♦

Example 5.13. The project number PNO in relation ASG is a foreign key matching the primary key PNO of relation PROJ. In other words, a project referred to in relation ASG must exist in relation PROJ.

```
PNO IN ASG REFERENCES PNO IN PROJ
```

♦

Example 5.14. The employee number functionally determines the employee name.

```
ENO IN EMP DETERMINES ENAME
```

♦

Precondition constraints express conditions that must be satisfied by all tuples in a relation for a given update type. The update type, which might be INSERT, DELETE, or MODIFY, permits restricting the integrity control. To identify in the constraint definition the tuples that are subject to update, two variables, NEW and OLD, are implicitly defined. They range over new tuples (to be inserted) and old tuples (to be deleted), respectively [Astrahan et al., 1976]. Precondition constraints can be expressed with the SQL CHECK statement enriched with the ability to specify the update type. The syntax of the CHECK statement is

```
CHECK ON ⟨ relation name ⟩ WHEN⟨ update type ⟩
         (⟨ qualification over relation name⟩)
```

Examples of precondition constraints are the following:

Example 5.15. The budget of a project is between 500K and 1000K.

```
CHECK ON PROJ (BUDGET+ >= 500000 AND  BUDGET <= 1000000)
```

♦

Example 5.16. Only the tuples whose budget is 0 may be deleted.

```
CHECK ON PROJ WHEN DELETE (BUDGET = 0)
```

♦

Example 5.17. The budget of a project can only increase.

```
CHECK ON PROJ (NEW.BUDGET > OLD.BUDGET
AND NEW.PNO = OLD.PNO)
```

♦

General constraints are formulas of tuple relational calculus where all variables are quantified. The database system must ensure that those formulas are always true. General constraints are more concise than precompiled constraints since the former may involve more than one relation. For instance, at least three precompiled constraints are necessary to express a general constraint on three relations. A general constraint may be expressed with the following syntax:

```
CHECK ON list of <variable name>:<relation name>,
(<qualification>)
```

Examples of general constraints are given below.

Example 5.18. The constraint of Example 5.8 may also be expressed as

```
CHECK ON e1:EMP, e2:EMP
     (e1.ENAME = e2.ENAME IF e1.ENO = e2.ENO)
```

♦

Example 5.19. The total duration for all employees in the CAD project is less than 100.

```
CHECK ON g:ASG, j:PROJ (SUM(g.DUR WHERE
     g.PNO=j.PNO)<100 IF j.PNAME="CAD/CAM")
```

♦

5.3.1.2 Integrity Enforcement

We now focus on enforcing semantic integrity that consists of rejecting update transactions that violate some integrity constraints. A constraint is violated when it becomes false in the new database state produced by the update transaction. A major difficulty in designing an integrity manager is finding efficient enforcement algorithms. Two basic methods permit the rejection of inconsistent update transactions. The first one is based on the *detection* of inconsistencies. The update transaction u is executed, causing a change of the database state D to D_u. The enforcement algorithm verifies, by applying tests derived from these constraints, that all relevant constraints hold in state D_u. If state D_u is inconsistent, the DBMS can try either to reach another consistent state, D'_u, by modifying D_u with compensation actions, or to restore state D by undoing u. Since these tests are applied *after* having changed the database state, they are generally called *posttests*. This approach may be inefficient if a large amount of work (the update of D) must be undone in the case of an integrity failure.

The second method is based on the *prevention* of inconsistencies. An update is executed only if it changes the database state to a consistent state. The tuples subject to the update transaction are either directly available (in the case of insert) or must be retrieved from the database (in the case of deletion or modification). The enforcement algorithm verifies that all relevant constraints will hold after updating those tuples. This is generally done by applying to those tuples tests that are derived from the integrity constraints. Given that these tests are applied *before* the database state is changed, they are generally called *pretests*. The preventive approach is more efficient than the detection approach since updates never need to be undone because of integrity violation.

The query modification algorithm [Stonebraker, 1975] is an example of a preventive method that is particularly efficient at enforcing domain constraints. It adds the assertion qualification to the query qualification by an AND operator so that the modified query can enforce integrity.

Example 5.20. The query for increasing the budget of the CAD/CAM project by 10%, which would be specified as

```
UPDATE PROJ
SET    BUDGET = BUDGET*1.1
WHERE  PNAME= "CAD/CAM"
```

will be transformed into the following query in order to enforce the domain constraint discussed in Example 5.9.

```
UPDATE PROJ
SET    BUDGET = BUDGET * 1.1
WHERE  PNAME= "CAD/CAM"
AND    NEW.BUDGET ≥ 500000
AND    NEW.BUDGET ≤ 1000000
```

♦

The query modification algorithm, which is well known for its elegance, produces pretests at run time by ANDing the assertion predicates with the update predicates of each instruction of the transaction. However, the algorithm only applies to tuple calculus formulas and can be specified as follows. Consider the assertion $(\forall x \in R)F(x)$, where F is a tuple calculus expression in which x is the only free variable. An update of R can be written as $(\forall x \in R)(Q(x) \Rightarrow update(x))$, where Q is a tuple calculus expression whose only free variable is x. Roughly speaking, the query modification consists in generating the update $(\forall x \in R)((Q(x)$ and $F(x)) \Rightarrow$update$(x))$. Thus x needs to be universally quantified.

Example 5.21. The foreign key constraint of Example 5.13 that can be rewritten as

$$\forall g \in \text{ASG}, \exists j \in \text{PROJ} : g.\text{PNO} = j.\text{PNO}$$

could not be processed by query modification because the variable j is not universally quantified. ♦

To handle more general constraints, pretests can be generated at constraint definition time, and enforced at run time when updates occur [Bernstein et al., 1980a; Bernstein and Blaustein, 1982; Blaustein, 1981; Nicolas, 1982]. The method described by Nicolas [1982] is restricted to updates that insert or delete a *single* tuple of a single relation. The algorithm proposed by Bernstein et al. [1980a] and Blaustein [1981] is an improvement, although updates are single single tuple. The algorithm builds a pretest at constraint definition time for each constraint and each update type (insert, delete). These pretests are enforced at run time. This method accepts multirelation, monovariable assertions, possibly with aggregates. The principle is the substitution of the tuple variables in the assertion by constants from an updated tuple. Despite its important contribution to research, the method is hardly usable in a real environment because of the restriction on updates.

In the rest of this section, we present the method proposed by Simon and Valduriez [1986, 1987], which combines the generality of updates supported by Stonebraker [1975] with at least the generality of assertions for which pretests can be produced by Blaustein [1981]. This method is based on the production, at assertion definition time,

of pretests that are used subsequently to prevent the introduction of inconsistencies in the database. This is a general preventive method that handles the entire set of constraints introduced in the preceding section. It significantly reduces the proportion of the database that must be checked when enforcing assertions in the presence of updates. This is a major advantage when applied to a distributed environment.

The definition of pretest uses differential relations, as defined in Section 5.1.3. A *pretest* is a triple (R, U, C) in which R is a relation, U is an update type, and C is an assertion ranging over the differential relation(s) involved in an update of type U. When an integrity constraint I is defined, a set of pretests may be produced for the relations used by I. Whenever a relation involved in I is updated by a transaction u, the pretests that must be checked to enforce I are only those defined on I for the update type of u. The performance advantage of this approach is twofold. First, the number of assertions to enforce is minimized since only the pretests of type u need be checked. Second, the cost of enforcing a pretest is less than that of enforcing I since differential relations are, in general, much smaller than the base relations.

Pretests may be obtained by applying transformation rules to the original assertion. These rules are based on a syntactic analysis of the assertion and quantifier permutations. They permit the substitution of differential relations for base relations. Since the pretests are simpler than the original ones, the process that generates them is called *simplification*.

Example 5.22. Consider the modified expression of the foreign key constraint in Example 5.15. The pretests associated with this constraint are

(ASG, **INSERT**, C_1), (PROJ, **DELETE**, C_2) and (PROJ, **MODIFY**, C_3)

where C_1 is

$\forall$ **NEW** $\in$ ASG$^+$, $\exists j \in$ PROJ: **NEW**.PNO = j.PNO

C_2 is

$\forall g \in$ ASG, $\forall$ **OLD** $\in$ PROJ$^-$: g.PNO $\neq$ **OLD**.PNO

and C_3 is

$\forall g \in$ ASG, $\forall$**OLD** $\in$ PROJ$^-$, $\exists$ **NEW** $\in$ PROJ$^+$: g.PNO $\neq$ **OLD**.PNO OR **OLD**.PNO = **NEW**.PNO

♦

The advantage provided by such pretests is obvious. For instance, a deletion on relation ASG does not incur any assertion checking.

The enforcement algorithm [Simon and Valduriez, 1984] makes use of pretests and is specialized according to the class of the assertions. Three classes of constraints are distinguished: single-relation constraints, multirelation constrainss, and constraints involving aggregate functions.

Let us now summarize the enforcement algorithm. Recall that an update transaction updates all tuples of relation R that satisfy some qualification. The algorithm acts in two steps. The first step generates the differential relations R^+ and R^- from R. The second step simply consists of retrieving the tuples of R^+ and R^-, which do not satisfy the pretests. If no tuples are retrieved, the constraint is valid. Otherwise, it is violated.

Example 5.23. Suppose there is a deletion on PROJ. Enforcing (PROJ, **DELETE**, C_2) consists in generating the following statement:

result ← retrieve all tuples of PROJ$^-$ where $\neg(C_2)$

Then, if the result is empty, the assertion is verified by the update and consistency is preserved. ♦

5.3.2 Distributed Semantic Integrity Control

In this section we present algorithms for ensuring the semantic integrity of distributed databases. They are extensions of the simplification method discussed previously. In what follows, we assume global transaction management capabilities, as provided for homogeneous systems or multidatabase systems. Thus, the two main problems of designing an integrity manager for such a distributed DBMS are the definition and storage of assertions, and the enforcement of these constraints. We will also discuss the issues involved in integrity constraint checking when there is no global transaction support.

5.3.2.1 Definition of Distributed Integrity Constraints

An integrity constraint is supposed to be expressed in tuple relational calculus. Each assertion is seen as a query qualification that is either true or false for each tuple in the Cartesian product of the relations determined by the tuple variables. Since assertions can involve data stored at different sites, the storage of the constraints must be decided so as to minimize the cost of integrity checking. There is a strategy based on a taxonomy of integrity constraints that distinguishes three classes:

1. *Individual constraints*: single-relation single-variable constraints. They refer only to tuples to be updated independently of the rest of the database. For instance, the domain constraint of Example 5.15 is an individual assertion.
2. *Set-oriented constraints*: include single-relation multivariable constraints such as functional dependency (Example 5.14) and multirelation multivariable constraints such as foreign key constraints (Example 5.13).

3. *Constraints involving aggregates*: require special processing because of the cost of evaluating the aggregates. The assertion in Example 5.19 is representative of a constraint of this class.

The definition of a new integrity constraint can be started at one of the sites that store the relations involved in the assertion. Remember that the relations can be fragmented. A fragmentation predicate is a particular case of assertion of class 1. Different fragments of the same relation can be located at different sites. Thus, defining an integrity assertion becomes a distributed operation, which is done in two steps. The first step is to transform the high-level assertions into pretests, using the techniques discussed in the preceding section. The next step is to store pretests according to the class of constraints. Constraints of class 3 are treated like those of class 1 or 2, depending on whether they are individual or set-oriented.

Individual constraints.

The constraint definition is sent to all other sites that contain fragments of the relation involved in the constraint. The constraint must be compatible with the relation data at each site. Compatibility can be checked at two levels: predicate and data. First, predicate compatibility is verified by comparing the constraint predicate with the fragment predicate. A constraint C is not compatible with a fragment predicate p if "C is true" implies that "p is false," and is compatible with p otherwise. If non-compatibility is found at one of the sites, the constraint definition is globally rejected because tuples of that fragment do not satisfy the integrity constraints. Second, if predicate compatibility has been found, the constraint is tested against the instance of the fragment. If it is not satisfied by that instance, the constraint is also globally rejected. If compatibility is found, the constraint is stored at each site. Note that the compatibility checks are performed only for pretests whose update type is "insert" (the tuples in the fragments are considered "inserted").

Example 5.24. Consider relation EMP, horizontally fragmented across three sites using the predicates

$$p_1 : 0 \leq \text{ENO} < \text{"E3"}$$
$$p_2 : \text{"E3"} \leq \text{ENO} \leq \text{"E6"}$$
$$p_3 : \text{ENO} > \text{"E6"}$$

and the domain constraint C: ENO $<$ "E4". Constraint C is compatible with p_1 (if C is true, p_1 is true) and p_2 (if C is true, p_2 is not necessarily false), but not with p_3 (if C is true, then p_3 is false). Therefore, constraint C should be globally rejected because the tuples at site 3 cannot satisfy C, and thus relation EMP does not satisfy C. ♦

Set-oriented constraints.

Set-oriented constraint are multivariable; that is, they involve join predicates. Although the assertion predicate may be multirelation, a pretest is associated with a single relation. Therefore, the constraint definition can be sent to all the sites that store a fragment referenced by these variables. Compatibility checking also involves fragments of the relation used in the join predicate. Predicate compatibility is useless here, because it is impossible to infer that a fragment predicate p is false if the constraint C (based on a join predicate) is true. Therefore C must be checked for compatibility against the data. This compatibility check basically requires joining each fragment of the relation, say R, with all fragments of the other relation, say S, involved in the constraint predicate. This operation may be expensive and, as any join, should be optimized by the distributed query processor. Three cases, given in increasing cost of checking, can occur:

1. The fragmentation of R is derived (see Chapter 3) from that of S based on a semijoin on the attribute used in the assertion join predicate.
2. S is fragmented on join attribute.
3. S is not fragmented on join attribute.

In the first case, compatibility checking is cheap since the tuple of S matching a tuple of R is at the same site. In the second case, each tuple of R must be compared with at most one fragment of S, because the join attribute value of the tuple of R can be used to find the site of the corresponding fragment of S. In the third case, each tuple of R must be compared with all fragments of S. If compatibility is found for all tuples of R, the constraint can be stored at each site.

Example 5.25. Consider the set-oriented pretest (ASG, **INSERT**, C_1) defined in Example 5.16, where C_1 is

$$\forall\, \mathbf{NEW} \in \text{ASG}^+, \exists j \in \text{PROJ} :\ \mathbf{NEW}.\text{PNO} = j.\text{PNO}$$

Let us consider the following three cases:

1. ASG is fragmented using the predicate

   $$\text{ASG} \ltimes_{\text{PNO}} \text{PROJ}_i$$

   where PROJ_i is a fragment of relation PROJ. In this case each tuple **NEW** of ASG has been placed at the same site as tuple j such that **NEW**.PNO = j.PNO. Since the fragmentation predicate is identical to that of C_1, compatibility checking does not incur communication.
2. PROJ is horizontally fragmented based on the two predicates

   $$p_1 : \text{PNO} < \text{“P3”}$$
   $$p_2 : \text{PNO} \geq \text{“P3”}$$

In this case each tuple **NEW** of ASG is compared with either fragment PROJ_1, if **NEW**.PNO $<$ "P3", or fragment PROJ_2 if **NEW**.PNO $\geq$ "P3".

3. PROJ is horizontally fragmented based on the two predicates

p_1 : PNAME = "CAD/CAM"
p_2 : PNAME $\neq$ "CAD/CAM"

In this case each tuple of ASG must be compared with both fragments PROJ_1 and PROJ_2.

♦

5.3.2.2 Enforcement of Distributed Integrity Assertions

Enforcing distributed integrity assertions is more complex than needed in centralized DBMSs, even with global transaction management support. The main problem is to decide where (at which site) to enforce the integrity constraints. The choice depends on the class of the constraint, the type of update, and the nature of the site where the update is issued (called the *query master site*). This site may, or may not, store the updated relation or some of the relations involved in the integrity constraints. The critical parameter we consider is the cost of transferring data, including messages, from one site to another. We now discuss the different types of strategies according to these criteria.

Individual constraints.

Two cases are considered. If the update transaction is an insert statement, all the tuples to be inserted are explicitly provided by the user. In this case, all individual constraints can be enforced at the site where the update is submitted. If the update is a qualified update (delete or modify statements), it is sent to the sites storing the relation that will be updated. The query processor executes the update qualification for each fragment. The resulting tuples at each site are combined into one temporary relation in the case of a delete statement, or two, in the case of a modify statement (i.e., R^+ and R^-). Each site involved in the distributed update enforces the assertions relevant at that site (e.g., domain constraints when it is a delete).

Set-oriented constraints.

We first study single-relation constraints by means of an example. Consider the functional dependency of Example 5.14. The pretest associated with update type INSERT is

(EMP, **INSERT**, C)

where C is

$$(\forall e \in \text{EMP})(\forall \textbf{NEW1} \in \text{EMP})(\forall \textbf{NEW2} \in \text{EMP}) \quad (1)$$

$$(\textbf{NEW1}.\text{ENO} = \text{e.ENO} \Rightarrow \textbf{NEW1}.\text{ENAME} = \text{e.ENAME}) \wedge \quad (2)$$

$$(\textbf{NEW1}.\text{ENO} = \textbf{NEW2}.\text{ENO} \Rightarrow \textbf{NEW1}.\text{ENAME} = \textbf{NEW2}.\text{ENAME}) \quad (3)$$

The second line in the definition of C checks the constraint between the inserted tuples (NEW1) and the existing ones (e), while the third checks it between the inserted tuples themselves. That is why two variables (NEW1 and NEW2) are declared in the first line.

Consider now an update of EMP. First, the update qualification is executed by the query processor and returns one or two temporary relations, as in the case of individual constraints. These temporary relations are then sent to all sites storing EMP. Assume that the update is an INSERT statement. Then each site storing a fragment of EMP will enforce constraint C described above. Because e in C is universally quantified, C must be satisfied by the local data of each site. This is due to the fact that $\forall x \in \{a_1, \ldots, a_n\} f(x)$ is equivalent to $[f(a_1) \wedge f(a_2) \wedge \cdots \wedge f(a_n)]$. Thus the site where the update is submitted must receive for each site a message indicating that this constraint is satisfied and that it is a condition for all sites. If the constraint is not true for one site, this site sends an error message indicating that the constraint has been violated. The update is then invalid, and it is the responsibility of the integrity manager to decide if the entire transaction must be rejected using the global transaction manager.

Let us now consider multirelation constraints. For the sake of clarity, we assume that the integrity constraints do not have more than one tuple variable ranging over the same relation. Note that this is likely to be the most frequent case. As with single-relation constraints, the update is computed at the site where it was submitted. The enforcement is done at the query master site, using the ENFORCE algorithm given in Algorithm 5.2.

Example 5.26. We illustrate this algorithm through an example based on the foreign key constraint of Example 5.13. Let u be an insertion of a new tuple into ASG. The previous algorithm uses the pretest (ASG, **INSERT**, C), where C is

$$\forall \textbf{NEW} \in \text{ASG}^+, \exists j \in \text{PROJ} : \textbf{NEW}.\text{PNO} = j.\text{PNO}$$

For this constraint, the retrieval statement is to retrieve all new tuples in ASG^+ where C is not true. This statement can be expressed in SQL as

```
SELECT NEW.*
FROM   ASG+ NEW, PROJ
WHERE  COUNT(PROJ.PNO WHERE NEW.PNO = PROJ.PNO)=0
```

Note that **NEW**.* denotes all the attributes of ASG^+. ♦

Thus the strategy is to send new tuples to sites storing relation PROJ in order to perform the joins, and then to centralize all results at the query master site. For each

Algorithm 5.2: ENFORCE Algorithm

```
Input: U: update type; R: relation
begin
    retrieve all compiled assertions (R, U, C_i) ;
    inconsistent ← false ;
    for each compiled assertion do
        result ← all new (respectively old), tuples of R where ¬(C_i)
    if card(result) ≠ 0 then
        inconsistent ← true
    if ¬inconsistent then
        send the tuples to update to all the sites storing fragments of R
    else
        reject the update
end
```

site storing a fragment of PROJ, the site joins the fragment with ASG^+ and sends the result to the query master site, which performs the union of all results. If the union is empty, the database is consistent. Otherwise, the update leads to an inconsistent state and should be rejected, using the global transaction manager. More sophisticated strategies that notify or compensate inconsistencies can also be devised.

Constraints involving aggregates.

These constraints are among the most costly to test because they require the calculation of the aggregate functions. The aggregate functions generally manipulated are MIN, MAX, SUM, and COUNT. Each aggregate function contains a projection part and a selection part. To enforce these constraints efficiently, it is possible to produce pretest that isolate redundant data which can be stored at each site storing the associated relation [Bernstein and Blaustein, 1982]. This data is what we called *materialized views* in Section 5.1.2.

5.3.2.3 Summary of Distributed Integrity Control

The main problem of distributed integrity control is that the communication and processing costs of enforcing distributed constraints can be prohibitive. The two main issues in designing a distributed integrity manager are the definition of the distributed assertions and of the enforcement algorithms, which minimize the cost of distributed integrity checking. We have shown in this chapter that distributed integrity control can be completely achieved, by extending a preventive method based on the compilation of semantic integrity constraints into pretests. The method is general since all types of constraints expressed in first-order predicate logic can be handled.

It is compatible with fragment definition and minimizes intersite communication. A better performance of distributed integrity enforcement can be obtained if fragments are defined carefully. Therefore, the specification of distributed integrity constraints is an important aspect of the distributed database design process.

The method described above assumes global transaction support. Without global transaction support as in some loosely-coupled multidatabase systems, the problem is more difficult [Grefen and Widom, 1997]. First, the interface between the constraint manager and the component DBMS is different since constraint checking can no longer be part of the global transaction validation. Instead, the component DBMSs should notify the integrity manager to perform constraint checking after some events, e.g., as a result of local transactions's commitments. This can be done using triggers whose events are updates to relations involved in global constraints. Second, if a global constraint violation is detected, since there is no way to specify global aborts, specific correcting transactions should be provided to produce global database states that are consistent. A family of protocols for global integrity checking has been proposed [Grefen and Widom, 1997]. The root of the family is a simple strategy, based on the computation of differential relations (as in the previous method), which is shown to be safe (correctly identifies constraint violations) but inaccurate (may raise an error event though there is no constraint violation). Inaccuracy is due to the fact that producing differential relations at different times at different sites may yield *phantom* states for the global database, i.e., states that never existed. Extensions of the basic protocol with either timestamping or using local transaction commands are proposed to solve that problem.

5.4 Conclusion

Semantic data and access control includes view management, security control, and semantic integrity control. In the relational framework, these functions can be uniformly achieved by enforcing rules that specify data manipulation control. Solutions initially designed for handling these functions in centralized systems have been significantly extended and enriched for distributed systems, in particular, support for materialized views and group-based discretionary access control. Semantic integrity control has received less attention and is generally not supported by distributed DBMS products.

Full semantic data control is more complex and costly in terms of performance in distributed systems. The two main issues for efficiently performing data control are the definition and storage of the rules (site selection) and the design of enforcement algorithms which minimize communication costs. The problem is difficult since increased functionality (and generality) tends to increase site communication. The problem is simplified if control rules are fully replicated at all sites and harder if site autonomy is to be preserved. In addition, specific optimizations can be done to minimize the cost of data control but with extra overhead such as managing materialized views or redundant data. Thus the specification of distributed data

control must be included in the distributed database design so that the cost of control for update programs is also considered.

5.5 Bibliographic Notes

Semantic data control is well-understood in centralized systems [Ramakrishnan and Gehrke, 2003] and all major DBMSs provide extensive support for it. Research on semantic data control in distributed systems started in the early 1980's with the R* project at IBM Research and has increased much since then to address new important applications such as data warehousing or data integration.

Most of the work on view management has concerned updates through views and support for materialized views. The two basic papers on centralized view management are [Chamberlin et al., 1975] and [Stonebraker, 1975]. The first reference presents an integrated solution for view and authorization management in System R. The second reference describes INGRES's query modification technique for uniformly handling views, authorizations, and semantic integrity control. This method was presented in Section 5.1.

Theoretical solutions to the problem of view updates are given in [Bancilhon and Spyratos, 1981; Dayal and Bernstein, 1978], and [Keller, 1982]. The first of these is the seminal paper on view update semantics [Bancilhon and Spyratos, 1981] where the authors formalize the view invariance property after updating, and show how a large class of views including joins can be updated. Semantic information about the base relations is particularly useful for finding unique propagation of updates. However, the current commercial systems are very restrictive in supporting updates through views.

Materialized views have received much attention. The notion of snapshot for optimizing view derivation in distributed database systems is due to [Adiba and Lindsay, 1980]. Adiba [1981] generalizes the notion of snapshot by that of derived relation in a distributed context. He also proposes a unified mechanism for managing views, and snapshots, as well as fragmented and replicated data. Gupta and Mumick [1999c] have edited a thorough collection of papers on materialized view management in. In [Gupta and Mumick, 1999a], they describe the main techniques to perform incremental maintenance of materialized views. The counting algorithm which we presented in Section 5.1.3 has been proposed in [Gupta et al., 1993].

Security in computer systems in general is presented in [Hoffman, 1977]. Security in centralized database systems is presented in [Lunt and Fernández, 1990; Castano et al., 1995]. Discretionary access control in distributed systems has first received much attention in the context of the R* project. The access control mechanism of System R Griffiths and Wade [1976] is extended in [Wilms and Lindsay, 1981] to handle groups of users and to run in a distributed environment. Multilevel access control for distributed DBMS has recently gained much interest. The seminal paper on multilevel access control is the Bell and Lapaduda model originally designed for operating system security [Bell and Lapuda, 1976]. Multilevel access control for

databases is described in [Lunt and Fernández, 1990; Jajodia and Sandhu, 1991]. A good introduction to multilevel security in relational DBMS can be found in [Rjaibi, 2004]. Transaction management in multilevel secure DBMS is addressed in [Ray et al., 2000; Jajodia et al., 2001]. Extensions of multilevel access control for distributed DBMS are proposed in [Thuraisingham, 2001].

The content of Section 5.3 comes largely from the work on semantic integrity control described in [Simon and Valduriez, 1984, 1986] and [Simon and Valduriez, 1987]. In particular, [Simon and Valduriez, 1986] extends a preventive strategy for centralized integrity control based on pretests to run in a distributed environment, assuming global transaction support. The initial idea of declarative methods, that is, to use assertions of predicate logic to specify integrity constraints, is due to [Florentin, 1974]. The most important declarative methods are in [Bernstein et al., 1980a; Blaustein, 1981; Nicolas, 1982; Simon and Valduriez, 1984], and [Stonebraker, 1975]. The notion of concrete views for storing redundant data is described in [Bernstein and Blaustein, 1982]. Note that concrete views are useful in optimizing the enforcement of constraints involving aggregates. [Civelek et al., 1988; Sheth et al., 1988b] and Sheth et al. [1988a] describe systems and tools for semantic data control, particularly view management. Semantic intergrity checking in loosely-coupled multidatabase systems without global transaction support is addressed in [Grefen and Widom, 1997].

Exercises

Problem 5.1. Define in SQL-like syntax a view of the engineering database V(ENO, ENAME, PNO, RESP), where the duration is 24. Is view V updatable? Assume that relations EMP and ASG are horizontally fragmented based on access frequencies as follows:

Site 1	Site 2	Site 3
EMP_1	EMP_2	
	ASG_1	ASG_2

where

$$EMP_1 = \sigma_{TITLE \neq \text{“Engineer”}}(EMP)$$
$$EMP_2 = \sigma_{TITLE = \text{“Engineer”}}(EMP)$$
$$ASG_1 = \sigma_{0<DUR<36}(ASG)$$
$$ASG_2 = \sigma_{DUR \geq 36}(ASG)$$

At which site(s) should the definition of V be stored without being fully replicated, to increase locality of reference?

Problem 5.2. Express the following query: names of employees in view V who work on the CAD project.

Problem 5.3 (*). Assume that relation PROJ is horizontally fragmented as

$\text{PROJ}_1 = \sigma_{\text{PNAME} = \text{“CAD”}}(\text{PROJ})$
$\text{PROJ}_2 = \sigma_{\text{PNAME} \neq \text{“CAD”}}(\text{PROJ})$

Modify the query obtained in Exercise 5.2 to a query expressed on the fragments.

Problem 5.4 (**). Propose a distributed algorithm to efficiently refresh a snapshot at one site derived by projection from a relation horizontally fragmented at two other sites. Give an example query on the view and base relations which produces an inconsistent result.

Problem 5.5 (*). Consider the view EG of Example 5.5 which uses relations EMP and ASG as base data and assume its state is derived from that of Example 3.1, so that EG has 9 tuples (see Figure 5.4). Assume that tuple ⟨E3, P3, Consultant, 10⟩ from ASG is updated to ⟨E3, P3, Engineer, 10⟩. Apply the basic counting algorithm for refreshing the view EG. What projected attributes should be added to view EG to make it self-maintainable?

Problem 5.6. Propose a relation schema for storing the access rights associated with user groups in a distributed database catalog, and give a fragmentation scheme for that relation, assuming that all members of a group are at the same site.

Problem 5.7 (**). Give an algorithm for executing the REVOKE statement in a distributed DBMS, assuming that the GRANT privilege can be granted only to a group of users where all its members are at the same site.

Problem 5.8 (**). Consider the multilevel relation PROJ** in Figure 5.8. Assuming that there are only two classification levels for attributes (S and C), propose an allocation of PROJ** on two sites using fragmentation and replication that avoids covert channels on read queries. Discuss the constraints on updates for this allocation to work.

Problem 5.9. Using the integrity constraint specification language of this chapter, express an integrity constraint which states that the duration spent in a project cannot exceed 48 months.

Problem 5.10 (*). Define the pretests associated with integrity constraints covered in Examples 5.11 to 5.14.

Problem 5.11. Assume the following vertical fragmentation of relations EMP, ASG and PROJ:

Site 1	Site 2	Site 3	Site 4
EMP_1	EMP_2		
	PROJ_1	PROJ_2	
		ASG_1	ASG_2

where

$EMP_1 = \Pi_{ENO, ENAME}(EMP)$
$EMP_2 = \Pi_{ENO, TITLE}(EMP)$
$PROJ_1 = \Pi_{PNO, PNAME}(PROJ)$
$PROJ_2 = \Pi_{PNO, BUDGET}(PROJ)$
$ASG_1 = \Pi_{ENO, PNO, RESP}(ASG)$
$ASG_2 = \Pi_{ENO, PNO, DUR}(ASG)$

Where should the pretests obtained in Exercise 5.9 be stored?

Problem 5.12 (**). Consider the following set-oriented constraint:

```
CHECK ON e:EMP, a:ASG
    (e.ENO = a.ENO and (e.TITLE = "Programmer")
    IF a.RESP = "Programmer")
```

What does it mean? Assuming that EMP and ASG are allocated as in the previous exercice, define the corresponding pretests and theri storage. Apply algorithm ENFORCE for an update of type INSERT in ASG.

Problem 5.13 (**). Assume a distributed multidatabase system with no global transaction support. Assume also that there are two sites, each with a (different) EMP relation and a integrity manager that communicates with the component DBMS. Suppose that we want to have a global unique key constraint on EMP. Propose a simple strategy using differential relations to check this constraint. Discuss the possible actions when a constraint is violated.

Chapter 6
Overview of Query Processing

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

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

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

M.T. Özsu and P. Valduriez, *Principles of Distributed Database Systems: Third Edition*,
DOI 10.1007/978-1-4419-8834-8_6,

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

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

6.1 Query Processing Problem

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

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

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

and the following simple user query:

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

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

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

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

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

and

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

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

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

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

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

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

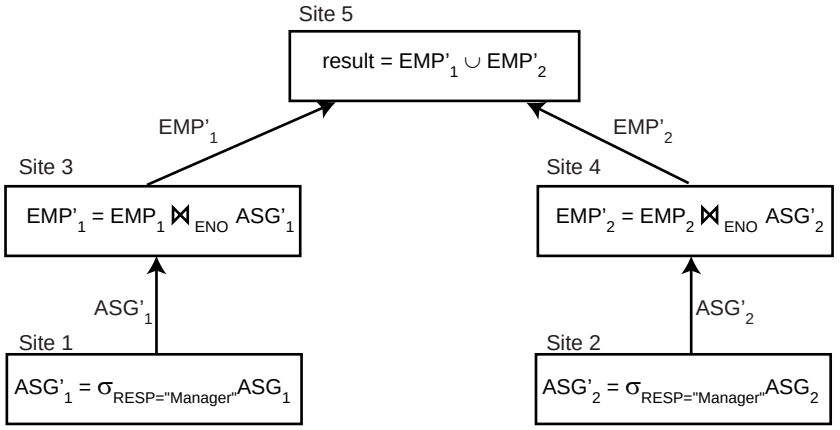
We assume that relations EMP and ASG are horizontally fragmented as follows:

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

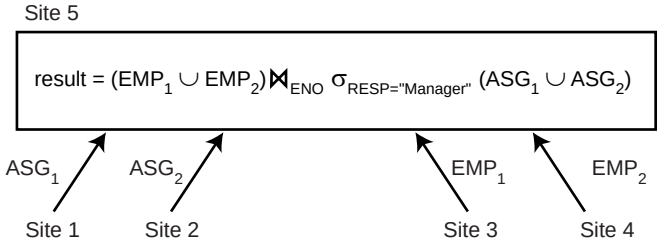
Fragments ASG_1, ASG_2, EMP_1, and EMP_2 are stored at sites 1, 2, 3, and 4, respectively, and the result is expected at site 5.

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

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



(a) Strategy A



(b) Strategy B

Fig. 6.1 Equivalent Distributed Execution Strategies

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

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

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

The cost of strategy B can be derived as follows:

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

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

6.2 Objectives of Query Processing

As stated before, the objective of query processing in a distributed context is to transform a high-level query on a distributed database, which is seen as a single database by the users, into an efficient execution strategy expressed in a low-level language on local databases. We assume that the high-level language is relational calculus, while the low-level language is an extension of relational algebra with communication operators. The different layers involved in the query transformation are detailed in Section 6.5. An important aspect of query processing is query optimization. Because many execution strategies are correct transformations of the same high-level query, the one that optimizes (minimizes) resource consumption should be retained.

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

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

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

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

6.3 Complexity of Relational Algebra Operations

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

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

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

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

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

Fig. 6.2 Complexity of Relational Algebra Operations

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

6.4 Characterization of Query Processors

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

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

6.4.1 Languages

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

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

6.4.2 Types of Optimization

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

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

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

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

6.4.3 Optimization Timing

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

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

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

6.4.4 Statistics

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

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

6.4.5 Decision Sites

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

6.4.6 Exploitation of the Network Topology

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

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

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

6.4.7 Exploitation of Replicated Fragments

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

6.4.8 Use of Semijoins

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

6.5 Layers of Query Processing

In Chapter 1 we have seen where query processing fits within the distributed DBMS architecture. The problem of query processing can itself be decomposed into several subproblems, corresponding to various layers. In Figure 6.3 a generic layering scheme for query processing is shown where each layer solves a well-defined subproblem. To simplify the discussion, let us assume a static and semicentralized query processor that does not exploit replicated fragments. The input is a query on global data expressed in relational calculus. This query is posed on global (distributed) relations, meaning that data distribution is hidden. Four main layers are involved in distributed query processing. The first three layers map the input query into an optimized

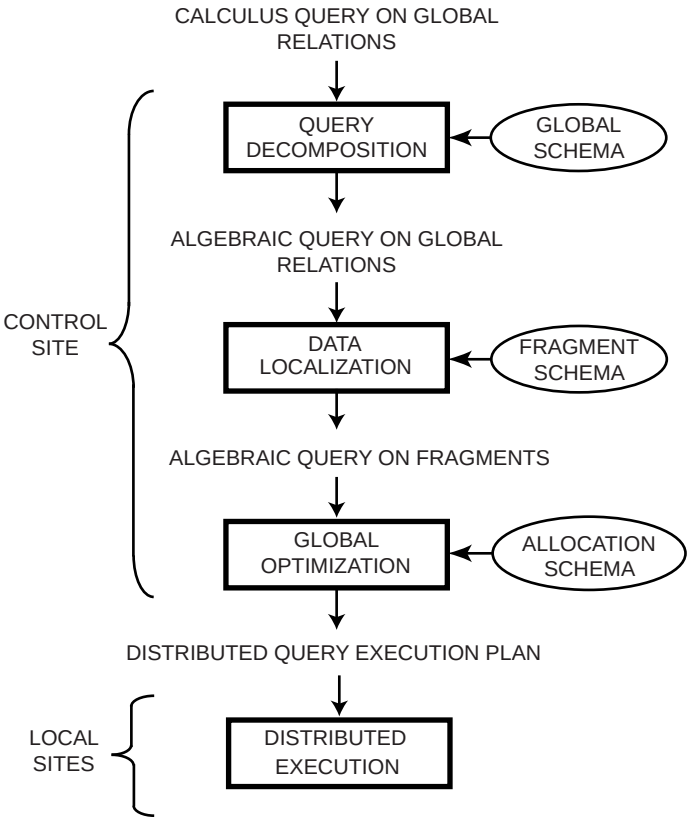


Fig. 6.3 Generic Layering Scheme for Distributed Query Processing

distributed query execution plan. They perform the functions of *query decomposition*, *data localization*, and *global query optimization*. Query decomposition and data localization correspond to query rewriting. The first three layers are performed by a central control site and use schema information stored in the global directory. The fourth layer performs *distributed query execution* by executing the plan and returns the answer to the query. It is done by the local sites and the control site. The first two layers are treated extensively in Chapter 7, while the two last layers are detailed in Chapter 8. In the remainder of this chapter we present an overview of these four layers.

6.5.1 Query Decomposition

The first layer decomposes the calculus query into an algebraic query on global relations. The information needed for this transformation is found in the global

conceptual schema describing the global relations. However, the information about data distribution is not used here but in the next layer. Thus the techniques used by this layer are those of a centralized DBMS.

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

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

Third, the correct query (still expressed in relational calculus) is *simplified*. One way to simplify a query is to eliminate redundant predicates. Note that redundant queries are likely to arise when a query is the result of system transformations applied to the user query. As seen in Chapter 5, such transformations are used for performing semantic data control (views, protection, and semantic integrity control).

Fourth, the calculus query is *restructured* as an algebraic query. Recall from Section 6.1 that several algebraic queries can be derived from the same calculus query, and that some algebraic queries are "better" than others. The quality of an algebraic query is defined in terms of expected performance. The traditional way to do this transformation toward a "better" algebraic specification is to start with an initial algebraic query and transform it in order to find a "good" one. The initial algebraic query is derived immediately from the calculus query by translating the predicates and the target statement into relational operators as they appear in the query. This directly translated algebra query is then restructured through transformation rules. The algebraic query generated by this layer is good in the sense that the worse executions are typically avoided. For instance, a relation will be accessed only once, even if there are several select predicates. However, this query is generally far from providing an optimal execution, since information about data distribution and fragment allocation is not used at this layer.

6.5.2 Data Localization

The input to the second layer is an algebraic query on global relations. The main role of the second layer is to localize the query's data using data distribution information in the fragment schema. In Chapter 3 we saw that relations are fragmented and stored in disjoint subsets, called fragments, each being stored at a different site. This layer determines which fragments are involved in the query and transforms the distributed query into a query on fragments. Fragmentation is defined by fragmentation predicates that can be expressed through relational operators. A global relation can be reconstructed by applying the fragmentation rules, and then deriving a program, called a *localization program*, of relational algebra operators, which then act on fragments. Generating a query on fragments is done in two steps. First, the query

is mapped into a fragment query by substituting each relation by its reconstruction program (also called *materialization program*), discussed in Chapter 3. Second, the fragment query is simplified and restructured to produce another "good" query. Simplification and restructuring may be done according to the same rules used in the decomposition layer. As in the decomposition layer, the final fragment query is generally far from optimal because information regarding fragments is not utilized.

6.5.3 Global Query Optimization

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

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

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

included on fragments. It is typically represented and saved (for future executions) as a *distributed query execution plan* .

6.5.4 Distributed Query Execution

The last layer is performed by all the sites having fragments involved in the query. Each subquery executing at one site, called a *local query*, is then optimized using the local schema of the site and executed. At this time, the algorithms to perform the relational operators may be chosen. Local optimization uses the algorithms of centralized systems (see Chapter 8).

6.6 Conclusion

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

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

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

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

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

6.7 Bibliographic Notes

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

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

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

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

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

Chapter 7
Query Decomposition and Data Localization

In Chapter 6 we discussed a generic layering scheme for distributed query processing in which the first two layers are responsible for query decomposition and data localization. These two functions are applied successively to transform a calculus query specified on distributed relations (i.e., global relations) into an algebraic query defined on relation fragments. In this chapter we present the techniques for query decomposition and data localization.

Query decomposition maps a distributed calculus query into an algebraic query on global relations. The techniques used at this layer are those of the centralized DBMS since relation distribution is not yet considered at this point. The resultant algebraic query is "good" in the sense that even if the subsequent layers apply a straightforward algorithm, the worst executions will be avoided. However, the subsequent layers usually perform important optimizations, as they add to the query increasing detail about the processing environment.

Data localization takes as input the decomposed query on global relations and applies data distribution information to the query in order to localize its data. In Chapter 3 we have seen that to increase the locality of reference and/or parallel execution, relations are fragmented and then stored in disjoint subsets, called fragments, each being placed at a different site. Data localization determines which fragments are involved in the query and thereby transforms the distributed query into a fragment query. Similar to the decomposition layer, the final fragment query is generally far from optimal because quantitative information regarding fragments is not exploited at this point. Quantitative information is used by the query optimization layer that will be presented in Chapter 8.

This chapter is organized as follows. In Section 7.1 we present the four successive phases of query decomposition: normalization, semantic analysis, simplification, and restructuring of the query. In Section 7.2 we describe data localization, with emphasis on reduction and simplification techniques for the four following types of fragmentation: horizontal, vertical, derived, and hybrid.

M.T. Özsu and P. Valduriez, *Principles of Distributed Database Systems: Third Edition*,
DOI 10.1007/978-1-4419-8834-8_7,

7.1 Query Decomposition

Query decomposition (see Figure 6.3) 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. 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.

7.1.1 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:

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

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

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

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. $\neg(\neg p) \Leftrightarrow p$

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 7.1. 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 P1 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

EMP.ENO = ASG.ENO $\wedge$ ASG.PNO = "P1" $\wedge$ (DUR = 12 $\vee$ DUR = 24)

while the qualification in disjunctive normal form is

(EMP.ENO = ASG.ENO $\wedge$ ASG.PNO = "P1" $\wedge$ DUR = 12) $\vee$
(EMP.ENO = ASG.ENO $\wedge$ ASG.PNO = "P1" $\wedge$ DUR = 24)

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

7.1.2 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 7.2. The following SQL query on the engineering database 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.

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

♦

A query is semantically incorrect if its components 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 one of which does not correspond to the result represents a join, whereas an edge whose destination node is the result represents a project. Furthermore, a non-result node may be labeled by a select or a self-join (join of the relation with itself) predicate. An important subgraph of the query graph is the *join graph*, in which only the joins are considered. The join graph is particularly useful in the query optimization phase.

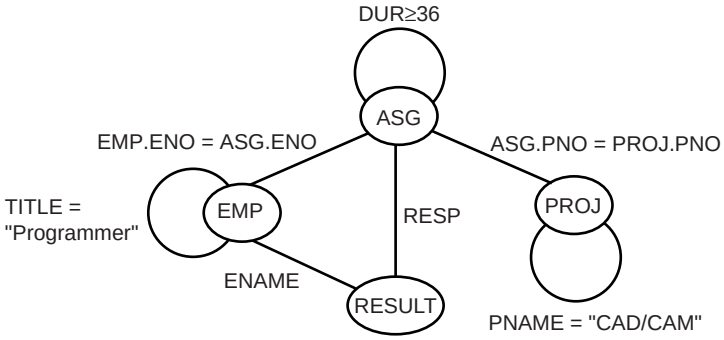
Example 7.3. Let us consider the following query:

> "Find the names and responsibilities of programmers who have been working on the CAD/CAM project for more than 3 years."

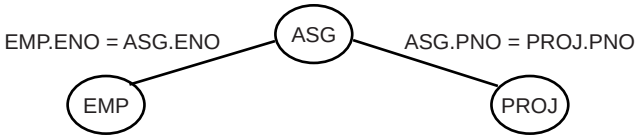
The query expressed in SQL is

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

The query graph for the query above is shown in Figure 7.1a. Figure 7.1b shows the join graph for the graph in Figure 7.1a. ♦



(a) Query graph



(b) Corresponding join graph

Fig. 7.1 Relation Graphs

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 7.4. Let us consider the following SQL query:

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

Its query graph, shown in Figure 7.2, 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 7.3. ♦

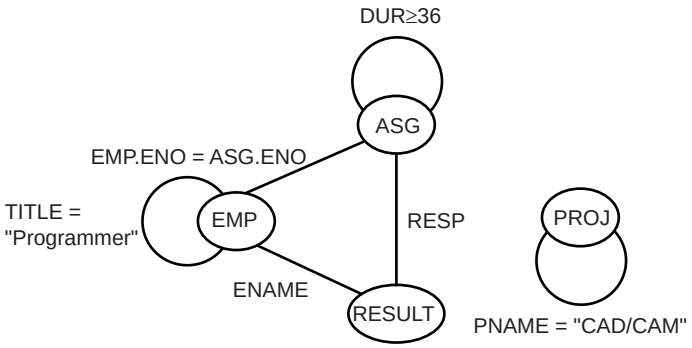


Fig. 7.2 Disconnected Query Graph

7.1.3 Elimination of Redundancy

As we saw in Chapter 5, 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. $p \vee false \Leftrightarrow p$
5. $p \wedge false \Leftrightarrow false$
6. $p \vee true \Leftrightarrow true$
7. $p \wedge \neg p \Leftrightarrow false$
8. $p \vee \neg p \Leftrightarrow true$
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 7.5. 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 7.1.1, 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 7.1.1, 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. ♦

7.1.4 Rewriting

The last step of query decomposition rewrites the query in relational algebra. 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 non-leaf 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 7.6. 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.ENO = EMP.ENO
AND    ASG.PNO = PROJ.PNO
AND    ENAME != "J. Doe"
AND    PROJ.PNAME = "CAD/CAM"
AND    (DUR = 12 OR DUR = 24)
```

can be mapped in a straightforward way in the tree in Figure 7.3. 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]. We now present the six most useful equivalence rules, which concern the basic relational algebra operators. The correctness of these rules has been proven [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:

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

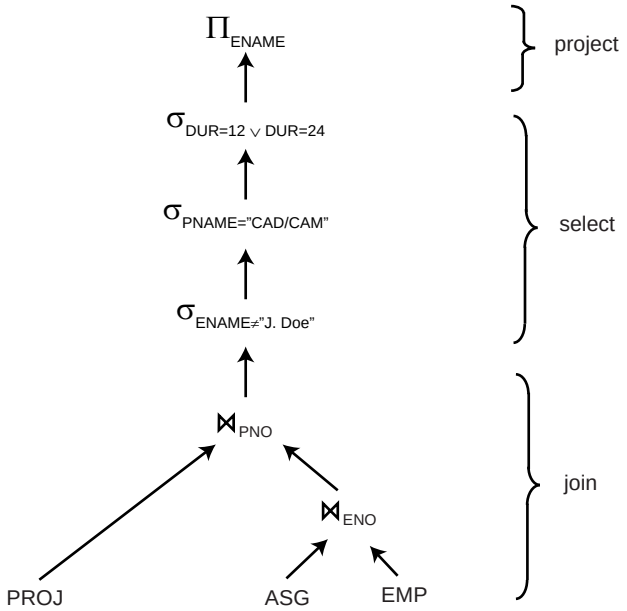


Fig. 7.3 Example of Operator Tree

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 $\sigma_{p_i(A_i)}$ on the same relation, where p_i 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:

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

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 to 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 7.4 is equivalent to the one in Figure 7.3. However, the one in Figure 7.4 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 uses a single heuristic that consists of applying unary operations (select/project) as soon as possible to reduce the size of intermediate relations [Ullman, 1982].

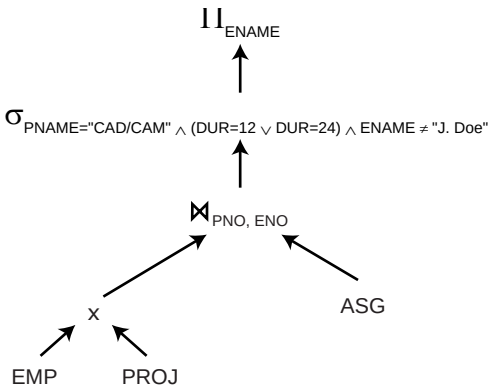


Fig. 7.4 Equivalent Operator Tree

Example 7.7. The restructuring of the tree in Figure 7.3 leads to the tree in Figure 7.5. The resulting tree is good in the sense that repeated access to the same relation (as in Figure 7.3) 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. ♦

7.2 Localization of Distributed Data

In Section 7.1 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 6, 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 3, 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

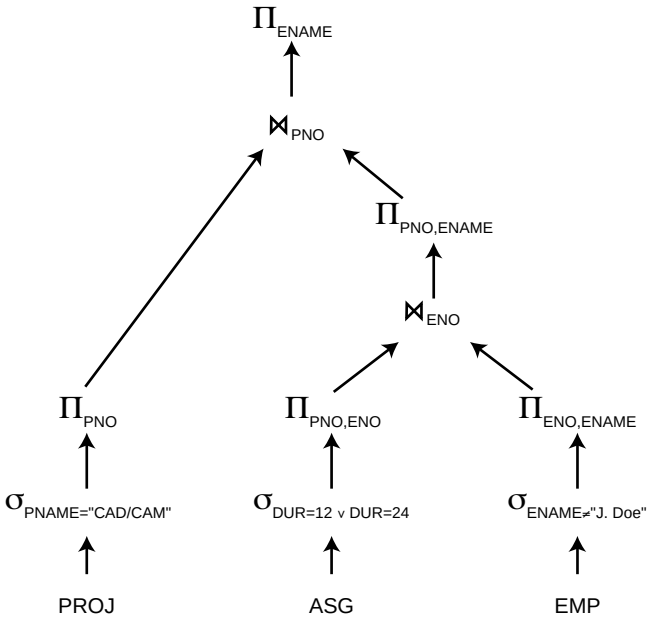


Fig. 7.5 Rewritten Operator Tree

fragments may be replicated, although this can improve performance. Replication is considered in Chapter 8.

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 *localized query*. In general, this approach is inefficient because important restructurings and simplifications of the localized 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 7.1.4.

7.2.1 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 7.8. Relation EMP(ENO, ENAME, TITLE) of Figure 2.3 can be split into three horizontal fragments EMP_1, EMP_2, and EMP_3, defined as follows:

$$\text{EMP}_1 = \sigma_{\text{ENO} \leq \text{"E3"}}(\text{EMP})$$
$$\text{EMP}_2 = \sigma_{\text{"E3"} < \text{ENO} \leq \text{"E6"}}(\text{EMP})$$
$$\text{EMP}_3 = \sigma_{\text{ENO} > \text{"E6"}}(\text{EMP})$$

Note that this fragmentation of the EMP relation is different from the one discussed in Example 3.12.

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

$$\text{EMP} = \text{EMP}_1 \cup \text{EMP}_2 \cup \text{EMP}_3$$

Thus the localized form of any query specified on EMP is obtained by replacing it by $(\text{EMP}_1 \cup \text{EMP}_2 \cup \text{EMP}_3$. ♦

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.

7.2.1.1 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 $R_j = \sigma_{p_j}(R)$, the rule can be stated formally as follows:

Rule 1: $\sigma_{p_i}(R_j) = \phi$ if $\forall x$ in $R : \neg(p_i(x) \wedge p_j(x))$

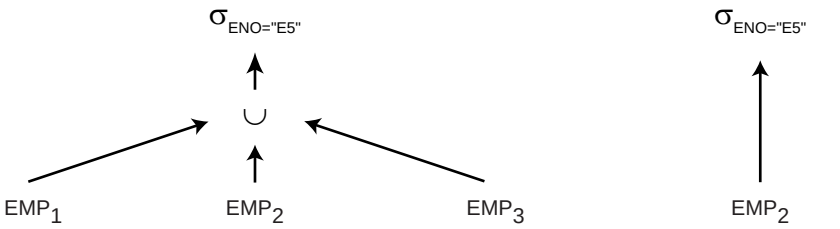
where p_i and p_j are selection predicates, x denotes a tuple, and $p(x)$ denotes "predicate p holds for x."

For example, the selection predicate ENO="E1" conflicts with the predicates of fragments EMP_2 and EMP_3 of Example 7.8 (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 7.9. 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 localized query of Figure 7.6a. By commuting the selection with the union operation, it is easy to detect that the selection predicate contradicts the predicates of



(a) Localized query (b) Reduced query

Fig. 7.6 Reduction for Horizontal Fragmentation (with Selection)

EMP_1 and EMP_3, thereby producing empty relations. The reduced query is simply applied to EMP_2 as shown in Figure 7.6b. ♦

7.2.1.2 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, thus yielding an empty result. 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: $R_i \bowtie R_j = \phi$ if $\forall x$ in $R_i, \forall y$ in $R_j : \neg(p_i(x) \wedge p_j(y))$

The determination of useless joins and their elimination using rule 2 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 localized query. The localized 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 7.10. 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

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

The equivalent localized query is given in Figure 7.7a. 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 7.7b). ♦

7.2.2 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 7.11. Relation EMP can be divided into two vertical fragments where the key attribute ENO is duplicated:

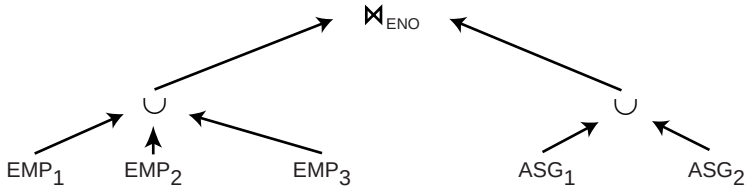
$$EMP_1 = \Pi_{ENO,ENAME}(EMP)$$
$$EMP_2 = \Pi_{ENO,TITLE}(EMP)$$

The localization program is

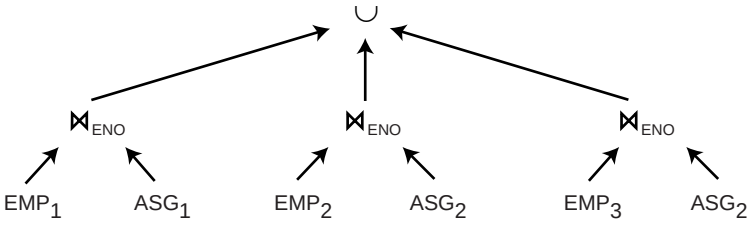
$$EMP = EMP_1 \bowtie_{ENO} EMP_2$$

♦

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



(a) Localized query



(b) Reduced query

Fig. 7.7 Reduction by Horizontal Fragmentation (with Join)

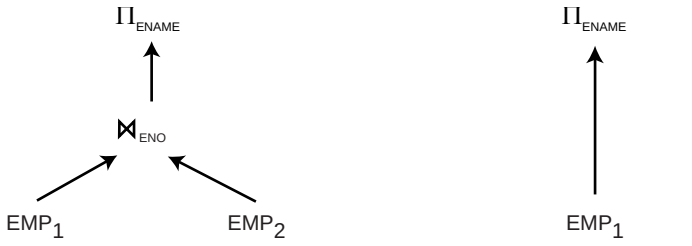
with the projection attributes (except the key of the relation) produce useless, though not empty relations. Given a relation R, defined over attributes $A = \{A_1, \ldots, A_n\}$, which is vertically fragmented as $R_i = \Pi_{A'}(R)$, where $A' \subseteq A$, the rule can be formally stated as follows:

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

Example 7.12. Let us illustrate the application of this rule using the following example query in SQL:

```
SELECT ENAME
FROM   EMP
```

The equivalent localized query on EMP_1 and EMP_2 (as obtained in Example 7.10) is given in Figure 7.8a. 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 7.8b. ♦



(a) Localized query (b) Reduced query

Fig. 7.8 Reduction for Vertical Fragmentation

7.2.3 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 by using primary horizontal fragmentation when the joined relations are fragmented 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 7.13. 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 3 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 localized query because the number of partial joins usually equals the number of fragments of *R*.

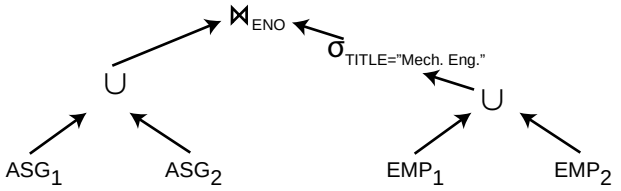
Example 7.14. The reduction by derived fragmentation is illustrated by applying it to the following SQL query, which retrieves all attributes of tuples from EMP and ASG that have the same value of ENO and the title "Mech. Eng.":

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

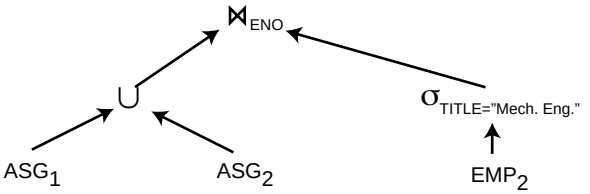
The localized query on fragments EMP_1, EMP_2, ASG_1, and ASG_2, defined previously is given in Figure 7.9a. By pushing selection down to fragments EMP_1 and EMP_2, the query reduces to that of Figure 7.9b. This is because the selection predicate conflicts with that of EMP_1, and thus EMP_1 can be removed. In order to discover conflicting join predicates, we distribute joins over unions. This produces the tree of Figure 7.9c. 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 7.9d is obtained. This example illustrates the value of fragmentation in improving the execution performance of distributed queries. ♦

7.2.4 Reduction for Hybrid Fragmentation

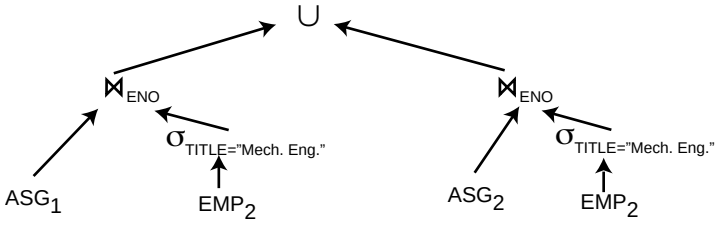
Hybrid fragmentation is obtained by combining the fragmentation functions discussed above. The goal of hybrid fragmentation is to support, efficiently, queries involving projection, selection, and join. Note that the optimization of an operation or of a



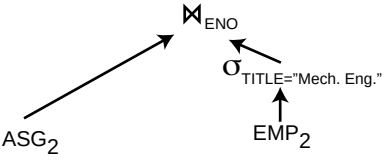
(a) Localized query



(b) Query after pushing selection down



(c) Query after moving unions up



(d) Reduced query after eliminating the left subtree

Fig. 7.9 Reduction for Indirect Fragmentation

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 7.15. Here is an example of hybrid fragmentation of relation EMP:

$$\text{EMP}_1 = \sigma_{\text{ENO} \leq \text{"E4"}}(\Pi_{\text{ENO,ENAME}}(\text{EMP}))$$
$$\text{EMP}_2 = \sigma_{\text{ENO} > \text{"E4"}}(\Pi_{\text{ENO,ENAME}}(\text{EMP}))$$
$$\text{EMP}_3 = \Pi_{\text{ENO,TITLE}}(\text{EMP})$$

In our example, the localization program is

$$\text{EMP} = (\text{EMP}_1 \cup \text{EMP}_2) \bowtie_{\text{ENO}} \text{EMP}_3$$

♦

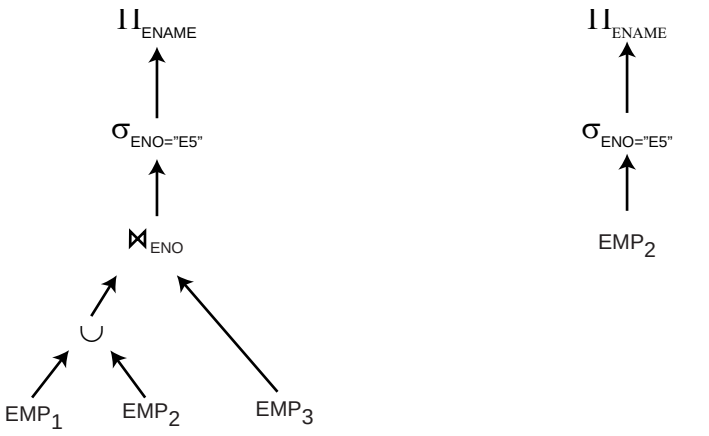
Queries on hybrid fragments can be reduced by combining the rules used, respectively, in primary horizontal, vertical, and derived horizontal fragmentation. These rules can be summarized as follows:

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.

Example 7.16. The following example query in SQL illustrates the application of rules (1) and (2) to the horizontal-vertical fragmentation of relation EMP into EMP_1, EMP_2 and EMP_3 given above:

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

The localized query of Figure 7.10a can be reduced by first pushing selection down, eliminating fragment EMP_1, and then pushing projection down, eliminating fragment EMP_3. The reduced query is given in Figure 7.10b. ♦



(a) Localized query (b) Reduced query

Fig. 7.10 Reduction for Hybrid Fragmentation

7.3 Conclusion

In this chapter we focused on the techniques for query decomposition and data localization layers of the localized query processing scheme that was introduced in Chapter 6. Query decomposition and data localization are the two successive functions that map a calculus query, expressed on distributed relations, into an algebraic query (query decomposition), expressed on relation fragments (data localization).

These two layers can produce a localized query corresponding to the input query in a naive way. Query decomposition can generate an algebraic query simply by translating into relational operations the predicates and the target statement as they appear. Data localization can, in turn, express this algebraic query on relation fragments, by substituting for each distributed relation an algebraic query corresponding to its fragmentation rules.

Many algebraic queries may be equivalent to the same input query. The queries produced with the naive approach are inefficient in general, since important simplifications and optimizations have been missed. Therefore, a localized query expression is restructured using a few transformation rules and heuristics. The rules enable separation of unary operations, grouping of unary operations on the same relation, commuting of unary operations with binary operations, and permutation of the binary operations. Examples of heuristics are to push selections down the tree and do projection as early as possible. In addition to the transformation rules, data localization uses reduction rules to simplify the query further, and therefore optimize it. Two main types of rules may be used. The first one avoids the production of empty relations which are generated by contradicting predicates on the same relation(s). The second type of rule determines which fragments yield useless attributes.

The query produced by the query decomposition and data localization layers is good in the sense that the worse executions are avoided. However, the subsequent layers usually perform important optimizations, as they add to the query increasing detail about the processing environment. In particular, quantitative information regarding fragments has not yet been exploited. This information will be used by the query optimization layer for selecting an "optimal" strategy to execute the query. Query optimization is the subject of Chapter 8.

7.4 Bibliographic NOTES

Traditional techniques for query decomposition are surveyed in [Jarke and Koch, 1984]. Techniques for semantic analysis and simplification of queries have their origins in [Rosenkrantz and Hunt, 1980]. The notion of query graph or connection graph is introduced in [Ullman, 1982]. The notion of query tree, which we called operator tree in this chapter, and the transformation rules to manipulate algebraic expressions have been introduced by Smith and Chang [1975] and developed in [Ullman, 1982]. Proofs of completeness and correctness of the rules are given in the latter reference.

Data localization is treated in detail in [Ceri and Pelagatti, 1983] for horizontally partitioned relations which are referred to as multirelations. In particular, an algebra of qualified relations is defined as an extension of relation algebra, where a qualified relation is a relation name and the qualification of the fragment. Proofs of correctness and completeness of equivalence transformations between expressions of algebra of qualified relations are also given. The formal properties of horizontal and vertical fragmentation are used in [Ceri et al., 1986] to characterize distributed joins over fragmented relations.

Exercises

Problem 7.1. Simplify the following query, expressed in SQL, on our example database using idempotency rules:

```
SELECT ENO
FROM   ASG
WHERE  RESP = "Analyst"
AND    NOT(PNO="P2" OR DUR=12)
AND    PNO != "P2"
AND    DUR=12
```

Problem 7.2. Give the query graph of the following query, in SQL, on our example database:

```
SELECT ENAME, PNAME
FROM   EMP, ASG, PROJ
WHERE  DUR > 12
AND    EMP.ENO = ASG.ENO
AND    PROJ.PNO = ASG.PNO
```

and map it into an operator tree.

Problem 7.3 (*). Simplify the following query:

```
SELECT ENAME, PNAME
FROM   EMP, ASG, PROJ
WHERE  (DUR > 12 OR RESP = "Analyst")
AND    EMP.ENO = ASG.ENO
AND    (TITLE = "Elect. Eng."
OR     ASG.PNO < "P3")
AND    (DUR > 12 OR RESP NOT= "Analyst")
AND    ASG.PNO = PROJ.PNO
```

and transform it into an optimized operator tree using the restructuring algorithm (Section 7.1.4) where select and project operations are applied as soon as possible to reduce the size of intermediate relations.

Problem 7.4 (*). Transform the operator tree of Figure 7.5 back to the tree of Figure 7.3 using the restructuring algorithm. Describe each intermediate tree and show which rule the transformation is based on.

Problem 7.5 (**). Consider the following query on our Engineering database:

```
SELECT ENAME,SAL
FROM   EMP,PROJ,ASG,PAY
WHERE  EMP.ENO = ASG.ENO
AND    EMP.TITLE = PAY.TITLE
AND    (BUDGET>200000 OR DUR>24)
AND    ASG.PNO = PROJ.PNO
AND    (DUR>24 OR PNAME = "CAD/CAM")
```

Compose the selection predicate corresponding to the WHERE clause and transform it, using the idempotency rules, into the simplest equivalent form. Furthermore, compose an operator tree corresponding to the query and transform it, using relational algebra transformation rules, to three equivalent forms.

Problem 7.6. Assume that relation PROJ of the sample database is horizontally fragmented as follows:

$$\text{PROJ}_1 = \sigma_{\text{PNO} \leq \text{"P2"}} (\text{PROJ})$$
$$\text{PROJ}_2 = \sigma_{\text{PNO} > \text{"P2"}} (\text{PROJ})$$

Transform the following query into a reduced query on fragments:

```
SELECT ENO, PNAME
FROM   PROJ,ASG
WHERE  PROJ.PNO = ASG.PNO
AND    PNO = "P4"
```

Problem 7.7 (*). Assume that relation PROJ is horizontally fragmented as in Problem 7.6, and that relation ASG is horizontally fragmented as

$$\text{ASG}_1 = \sigma_{\text{PNO} \leq \text{"P2"}} (\text{ASG})$$
$$\text{ASG}_2 = \sigma_{\text{"P2"} < \text{PNO} \leq \text{"P3"}} (\text{ASG})$$
$$\text{ASG}_3 = \sigma_{\text{PNO} > \text{"P3"}} (\text{ASG})$$

Transform the following query into a reduced query on fragments, and determine whether it is better than the localized query:

```
SELECT RESP, BUDGET
FROM   ASG, PROJ
WHERE  ASG.PNO = PROJ.PNO
AND    PNAME = "CAD/CAM"
```

Problem 7.8 (**). Assume that relation PROJ is fragmented as in Problem 7.6. Furthermore, relation ASG is indirectly fragmented as

$$\text{ASG}_1 = \text{ASG} \ltimes_{\text{PNO}} \text{PROJ}_1$$
$$\text{ASG}_2 = \text{ASG} \ltimes_{\text{PNO}} \text{PROJ}_2$$

and relation EMP is vertically fragmented as

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

Transform the following query into a reduced query on fragments:

```
SELECT ENAME
FROM   EMP,ASG,PROJ
WHERE  PROJ.PNO = ASG.PNO
AND    PNAME = "Instrumentation"
AND    EMP.ENO = ASG.ENO
```

Chapter 8
Optimization of Distributed Queries

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

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

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

M.T. Özsu and P. Valduriez, *Principles of Distributed Database Systems: Third Edition*,
DOI 10.1007/978-1-4419-8834-8_8,

binary operations, such as union, intersection and difference. We also discuss how the semijoin operation can help to process join queries efficiently.

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

8.1 Query Optimization

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

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

8.1.1 Search Space

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

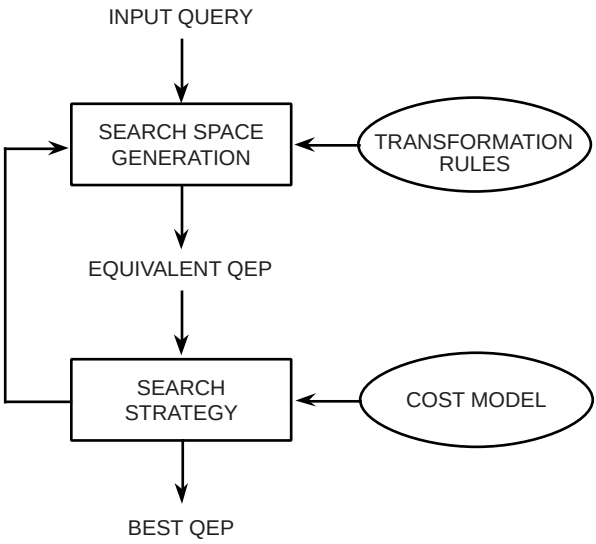


Fig. 8.1 Query Optimization Process

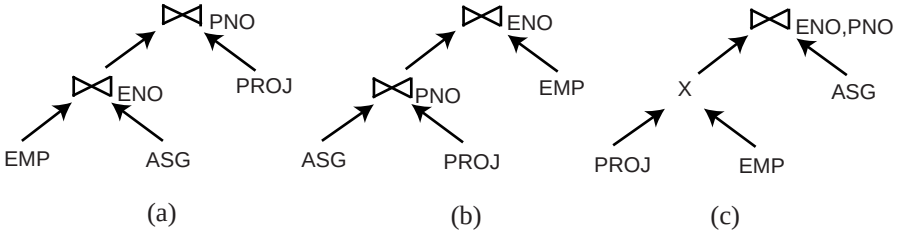


Fig. 8.2 Equivalent Join Trees

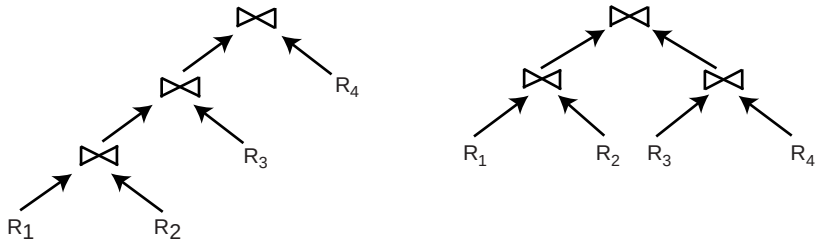
operators are join or Cartesian product. This is because permutations of the join order have the most important effect on performance of relational queries.

Example 8.1. Consider the following query:

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

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

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



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

Fig. 8.3 The Two Major Shapes of Join Trees

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

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

8.1.2 Search Strategy

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

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

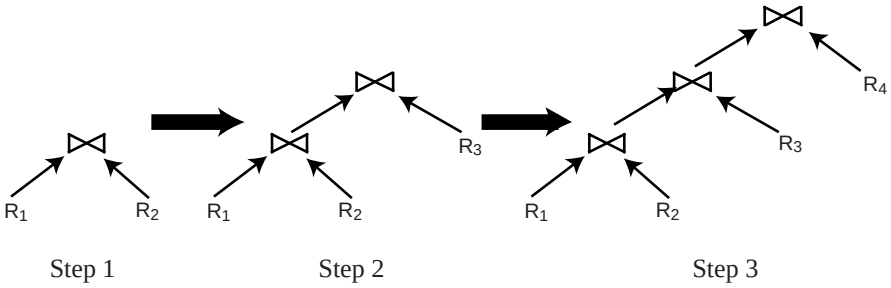


Fig. 8.4 Optimizer Actions in a Deterministic Strategy

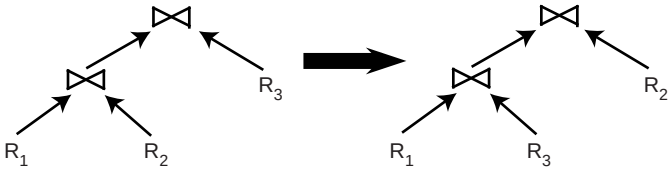


Fig. 8.5 Optimizer Action in a Randomized Strategy

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

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

8.1.3 Distributed Cost Model

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

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

8.1.3.1 Cost Functions

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

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

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

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

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

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

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

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

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

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

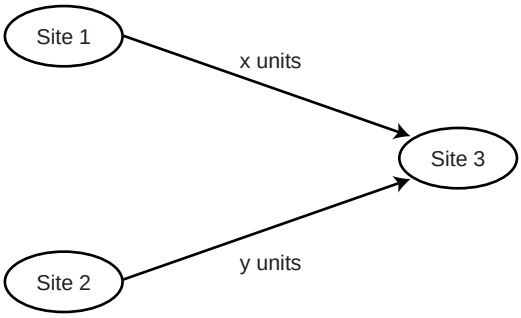


Fig. 8.6 Example of Data Transfers for a Query

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

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

The response time of the same query can be approximated as

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

since the transfers can be done in parallel. ♦

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

8.1.3.2 Database Statistics

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

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

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

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

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

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

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

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

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

8.1.3.3 Cardinalities of Intermediate Results

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

Selection.

The cardinality of selection is

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

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

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

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

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

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

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

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

Projection.

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

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

Cartesian product.

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

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

Join.

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

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

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

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

Semijoin.

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

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

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

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

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

Union.

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

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

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

Difference.

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

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

8.1.3.4 Using Histograms for Selectivity Estimation

The formulae above for estimating the cardinalities of intermediate results of queries rely on the strong assumption that the distribution of attribute values in a relation is uniform. The advantage of this assumption is that the cost of managing the statistics

is minimal since only the number of distinct attribute values is needed. However, this assumption is not practical. In case of skewed data distributions, it can result in fairly inaccurate estimations and QEPs which are far from the optimal.

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

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

Equality predicate.

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

Range predicate.

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

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

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

Example 8.3. Figure 8.7 shows a possible 4-bucket histogram for attribute DUR of a relation ASG with 300 tuples. Let us consider the equality predicate ASG.DUR=18. Since the value "18" fits in bucket b_3, the selectivity factor is 1/12. Since the cardinalty

of b_3 is 50, the cardinality of the selection is 50/12 which is approximately 5 tuples. Let us now consider the range predicate ASG.DUR ≤ 18. We have $min(range_3) = 12$ and $max(range_3) = 24$. The cardinality of the selection is: $100 + 75 + (((18 - 12)/(24 - 12)) * 50) = 200$ tuples. ♦

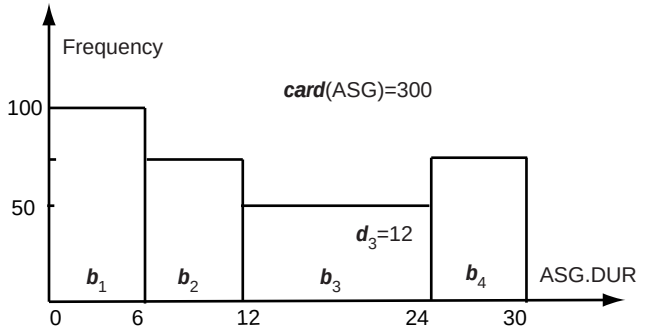


Fig. 8.7 Histogram of Attribute ASG.DUR

8.2 Centralized Query Optimization

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

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

8.2.1 Dynamic Query Optimization

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

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

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

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

```
SELECT R2.A2, R3.A3, ..., Rn.An
FROM   R1, R2, ..., Rn
WHERE  P1(R1.A1')
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 R1.A1 INTO R1'
    FROM   R1
    WHERE  P1(R1.A1')
```

where R_1' is a temporary relation containing the information necessary for the continuation of the query:

```
q'': SELECT R2.A2, ..., Rn.An
     FROM   R1', R2, ..., Rn
     WHERE  P2(R1'.A1, ..., Rn.An)
```

This step has the effect of reducing the size of the relation on which the query q'' is defined. Furthermore, the created relation R_1' may be stored in a particular structure to speed up the following subqueries. For example, the storage of R_1' in a hashed file

on the join attributes of q'' will make processing the join more efficient. Detachment extracts the select operations, which are usually the most selective ones. Therefore, detachment is systematically done whenever possible. Note that this can have adverse effects on performance if the selection has bad selectivity.

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

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

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

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

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

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

The successive detachments of q' may generate

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

Note that other subqueries are also possible.

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

Multirelation queries, which cannot be further detached (e.g., q_{12} and q_{13}), are *irreducible*. A query is irreducible if and only if its query graph is a chain with two nodes or a cycle with k nodes where $k > 2$. Irreducible queries are converted into monorelation queries by tuple substitution. Given an n-relation query q, the tuples of one relation are substituted by their values, thereby producing a set of $(n-1)$-relation

queries. Tuple substitution proceeds as follows. First, one relation in q is chosen for tuple substitution. Let R_1 be that relation. Then for each tuple t_{1i} in R_1, the attributes referred to by in q are replaced by their actual values in t_{1i}, thereby generating a query q' with $n-1$ relations. Therefore, the total number of queries q' produced by tuple substitution is $card(R_1)$. Tuple substitution can be summarized as follows:

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

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

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

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

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

q_{131}:
```
SELECT EMP.ENAME
FROM   EMP
WHERE  EMP.ENO="E1"
```
q_{132}:
```
SELECT EMP.ENAME
FROM   EMP
WHERE  EMP.ENO="E2"
```

These queries may then be executed. ♦

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

Algorithm 8.1: Dynamic-QOA

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

8.2.2 Static Query Optimization

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

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

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

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

The cost of a candidate strategy is a weighted combination of I/O and CPU costs (times). The estimation of such costs (at compile time) is based on a cost model that

provides a cost formula for each low-level operation (e.g., select using a B-tree index with a range predicate). For most operations (except exact match select), these cost formulas are based on the cardinalities of the operands. The cardinality information for the relations stored in the database is found in the database statistics. The cardinality of the intermediate results is estimated based on the operation selectivity factors discussed in Section 8.1.3.

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

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

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

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

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

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

Algorithm 8.2: Static-QOA

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

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

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

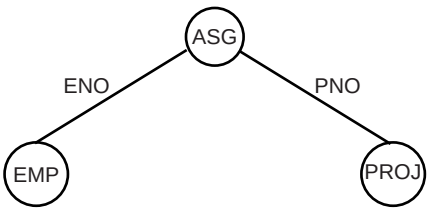


Fig. 8.8 Join Graph of Query q_1

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

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

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

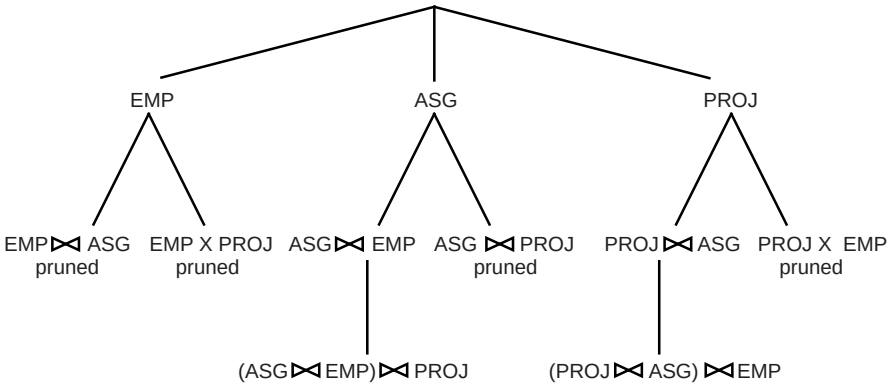


Fig. 8.9 Alternative Join Orders

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

♦

The performance measurements substantiate the important contribution of the CPU time to the total time of the query[Mackert and Lohman, 1986]. The accuracy of the optimizer's estimations is generally good when the relations can be contained in the main memory buffers, but degrades as the relations increase in size and are

written to disk. An important performance parameter that should also be considered for better predictions is buffer utilization.

8.2.3 Hybrid Query Optimization

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

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

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

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

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

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

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

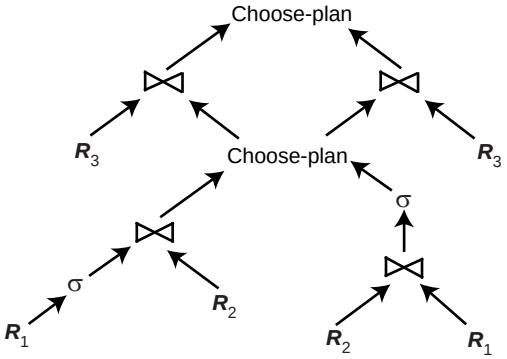


Fig. 8.10 A Dynamic Execution Plan

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

Given a dynamic QEP, produced by a static query optimizer, the choose-plan decisions must be made at query startup time. The most effective solution is to simply evaluate the costs of the participating subplans and compare them. In Algorithm 8.3,

we describe the startup procedure (called Hybrid-QOA) which makes the optimization decisions to produce the final QEP and run it. The algorithm executes the choose-plan operators in bottom-up order and propagates cost information upward in the QEP.

Algorithm 8.3: Hybrid-QOA

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

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

8.3 Join Ordering in Distributed Queries

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

8.3.1 Join Ordering

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

A number of assumptions are necessary to concentrate on the main issues. Since the query is localized and expressed on fragments, we do not need to distinguish

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

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

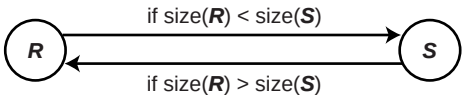


Fig. 8.11 Transfer of Operands in Binary Operation

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

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

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

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

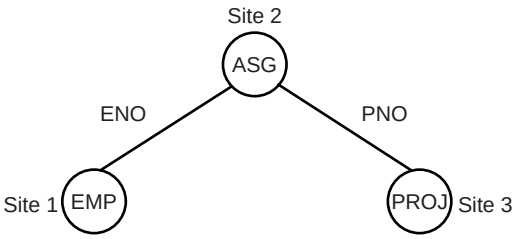


Fig. 8.12 Join Graph of Distributed Query

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

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

8.3.2 Semijoin Based Algorithms

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

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

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

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

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

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

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

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

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

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

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

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

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

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

since if $size(\text{ASG} \ltimes \text{PROJ}) \leq size(\text{ASG})$, 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 8.9. Consider the following relations, where attribute CITY has been added to relations EMP (renamed ET), PROJ (renamed PT) and ASG (renamed AT) of the engineering database. Attribute CITY of AT corresponds to the city where the employee identified by ENO lives.

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

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

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

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

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

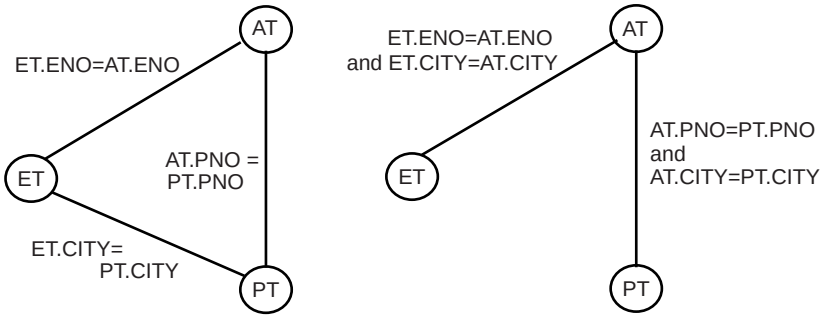


Fig. 8.13 Transformation of Cyclic Query

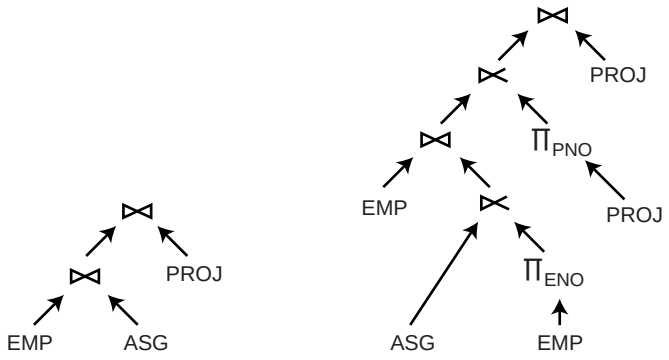
predicate is preserved by transitivity [Kambayashi et al., 1982]. In the example of Figure 8.13b, where the arc (ET, PT) is removed, the additional predicate ET.CITY = AT.CITY and AT.CITY = PT.CITY imply ET.CITY = PT.CITY by transitivity. Thus the acyclic query is equivalent to the cyclic query.

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

8.3.3 Join versus Semijoin

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

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



(a) Join approach (b) Semijoin approach

Fig. 8.14 Join versus Semijoin Approaches

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

8.4 Distributed Query Optimization

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

8.4.1 Dynamic Approach

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

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

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

Algorithm 8.4: Dynamic*-QOA

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

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

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

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

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

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

Site 1	Site 2
EMP_1	EMP_2
ASG	PROJ

There are several possible strategies, including the following:

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

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

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

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

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

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

The rule can be stated as

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

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

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

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

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

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

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

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

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

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

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

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

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

8.4.2 Static Approach

We illustrate the static approach with the algorithm of R* [Selinger and Adiba, 1980; Lohman et al., 1985] which is a substantial extension of the techniques we described in Section 8.2.2). This algorithm performs an exhaustive search of all alternative

strategies in order to choose the one with the least cost. Although predicting and enumerating these strategies may be costly, the overhead of exhaustive search is rapidly amortized if the query is executed frequently. Query compilation is a distributed task, coordinated by a *master site*, where the query is initiated. The optimizer of the master site makes all intersite decisions, such as the selection of the execution sites and the fragments as well as the method for transferring data. The *apprentice sites*, which are the other sites that have relations involved in the query, make the remaining local decisions (such as the ordering of joins at a site) and generate local access plans for the query. The objective function of the optimizer is the general total time function, including local processing and communications costs (see Section 8.1.1).

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

Algorithm 8.5: Static*-QOA

```
Input: QT: query tree
Output: strat: minimum cost strategy
begin
    for each relation R_i ∈ QT do
        for each access path AP_ij to R_i do
            compute cost(AP_ij)
        best_AP_i ← AP_ij with minimum cost
    for each order (R_i1, R_i2, ··· , R_in) with i = 1, ··· , n! do
        build strategy (...((best AP_i1 ⋈ R_i2) ⋈ R_i3) ⋈ ... ⋈ R_in) ;
        compute the cost of strategy
    strat ← strategy with minimum cost ;
    for each site k storing a relation involved in QT do
        LS_k ← local strategy (strategy, k) ;
        send (LS_k, site k)          {each local strategy is optimized at site k}
end
```

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

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

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

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

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

Strategy 1.

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

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

Strategy 2.

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