Chapter 6

SEMANTIC DATA CONTROL

An important requirement of a centralized or a distributed DBMS is the ability to support semantic data control. Semantic data control typically includes view management, security control, and semantic integrity control. Informally, these functions must ensure that *authorized* users perform *correct* operations on the database, contributing to the maintenance of database integrity. The functions necessary for maintaining the physical integrity of the database in the presence of concurrent accesses and failures are studied separately in Chapters 10 through 12. In the relational framework, semantic data control can be achieved in a uniform fashion. Views, security constraints, and semantic integrity constraints can be defined as rules that the system automatically enforces. The violation of some such rule by a user program (a set of database operations) generally implies the rejection of the effects of that program.

The definition of the rules for controlling data manipulation is part of the administration of the database, a function generally performed by a database administrator (DBA). This person is also in charge of applying the organizational policies. Well-known solutions for semantic data control have been proposed for centralized DBMSs. In this chapter we briefly review the centralized solution to semantic data control, and present the problems incurred in a distributed environment and solutions to these problems. The cost of enforcing semantic data control, which is high in terms of resource utilization in a centralized DBMS, can be prohibitive in a distributed environment.

Since the rules for semantic data control must be stored in a catalog, the management of a distributed directory (also called a catalog) is also relevant in this chapter. We discussed directories in Chapter 4. Remember that the directory of a distributed DBMS is itself a distributed database. There are several ways to store semantic data control definitions, according to the way the directory is managed. Directory information can be stored differently according to its type; in other words, some information might be fully duplicated whereas other information might be distributed. For example, information that is useful at compile time, such as security control information, could be duplicated. In this chapter we emphasize the impact of directory management on the performance of semantic data control mechanisms.

This chapter is organized as follows. View management is the subject of Section 6.1. Security control is presented in Section 6.2. Finally, semantic integrity control is treated in Section 6.3. For each section we first outline the solution in a centralized DBMS and then give the distributed solution, which is often an extension of the centralized one, although more difficult.

6.1 VIEW MANAGEMENT

One of the main advantages of the relational model is that it provides full logical data independence. As introduced in Chapter 4, external schemas enable user groups to have their particular *view* of the database. In a relational system, a view is a *virtual relation*, defined as the result of a query on *base relations* (or real relations), but not materialized like a base relation, which is stored in the database. A view is a dynamic window in the sense that it reflects all updates to the database. An external schema can be defined as a set of views and/or base relations. Besides their use in external schemas, views are useful for ensuring data security in a simple way. By selecting a subset of the database, views *hide* some data. If users may only access the database through views, they cannot see or manipulate the hidden data, which are therefore secure.

In the remainder of this section we look at view management in centralized and distributed systems as well as the problems of updating views. Note that in a distributed DBMS, a view can be derived from distributed relations, and the access to a view requires the execution of the distributed query corresponding to the view definition. An important issue in a distributed DBMS is to make view materialization efficient. We will see how the concept of snapshots helps to solve this problem, but first let us concentrate on centralized DBMSs.

6.1.1 Views in Centralized DBMSs

Most relational DBMSs use a view mechanism similar to those of INGRES [Stonebraker, 1975] and System *R* [Chamberlin et al., 1975]. In this context a view is a relation derived from base relations as the result of a relational query. It is defined by associating the name of the view with the retrieval query that specifies it.

Example 6.1

The view of system analysts (SYSAN) derived from relation EMP (ENO,ENAME,TITLE), can be defined by the following SQL query:

```
CREATE   VIEW     SYSAN(ENO, ENAME)
AS       SELECT   ENO, ENAME
         FROM     EMP
         WHERE    TITLE = "Syst. Anal."
```

The single effect of this statement is the storage of the view definition in the catalog. No other information needs to be recorded. Therefore, the result to the

SYSAN

ENO	ENAME
E2	M. Smith
E5	B. Casey
E8	J. Jones

Figure 6.1. Relation Corresponding to the View SYSAN

query defining the view (i.e., a relation having the attributes ENO and ENAME for the system analysts as shown in Figure 6.1) is *not* produced. However, the view SYSAN can be manipulated as a base relation.

Example 6.2

The query

"Find the names of all the system analysts with their project number and responsibility(ies)"

involving the view SYSAN and relation ASG(ENO,PNO,RESP,DUR) can be expressed as

```
SELECT   ENAME,  PNO,  RESP
FROM     SYSAN,  ASG
WHERE    SYSAN.ENO  =  ASG.ENO
```

Mapping a query expressed on views into a query expressed on base relations can be done by *query modification* [Stonebraker, 1975]. With this technique the variables are changed to range on base relations and the query qualification is merged (ANDed) with the view qualification.

Example 6.3

The preceding query can be modified to

```
SELECT   ENAME,  PNO,  RESP
FROM     EMP,  ASG
WHERE    EMP.ENO  =  ASG.ENO
AND      TITLE-  "Syst.  Anal."
```

ENAME	PNO	RESP
M. Smith	P1	Analyst
M. Smith	P2	Analyst
B. Casey	P3	Manager
J. Jones	P4	Manager

Figure 6.2. Result of Query involving View SYSAN

The result of this query is illustrated in Figure 6.2.

The modified query is expressed on base relations and can therefore be processed by the query processor. It is important to note that view processing can be done at compile time. The view mechanism can also be used for refining the access controls to include subsets of objects. To specify any user from whom one wants to hide data, the key word USER generally refers to the logged-on user identifier.

Example 6.4

The view ESAME restricts the access by any user to those employees having the same title:

```
CREATE   VIEW    ESAME
AS       SELECT  *
         FROM    BMP  E1,  EMP  E2
         WHERE   E1.TITLE  =  E2.TITLE
         AND     E1.END  =  USER
```

In the view definition above, * stands for "all attributes" and the two tuple variables (E1 and E2) ranging over relation EMP are required to express the join of one tuple of EMP (the one corresponding to the logged-on user) with all tuples of EMP based on the same title. For example, the following query issued by the user PROJ. Doe,

```
SELECT   *
FROM     ESAME
```

returns the relation of Figure 6.3. Note that the user PROJ. Doe also appears in the result. If the user who creates ESAME is an electrical engineer, as in this case, the view represents the set of all electrical engineers.

6.1.2 Updates through Views

Views can be defined using arbitrarily complex relational queries involving selection, projection, join, aggregate functions, and so on. All views can be interrogated as base relations, but not all views can be manipulated as such. Updates through views can be handled automatically only if they can be propagated correctly to the base relations. We can classify views as being updatable and not updatable. A view is updatable only if the updates to the view can be propagated to the base relations without ambiguity. The view SYSAN above is updatable; the insertion, for example, of a new system analyst <201, Smith> will be mapped into the insertion of a new

ENO	ENAME	TITLE
E1	J. Doe	Elect. Eng
E2	L. Chu	Elect. Eng

Figure 6.3. Result of Query on View ESAME

employee <201, Smith, Syst. Anal.>. If attributes other than TITLE were hidden by the view, they would be assigned *null values*. The following view, however, is not updatable:

```
CREATE  VIEW    EG(ENAME, RESP)
AS      SELECT  ENAME, RESP
        FROM    EMP, ASG
        WHERE   EMP.ENO = ASG.ENO
```

The deletion, for example, of the tuple <Smith, Analyst> cannot be propagated, since it is ambiguous. Deletions of Smith in relation EMP or analyst in relation ASG are both meaningful, but the system does not know which is correct.

Current systems are very restrictive about supporting updates through views. Views can be updated only if they are derived from a single relation by selection and projection. This precludes views defined by joins, aggregates, and so on. However, it is theoretically possible to automatically support updates of a larger class of views ([Bancilhon and Spyratos, 1981], [Dayal and Bernstein, 1978], [Keller, 1982]). It is interesting to note that views derived by join are updatable if they include the keys of the base relations.

6.1.3 Views in Distributed DBMSs

The definition of a view is similar in a distributed DBMS and in centralized systems. However, a view in a distributed system may be derived from fragmented relations stored at different sites. When a view is defined, its name and its retrieval query are stored in the catalog.

Since views may be used as base relations by application programs, their definition should be stored in the directory in the same way as the base relation descriptions. Depending on the degree of site autonomy offered by the system [Williams et al., 1982], view definitions can be centralized at one site, partially duplicated, or fully duplicated. In any case, the information associating a view name to its definition site should be duplicated. If the view definition is not present at the site where the query is issued, remote access to the view definition site is necessary.

Views derived from distributed relations may be costly to evaluate. Since in a given organization it is likely that many users access the same views, some proposals have been made to optimize view derivation. We saw in Section 6.1.1 that view derivation is done by merging the view qualification with the query qualification. An alternative solution proposed in [Adiba and Lindsay, 1980] is to avoid view derivation by maintaining actual versions of the views, called *snapshots*. A snapshot represents a particular state of the database and is therefore *static*, meaning that it does not reflect updates to base relations. Snapshots are useful when users are not particularly interested in seeing the most recent version of the database. They are managed as temporary relations in the sense that they do not have access methods other than sequential scanning. Therefore, a query expressed on a snapshot will not exploit indices available on the base relations from which

it is derived. Access through snapshots seems more adequate for queries that have bad selectivity and scan the entire snapshot. In this case a snapshot behaves more like a predefined answer to a query. It is necessary to recalculate snapshots periodically. However, this can be done when the system is idle. In addition, for snapshots derived by selection and projection, only the difference needs to be calculated [Blakeley et al., 1986].

6.2 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 *authorization 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 protection techniques can be found in [Fernandez et al., 1981].

Authorization 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. Authorization control has long been provided by operating systems, and more recently, by distributed operating systems [Tanenbaum and van Renesse, 1985] 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.

Authorization 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.

From solutions to authorization 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 authorization control in centralized systems and then the additional problems and their solutions in distributed systems.

6.2.1 Centralized Authorization Control

Three main actors are involved in authorization control: the *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 (user, 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 (user, operation type, object definition) which specifies that the user has the right to perform an operation of operation type on a object. To control authorizations properly, the DBMS requires users, objects, and rights to be defined.

The introduction of a user (a person or a group of persons) 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 the network or hierarchical DBMS ([Taylor and Frank, 1976] and [Tsichritzis and Lochovsky, 1976]), it is the object type (e.g., record, file). 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 6.1 permits the protection of objects simply by hiding subsets of relations (attributes or tuples) to unauthorized users.

A right expresses a relationship between a user 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 <user(s)>
REVOKE <operation type(s)> FROM <object> TO <user(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. Granting the

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

Figure 6.4. Example of Authorization Matrix

GRANT privilege means that all the rights of the grantor performing the statement are given to the specified users. 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. A complete scheme for handling revocation is proposed in [Fagin, 1978].

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 6.4 gives an example of an authorization matrix where objects are either relations (EMP and ASG) or attributes (ENAME).

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 disadvantages and advantages 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.

6.2.2 Distributed Authorization Control

The additional problems of authorization control in a distributed environment stem from the fact that objects and subjects are distributed. These problems are: remote user authentication, management of distributed authorization rules, as well as handling of views and of user groups.

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 (e.g., from a site that is not part of the distributed DBMS), users must also be identified and authenticated at the accessed site. Two solutions are possible:

1. 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.

2. All sites of the distributed DBMS identify and authenticate themselves similarly 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 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 second 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 accessed 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, in [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 authorization rules are fully replicated at all sites, the enforcement of access rights is similar to that of a centralized system. However, maintaining this replication is quite expensive. The problem is more difficult if site autonomy (with decentralized control) must be maintained. Several solutions to this problem are identified in [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.

In conclusion, full replication of authorization information has two strong advantages: authorization control is much simpler and can be done at compile time. However, the overhead cost incurred for managing the distributed catalog can be significant if there are many sites in the system.

6.3 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. Semantic integrity control ensures database consistency by rejecting update programs which lead to inconsistent database states, or by activating specific actions on the database state, which compensate for the effects of the update programs. 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 which 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 network 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 recent development of database design aid tools call for powerful integrity constraints which 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, formerly suggested in [Florentin, 1974], is to express integrity constraints using assertions of predicate calculus. Thus a set of semantic integrity assertions defines database consistency. This approach allows one to easily declare and modify complex integrity assertions.

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

Various solutions have been investigated to design an integrity subsystem by combining optimization strategies. Their purpose is to (1) limit the number of assertions that need to be enforced, (2) decrease the number of data accesses to enforce a given assertion in the presence of an update program, (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. The content of this section is strongly based on the work on semantic integrity control for centralized systems reported in [Simon and Valduriez, 1984] and [Simon and Valduriez, 1987] and for distributed systems [Simon and Valduriez, 1986]. In addition, the relational integrity subsystem proposed in [Simon and Valduriez, 1987] offers a rich functionality

by supporting a large subset of multivariable, multirelation assertions with aggregates, and general database programs.

6.3.1 Centralized Semantic Integrity Control

A semantic integrity subsystem has two main components: a language for expressing and manipulating integrity assertions, and an enforcement mechanism that performs specific actions to enforce database integrity at updates.

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, defined in [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.

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, precompiled, 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 nonnull attribute, unique key, foreign key, or functional dependency [Fagin and Vardi, 1984]. Examples 6.5 through 6.8 demonstrate predefined constraints.

Example 6.5 *Nonnull Attribute*

Employee number in relation EMP cannot be null.

```
END NOT NULL IN EMP
```

Example 6.6 *Unique Key*

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

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

Example 6.7 *Foreign Key*

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 6.8 *Functional Dependency*

The employee number functionally determines the employee name.

```
ENO IN EMP DETERMINES ENAME
```

Precompiled constraints express preconditions 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., 1979]. Precompiled 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 precompiled constraints are the following:

Example 6.9 *Domain Constraint*

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

```
CHECK ON PROJ (BUDGET ≥ 500000 AND BUDGET ≤ 1000000)
```

Example 6.10 *Domain Constraint on Deletion*

Only the tuples whose budget is 0 may be deleted.

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

Example 6.11 *Transition Constraint*

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 6.12 *Functional Dependency*

The constraint of Example 6.8 may also be expressed as

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

Example 6.13 *Constraint with Aggregate Function*

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

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

Integrity Enforcement

Enforcing semantic integrity consists of rejecting update programs that violate some integrity constraints. A constraint is violated when it becomes false in the new database state produced by the update. A major difficulty in designing an integrity subsystem is finding efficient enforcement algorithms. Two basic methods permit the rejection of inconsistent updates. The first one is based on the *detection* of inconsistencies. The update 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 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 6.14

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 6.9.

```
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 $((\forall x \in R)(Q(x) \Rightarrow \text{update}(x))$, 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 $F(x)) \Rightarrow \text{update}(x))$. Thus x needs to be universally quantified.

Example 6.15

The foreign key assertion of Example 6.7 that can be rewritten as

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

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

The definition of compiled assertions is based on the notion of differential relations. Let u be an update of relation R. R^+ and R^- are *differential relations* of R by u, where R^+ contains the tuples inserted by u into R, and R^- contains the tuples of R deleted by u. If u is an insertion, R^- is empty. If u is a deletion, R^+ is empty. Finally, if u is a modification, relation R after modification is equal to $R^+ \cup (R - R^-)$.

A *compiled assertion* is a triple *(R, T, C)* in which *R* is a relation, *T* is an update type, and C is an assertion ranging over the differential relation(s) involved in an update of type *T*. When an integrity constraint *I* is defined, a set of compiled assertions may be produced for the relations used by *I*. Whenever a relation involved in *I* is updated by a program *u*, the compiled assertions 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 compiled assertions of type *u* need be checked. Second, the cost of enforcing a compiled assertion is less than that of enforcing *I* since differential relations are, in general, much smaller than the base relations.

Compiled assertions 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 relation. Since the compiled assertions are simpler than the original ones, the process that generates them is called *simplification*.

Example 6.16

Consider the modified expression of the foreign key constraint in Example 6.15. The compiled assertions associated with this constraint are

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

where C_1 is

$\forall$ **NEW** $\in$ ASFG$^+$, $\exists$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 compiled assertions is obvious. For instance, a deletion on relation ASG does not incur any assertion checking.

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

Let us now summarize the enforcement algorithm. We recall that an update program 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 compiled assertions. If no tuples are retrieved, the assertion is valid.

Example 6.17

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

$$result \leftarrow \text{retrieve all tuples of PROJ}^- \text{ where } \neg (C_2)$$

Then, if the result is empty, the assertion is verified by the update.

6.3.2 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 site autonomy, meaning that each site processes local queries and performs data control as a centralized DBMS. This assumption simplifies the description of the method. However, the method obviously works with replicated directories. The two main problems of designing an integrity subsystem for a distributed DBMS are the definition and storage of assertions, and the enforcement of these assertions.

Definition of Distributed Integrity Assertions

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

1. *Individual assertions*: single-relation single-variable assertions. They refer only to tuples to be updated independently of the rest of the database. For instance, domain constraint of Example 6.9 is an individual assertion.

2. *Set-oriented assertions*: include single-relation multivariable constraints such as functional dependency (Example 6.8) and multirelation multivariable constraints such as foreign key constraints (Example 6.7).

3. *Assertions involving aggregates*: require special processing because of the cost of evaluating the aggregates. The assertion in Example 6.13 is representative of an assertion of this class.

Individual assertions. The assertion definition is sent to all other sites that contain fragments of the relation involved in the assertion. The assertion 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 assertion predicate with the fragment predicate. An assertion 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 noncompatibility is found at one of the sites, the assertion definition is globally rejected because tuples of that fragment do not satisfy the integrity constraints. Second, if predicate compatibility has been found, the assertion is tested against the instance of the fragment. If it is not satisfied by that instance, the assertion is also globally rejected. If compatibility is found, the assertion is stored at each site. Note that the compatibility checks are performed only for compiled assertions whose update type is "insert" (the tuples in the fragments are considered "inserted").

Example 6.18

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

$$P_1: \ 0 \leq \text{ENO} < \text{"E3"}$$
$$P_2: \ \text{"E3"} \leq \text{ENO} \ £ \ \text{"E6"}$$
$$p_3: \ \text{ENO} > \text{"E6"}$$

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

Set-oriented assertions. Set-oriented assertions are multivariable; that is, they involve join predicates. Although the assertion predicate may be multirelation, a compiled assertion is associated with a single relation. Therefore, the assertion 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 assertion 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 assertion 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 5) 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 with 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 assertion is stored at each site.

Example 6.19

Consider the set-oriented compiled assertion (ASG, **INSERT**, C_1) defined in Example 6.16, where C_1 is

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

Let us consider the three following 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

$$p_1 : \text{PNO} < \text{“J3”}$$
$$p_2 : \text{PNO} \geq \text{“J3”}$$

In this case each tuple NEW of ASG is compared with either fragment PROJi, if NEW.PNO < “J3”, or fragment PROJ_2 if NEW.PNO ≥ “J3”.

3. PROJ is horizontally fragmented based on the two predicates

$$p_1 : \text{PNAME} = \text{“CAD/CAM”}$$
$$p_2 : \text{PNAME} \neq \text{“CAD/CAM”}$$

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

Enforcement of Distributed Integrity Assertions

Enforcing distributed integrity assertions is more complex than needed in centralized DBMSs. The main problem is to decide where (at which site) to enforce the integrity assertions. The choice depends on the class of the assertion, 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 assertions. 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 assertions. Two cases are considered. If the update is an insert statement, all the tuples to be inserted are explicitly provided by the user. In this case, all individual assertions 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 assertions. We first study single-relation constraints by means of an example. Consider the functional dependency of Example 6.8. The compiled assertion associated with update type INSERT is

$$(\text{EMP}, \textbf{INSERT}, C)$$

where C is

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

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

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

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 assertions. 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 assertion 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 $[f(a_1) \wedge f(02) \wedge \cdots \wedge f(a_n)]$.

Thus the site where the update is submitted must receive for each site a message indicating that this assertion is satisfied and that it is a condition for all sites. If the "assertion is not true for one site, this site sends an error message indicating that the assertion has been violated. The update is then invalid, and it is the responsibility of the integrity subsystem to decide if the entire program must be rejected.

Let us now consider multirelation assertions. For the sake of clarity, we assume that the integrity assertions 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 assertions, 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 6.1.

Algorithm 6.1 *ENFORCE*

input: *T:* update type; *R:* relation
begin
  retrieve all compiled assertions (R, T, C_i)
  inconsistency ← **false**
  for each compiled assertion do
begin
    result ← retrieve all new (respectively old), tuples of R where $\neg (C_i)$
    if *card(result)* $\neq 0$ then
    begin
      inconsistency ← **true**
      exit
    end-if
end-for
if not (*inconsistency*) **then**
    send the tuples to update to all the sites storing fragments of R
else reject the update
end-if
end. {ENFORCE}

Example 6.20

We illustrate this algorithm through an example based on the foreign key assertion of Example 6.7. Let u be an insertion of a new tuple into ASG. The previous algorithm uses the compiled assertion (ASG, **INSERT**, C), where C is

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

For this assertion, 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 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. The rejection of the program depends on the strategy chosen by the program manager of the distributed DBMS.

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

REVIEW QUESTIONS

6.1 Explain views in centralized DBMSs with examples.
6.2 What do you mean by data security?
6.3 Explain centralized semantic integrity control with examples.
6.4 What do you mean by distributed semantic integrity control? Explain with examples.
6.5 What do you mean by set-oriented assertions?