Chapter 5
Data and Access Control

An important requirement of a centralized or a distributed DBMS is the ability to support semantic data control, i.e., data and access control using high-level semantics. 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 context of transaction management. 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 rules by a user program (a set of database operations) generally implies the rejection of the effects of that program (e.g., undoing its updates) or propagating some effects (e.g., updating related data) to preserve the database integrity.

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 special problems encountered 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 Section 3.5. 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 replicated whereas other information might be distributed. For example, information that is useful at compile time, such as security control

M.T. Özsu and P. Valduriez, *Principles of Distributed Database Systems: Third Edition*,
DOI 10.1007/978-1-4419-8834-8_5, © Springer Science+Business Media, LLC 2011

information, could be replicated. 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 5.1. Security control is presented in Section 5.2. Finally, semantic integrity control is treated in Section 5.3. 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.

5.1 View Management

One of the main advantages of the relational model is that it provides full logical data independence. As introduced in Chapter 1, 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 materialized views helps in solving this problem, among others, but requires efficient techniques for materialized view maintenance.

5.1.1 Views in Centralized DBMSs

Most relational DBMSs use a view mechanism where a view is a relation derived from base relations as the result of a relational query (this was first proposed within the INGRES [Stonebraker, 1975] and System R [Chamberlin et al., 1975] projects). It is defined by associating the name of the view with the retrieval query that specifies it.

Example 5.1. The view of system analysts (SYSAN) derived from relation EMP (ENO,ENAME,TITLE), can be defined by the following SQL query:

SYSAN

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

Fig. 5.1 Relation Corresponding to the View SYSAN

```
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 of the query defining the view (i.e., a relation having the attributes ENO and ENAME for the system analysts as shown in Figure 5.1) is *not* produced. However, the view SYSAN can be manipulated as a base relation.

Example 5.2. 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 5.3. The preceding query can be modified to

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

The result of this query is illustrated in Figure 5.2. ♦

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 keyword USER generally refers to the logged-on user identifier.

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

Fig. 5.2 Result of Query involving View SYSAN

Example 5.4. The view ESAME restricts the access by any user to those employees having the same title:

```
CREATE VIEW   ESAME
AS     SELECT *
       FROM   EMP E1, EMP E2
       WHERE  E1.TITLE = E2.TITLE
       AND    E1.ENO = 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 J. Doe,

```
SELECT *
FROM   ESAME
```

returns the relation of Figure 5.3. Note that the user J. 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. ♦

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

Fig. 5.3 Result of Query on View ESAME

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 employee ⟨201, Smith, Syst. Anal.⟩. If attributes other than TITLE were hidden by the view, they would be assigned *null values*.

Example 5.5. The following view, however, is not updatable:

```
CREATE VIEW   EG(ENAME, RESP)
AS     SELECT DISTINCT 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.

5.1.2 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.

The mapping of a query expressed on views into a query expressed on base relations (which can potentially be fragmented) can also be done in the same way as

in centralized systems, that is, through query modification. With this technique, the qualification defining the view is found in the distributed database catalog and then merged with the query to provide a query on base relations. Such a modified query is a *distributed query*, which can be processed by the distributed query processor (see Chapter 6). The query processor maps the distributed query into a query on physical fragments.

In Chapter 3 we presented alternative ways of fragmenting base relations. The definition of fragmentation is, in fact, very similar to the definition of particular views. It is possible to manage views and fragments using a unified mechanism [Adiba, 1981]. This is based on the observation that views in a distributed DBMS can be defined with rules similar to fragment definition rules. Furthermore, replicated data can be handled in the same way. The value of such a unified mechanism is to facilitate distributed database administration. The objects manipulated by the database administrator can be seen as a hierarchy where the leaves are the fragments from which relations and views can be derived. Therefore, the DBA may increase locality of reference by making views in one-to-one correspondence with fragments. For example, it is possible to implement the view SYSAN illustrated in Example 5.1 by a fragment at a given site, provided that most users accessing the view SYSAN are at the same site.

Evaluating views derived from distributed relations may be costly. In a given organization it is likely that many users access the same view which must be recomputed for each user. We saw in Section 5.1.1 that view derivation is done by merging the view qualification with the query qualification. An alternative solution is to avoid view derivation by maintaining actual versions of the views, called *materialized views*. A *materialized view* stores the tuples of a view in a database relation, like the other database tuples, possibly with indices. Thus, access to a materialized view is much faster than deriving the view, in particular, in a distributed DBMS where base relations can be remote. Introduced in the early 1980s [Adiba and Lindsay, 1980], materialized views have since gained much interest in the context of data warehousing to speed up On Line Analytical Processing (OLAP) applications [Gupta and Mumick, 1999c]. Materialized views in data warehouses typically involve aggregate (such as SUM and COUNT) and grouping (GROUP BY) operators because they provide compact database summaries. Today, all major database products support materialized views.

Example 5.6. The following view over relation PROJ(PNO,PNAME,BUDGET,LOC) gives, for each location, the number of projects and the total budget.

```
CREATE VIEW     PL(LOC, NBPROJ, TBUDGET)
AS     SELECT   LOC, COUNT(*),SUM(BUDGET)
       FROM     PROJ
       GROUP BY LOC
```

♦

5.1.3 Maintenance of Materialized Views

A materialized view is a copy of some base data and thus must be kept consistent with that base data which may be updated. *View maintenance* is the process of updating (or refreshing) a materialized view to reflect the changes made to the base data. The issues related to view materialization are somewhat similar to those of database replication which we will address in Chapter 13. However, a major difference is that materialized view expressions, in particular, for data warehousing, are typically more complex than replica definitions and may include join, group by and aggregate operators. Another major difference is that database replication is concerned with more general replication configurations, e.g., with multiple copies of the same base data at multiple sites.

A view maintenance policy allows a DBA to specify *when* and *how* a view should be refreshed. The first question (when to refresh) is related to consistency (between the view and the base data) and efficiency. A view can be refreshed in two modes: *immediate* or *deferred*. With the immediate mode, a view is refreshed immediately as part as the transaction that updates base data used by the view. If the view and the base data are managed by different DBMSs, possibly at different sites, this requires the use of a distributed transaction, for instance, using the two-phase commit (2PC) protocol (see Chapter 12). The main advantages of immediate refreshment are that the view is always consistent with the base data and that read-only queries can be fast. However, this is at the expense of increased transaction time to update both the base data and the views within the same transactions. Furthermore, using distributed transactions may be difficult.

In practice, the deferred mode is preferred because the view is refreshed in separate (refresh) transactions, thus without performance penalty on the transactions that update the base data. The refresh transactions can be triggered at different times: *lazily*, just before a query is evaluated on the view; *periodically*, at predefined times, e.g., every day; or *forcedly*, after a predefined number of updates to the base data. Lazy refreshment enables queries to see the latest consistent state of the base data but at the expense of increased query time to include the refreshment of the view. Periodic and forced refreshment allow queries to see views whose state is not consistent with the latest state of the base data. The views managed with these strategies are also called *snapshots* [Adiba, 1981; Blakeley et al., 1986].

The second question (how to refresh a view) is an important efficiency issue. The simplest way to refresh a view is to recompute it from scratch using the base data. In some cases, this may be the most efficient strategy, e.g., if a large subset of the base data has been changed. However, there are many cases where only a small subset of view needs to be changed. In these cases, a better strategy is to compute the view *incrementally*, by computing only the changes to the view. Incremental view maintenance relies on the concept of differential relation. 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 can be obtained by computing $(R - R^-) \cup R^+$. Similarly, a materialized

view V can be refreshed by computing $(V - V^-) \cup V^+$. Computing the changes to the view, i.e., V^+ and V^-, may require using the base relations in addition to differential relations.

Example 5.7. 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). Let EMP^+ consist of one tuple ⟨E9, B. Martin, Programmer⟩ to be inserted in EMP, and ASG^+ consist of two tuples ⟨E4, P3, Programmer, 12⟩ and ⟨E9, P3, Programmer, 12⟩ to be inserted in ASG. The changes to the view EG can be computed as:

```
EG+ =   (SELECT  ENAME, RESP
         FROM    EMP, ASG+
         WHERE   EMP.ENO = ASG+.ENO)
                 UNION
         (SELECT  ENAME, RESP
         FROM    EMP+, ASG
         WHERE   EMP+.ENO = ASG.ENO)
                 UNION
         (SELECT  ENAME, RESP
         FROM    EMP+, ASG+
         WHERE   EMP+.ENO = ASG+.ENO)
```

which yields tuples ⟨B. Martin, Programmer⟩ and ⟨J. Miller, Programmer⟩. Note that integrity constraints would be useful here to avoid useless work (see Section 5.3.2). Assuming that relations EMP and ASG are related by a referential constraint that says that ENO in ASG must exist in EMP, the second SELECT statement is useless as it produces an empty relation. ♦

EG

ENAME	RESP
J. Doe	Manager
M. Smith	Analyst
A. Lee	Consultant
A. Lee	Engineer
J. Miller	Programmer
B. Casey	Manager
L. Chu	Manager
R. Davis	Engineer
J.Jones	Manager

Fig. 5.4 State of View EG

Efficient techniques have been devised to perform incremental view maintenance using both the materialized views and the base relations. The techniques essentially differ in their views' expressiveness, their use of integrity constraints, and the way they handle insertion and deletion. Gupta and Mumick [1999a] classify

these techniques along the view expressiveness dimension as non-recursive views, views involving outerjoins, and recursive views. For non-recursive views, i.e., select-project-join (SPJ) views that may have duplicate elimination, union and aggregation, an elegant solution is the counting algorithm [Gupta et al., 1993]. One problem stems from the fact that individual tuples in the view may be derived from several tuples in the base relations, thus making deletion in the view difficult. The basic idea of the counting algorithm is to maintain a count of the number of derivations for each tuple in the view, and to increment (resp. decrement) tuple counts based on insertions (resp. deletions); a tuple in the view of which count is zero can then be deleted.

Example 5.8. Consider the view EG in Figure 5.4. Each tuple in EG has one derivation (i.e., a count of 1) except tuple ⟨M. Smith, Analyst⟩ which has two (i.e., a count of 2). Assume now that tuples ⟨E2, P1, Analyst, 24⟩ and ⟨E3, P3, Consultant, 10⟩ are deleted from ASG. Then only tuple ⟨A. Lee, Consultant⟩ needs to be deleted from EG. ♦

We now present the basic counting algorithm for refreshing a view V defined over two relations R and S as a query $q(R,S)$. Assuming that each tuple in V has an associated derivation count, the algorithm has three main steps (see Algorithm 5.1). First, it applies the view differentiation technique to formulate the differential views V^+ and V^- as queries over the view, the base relations, and the differential relations. Second, it computes V^+ and V^- and their tuple counts. Third, it applies the changes V^+ and V^- in V by adding positive counts and subtracting negative counts, and deleting tuples with a count of zero.

Algorithm 5.1: COUNTING Algorithm

Input: V: view defined as $q(R,S)$; R, S: relations; R^+, R^-: changes to R
begin
  $V^+ = q^+(V,\ R^+,\ R,\ S)$;
  $V^- = q^-(V,\ R^-,\ R,\ S)$;
  compute V^+ with positive counts for inserted tuples;
  compute V^- with negative counts for deleted tuples;
  compute $(V - V^-) \cup V^+$ by adding positive counts and substracting
  negative counts deleting each tuple in V with count = 0;
end

The counting algorithm is optimal since it computes exactly the view tuples that are inserted or deleted. However, it requires access to the base relations. This implies that the base relations be maintained (possibly as replicas) at the sites of the materialized view. To avoid accessing the base relations so the view can be stored at a different site, the view should be maintainable using only the view and the differential relations. Such views are called *self-maintainable* [Gupta et al., 1996].

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 : $0 \leq$ ENO $<$ "E3"

p_2 : "E3" $\leq$ ENO $\leq$ "E6"

p_3 : ENO $>$ "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

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

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

$PROJ_1 = \sigma_{PNAME = \text{“CAD”}}(PROJ)$
$PROJ_2 = \sigma_{PNAME \neq \text{“CAD”}}(PROJ)$

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.