Corso BASI DI DATI E LABORATORIO
Prof. CASTANO
Dispense su sicurezza
Copia da lasciare in copisteria

[illegible]

DATABASE SECURITY

Silvana Castano
Dipartimento di Scienze dell'Informazione
Università di Milano

Maria Grazia Fugini
Dipartimento di Informatica e Sistemistica
Università di Pavia

Giancarlo Martella
Dipartimento di Scienze dell'Informazione
Università di Milano

Pierangela Samarati
Dipartimento di Scienze dell'Informazione
Università di Milano

acm PRESS

ADDISON-WESLEY

Harlow, England • Reading, Massachusetts • Menlo Park, California • New York
Don Mills, Ontario • Amsterdam • Bonn • Sydney • Singapore
Tokyo • Madrid • San Juan • Milan • Mexico City • Seoul • Taipei

1 Information security

1.1 Introduction

The increasing development of information technology in the past few years has led to the widespread use of computer systems in various public and private organizations, such as banks, universities, manufacturing or service companies, hospitals, libraries, central or distributed administration and so on. The increased reliability now offered in hardware and software technologies, coupled with the continuous reduction of costs, the increasing professional expertise of information specialists and the availability of support tools, have all contributed to encourage the widespread use of computing services.

This has meant that more data than ever before is now stored and managed by computer systems, or rather by the tools and techniques capable of supporting and meeting these application requirements. Such requirements have been largely satisfied by database technology employing *Database Management Systems (DBMSs)*.

A database (Ullman, 1988; Date, 1990) is a collection of permanent data, managed by the DBMS software. Database design methodologies have been developed to support the different information requirements and operation environments of applications. Conceptual and logical data models have been studied, with the associated languages and tools for data definition, manipulation and querying. The target has been the production of DBMSs able to access efficiently application-defined subsets of data in a database.

A further basic feature of a DBMS is its ability to manage transactions simultaneously on behalf of concurrent applications; thus, each application has the view of a virtually dedicated database. This feature turns out to be relevant when considering, for example, a banking database and its several on-line users, or an airline or railway database and its related booking activities.

Distributed processing has also contributed to advances in the development and automation of information systems. Today, the processing units of an organization and its remote branches can communicate rapidly with each other via computer networks, personal computers and workstations, thus allowing a rapid transfer of a large amount of data.

Although the increasingly widespread use of both centralized and distributed databases has proved necessary to support business functions, it has also posed serious problems of data security (Denning and Denning, 1979; Parker, 1984). In fact, damage in a database environment does not only affect a single user or application but rather the whole information system: hence, consequences are *a priori* unpredictable (Dobkin *et al.*, 1979). Advances in information processing techniques (tools and languages) aimed at a simplification of human/machine interfaces have served to make databases available to different types of user; consequently, more serious security problems arise (Denning, 1990). Therefore, in computer-based information systems, technologies, tools and procedures concerning security are essential both to assure system continuity and reliability and to protect data and programs from intrusions, modifications, theft and unauthorized disclosure (Saltzer and Schroeder, 1975; Summers, 1984).

The complexity of the design and implementation of a secure information system depends on the number of factors to be taken into account, such as the heterogeneity of system users, the granularity and territorial extension of information systems (both at national and international levels), the uncontrollable and unpredictable consequences of loss of information, and the difficulties in modelling, specifying and verifying data security (Fernandez, *et al.*, 1981; Denning, 1982; Perry, 1982; Pfleeger, 1989; Hruska and Jackson, 1990; McLean, 1990).

1.1.1 Database security

Information security in a database includes three main aspects: *secrecy, integrity, and availability* (Russell and Gangemi, 1991). Note that in this book, the terms authorization, protection and security will be used interchangeably. More precisely, authorization is used in database systems, protection is typical of operating systems, and security is the most general term. We use the same term employed by the literature regarding each model or system considered.

Ensuring **secrecy** means preventing/detecting/deterring the improper disclosure of information. In general, secrecy properly refers to protection of data involved in highly protected environments, such as military environments or departments of commercial environments. *Privacy* refers to information about individuals, and is sometimes defined as 'the right of an individual, group or institution to determine when, how and for what purpose information concerning himself/itself can be collected, stored and released to other people or entities'. Therefore, privacy refers to environments where data about people or legal individuals is maintained; privacy is ensured by laws and rules in many countries. *Secrecy* is a most relevant aspect of security-critical environments. For example, the target coordinates of a missile should not be improperly disclosed. In commercial environments, secrecy of information is strictly coupled with the application environment and with the internal policies or market strategies and regulations of the organization (Sterne *et al.*, 1991). Consequently, in such environments, secrecy will be ensured only for a portion of data that is defined as *critical* for the organization. As another example, a general policy of commercial environments is that employees should not come to know the salaries of their managers.

Ensuring **integrity** of information means preventing/detecting/deterring the improper modification of information (Sandhu and Jajodia, 1990). For example, in a military environment, the target coordinates of a missile should not be improperly modified. Also, in commercial environments, data integrity is a relevant aspect: the good working of an organization depends on correct operations on correct and coherent data. For example, an employee should not be able to modify his or her own salary, or improperly alter data regarding an electronic payment.

Ensuring **system availability** (that is, avoiding *denial of service*) means preventing/detecting/deterring improper denial of access to services provided by the system (Courtney, 1990). For example, in a commercial environment, payment orders regarding taxes should be made on time as fixed by law. Analogously, in a military environment, when the proper command is issued, the missile should fire.

In many environments, such as public institutions, secrecy and integrity are often needed in combination. For example, in hospitals, airline companies or credit institutions, both secrecy and integrity are required since, besides privacy constraints, correctness of data is vital. Wrong data would entail heavy damage ranging from financial loss to loss of human lives.

1.2 A survey of database concepts

A database is a collection of mutually correlated data permanently stored on persistent storage supports. It is used within an organization by different applications, each one having its own aims and purposes. In a database, data (*entities*) and entity associations that characterize an organization are described.

Data management functions, usually supplied by an operating system, are extended through a set of programs globally called the **Database Management System (DBMS)**. Great quantities of data can be accessed rapidly and efficiently by a DBMS which supplies a set of targeted functions, such as schema management, concurrent transaction management, data access control, logging, recovery of the database after system failures. This is useful for both the database and the application management. Management of a *logical model* describing data and associations among data is a further basic feature of a DBMS, which makes high-level languages available for this purpose.

In database design, a *conceptual phase* and a *logical phase* are distinguished, where *conceptual and logical models* are respectively used for describing the structure of the database. Of these **data models**, the logical model is DBMS dependent (for example, relational or hierarchical), while the conceptual model is used for *conceptual design* independent of the particular DBMS to be used. The *Entity–Relationship* model is one of the most popular conceptual models, based on the concepts of *entity*, as a class of real-world objects to be described in the database, and of *relationship*, modelling associations between two or more entities.

During logical design, the conceptual schema is translated into a *logical schema* describing data according to the logical model supported by the selected DBMS. Hierarchical, network and relational models are the logical models managed by traditional DBMS technology.

The **languages** available in a DBMS comprise a *Data Definition Language (DDL)*, a *Data Manipulation Language (DML)*, and a *Query Language (QL)*. The DDL supports the definition of the logical database schema. Operations on data are specified using the DML or the QL. Database operations are concerned with data retrieval, insertion, deletion and update. The DML, which generally requires full knowledge of the logical model and schema (procedural properties), is used by specialized users, such as application developers. Query languages, on the contrary, are declarative languages oriented to support non-practitioner end-users. DML languages can be 'hosted' within an ordinary programming language, called the host language. Thus, applications using programming languages can include DML instructions for data-oriented operations.

1.2.1 Components of a DBMS

The typical architecture of a DBMS is shown in Figure 1.1.

The modules of Figure 1.1 correspond to the following functionalities (Ullman, 1988):

- DDL compilation
- DML language processing
- Database querying
- Database management
- File management.

A set of data supports the functionality of these modules:

- Database description tables
- Authorization tables
- Concurrent access tables.

Sets of data in a database are requested by end users or application programs through DML instructions or via QL instructions. These instructions are then interpreted by the DBMS through the DML and QL processor. The purpose is query optimization according to the database schema, described in a set of database description tables. These are defined through DDL instructions compiled by the DDL compiler. Optimized queries are processed by the database manager and translated into operations on the physical files of data.

The database manager also checks the users' or programs' authorizations to data access by consulting the authorization tables. Authorized operations are

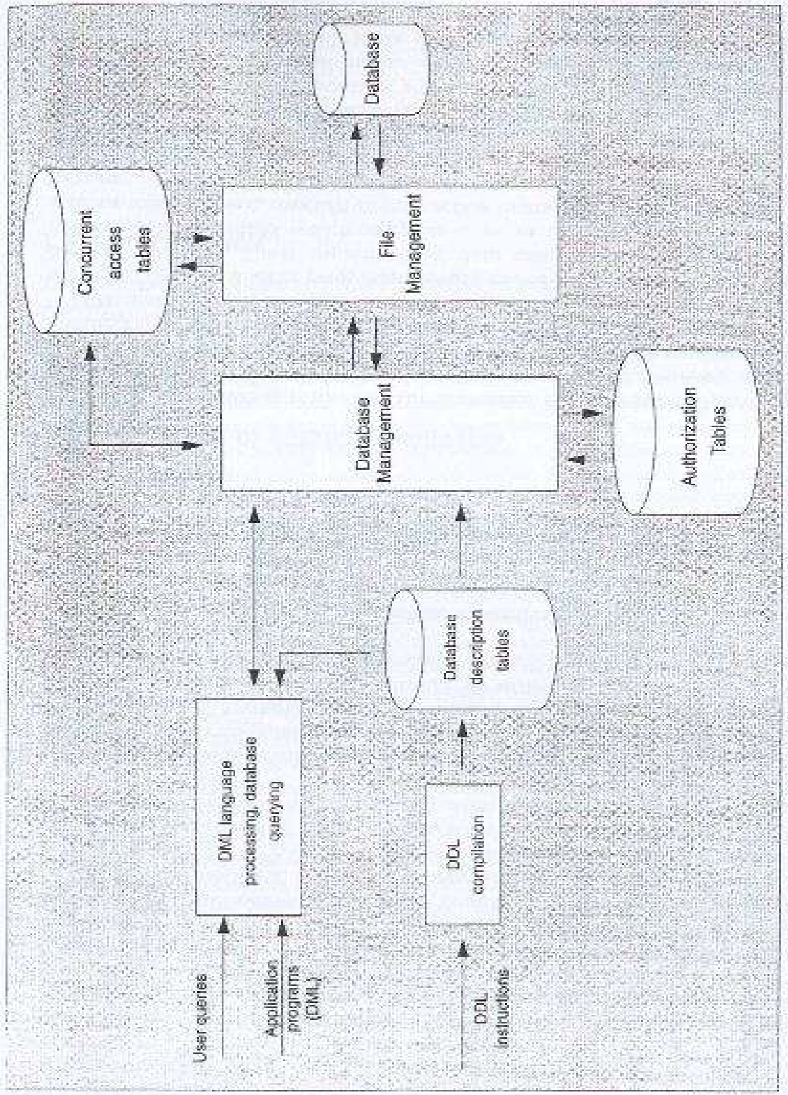

Figure 1.1 DBMS architecture.

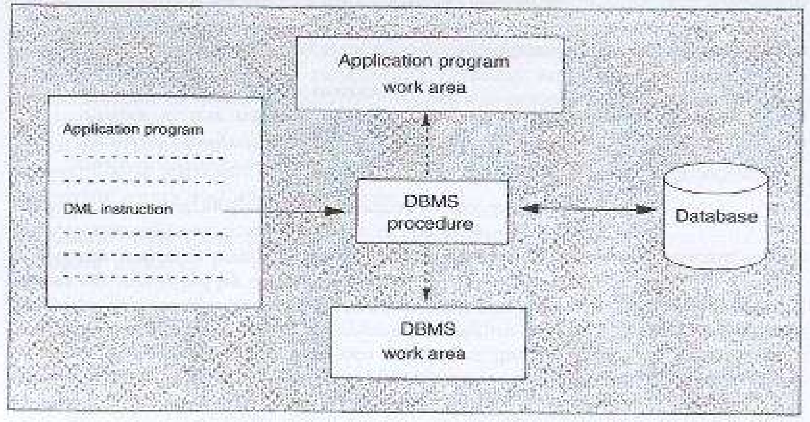

Figure 1.2 Interaction between an application program and a database.

forwarded to the file manager. The database manager is also responsible for concurrent access management: that is, for managing simultaneous access requests by applications to data. A set of system data is maintained for this purpose, such as information about active locks at a certain instant. The file manager executes operations on files.

Figure 1.2 shows the interaction between an application program containing DML statements, and a database. The execution of a DML instruction corresponds to a DBMS procedure accessing the database. The procedure fetches data from the database into the application work area (retrieval instructions), moves data from a work area into the database (insert, update instructions), or deletes data from the database (delete instructions).

1.2.2 Data description levels

Data description levels of a DBMS are shown in Figure 1.3 (Ullman, 1988). Each level provides an abstraction of the database. Referring to Figure 1.3, the following data description levels can be considered in a DBMS:

- *Logical views*
  According to the features of the selected logical model and the purposes of applications, views on the whole logical schema of data are provided, tailored to the various application needs. Logical views are the description of a portion of the database logical schema. Generally available

  are a DDL to define logical views, and a DML to operate on the logical views. The DDL for logical views is often similar to the DDL for logical schemes. DDL instructions are compiled by the DDL compilation module, to obtain the database description tables.
- *Logical data schema*
  At this level, all the data of the database is described using the logical model of the selected DBMS (hierarchical, network, relational). All data and its relationships are described through the DDL of the selected DBMS, and the various operations on the logical schema are specified through the DML of that DBMS.
- *Physical data schema*
  At this level, the storage structure of data within the files in secondary memory is described. Data is physically stored as records (of fixed or variable length) and record pointers.

The different levels allowed by a DBMS in the description of data support the notion of **logical and physical independence of data**.

Logical independence means that a logical schema can be modified without modifying the application programs operating on that schema. In this case modifications of the logical schema correspond to redefining the correspondence between the logical schema and the logical views of the applications on that schema.

Physical independence means that the physical data schema can be modified without modifying the applications accessing that data. Sometimes it also means that the physical structures for data storage can be modified without affecting the description of the logical data schema. This is due to the presence of an intermediate level between the logical schema level and the physical level (in the ANSI/SPARC proposals, this is called 'internal level') that directly interfaces these modifications to the logical schema.

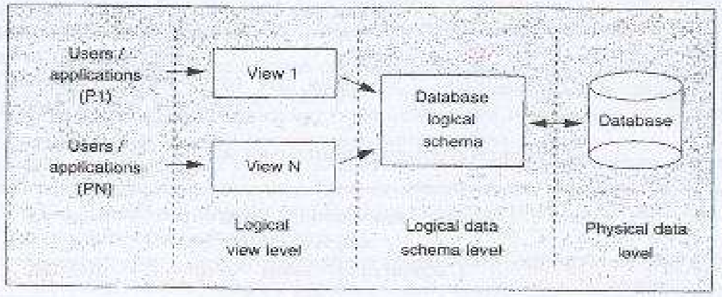

Figure 1.3 Data description levels.

1.2.3 Elements of the relational data model

DBMSs based on the relational model are, nowadays, the most up-to-date commercially used technology for DBMSs. The reason for this is that they rely on a formal model, which provides high independence of the physical structures of data (Atzeni and De Antonellis, 1993). Additionally, the model is flexible in that it allows a variety of operations and queries that are not bound to the underlying physical features, unlike the hierarchical and network models.

The basis of the relational model is the mathematical concept of *relation*, which, according to the set theory, is a subset of the Cartesian product $D_1 \times D_2 \times ... \times D_n$, (where $D_1, D_2, ..., D_n$ is a set of domains). A relation is, therefore, a set of n-tuples $(d_1, d_2,..., d_n)$ such that:

$$d_1 \in D_1, d_2 \in D_2, ..., d_n \in D_n$$

The number n of domains, on which the relation is defined, is named the *degree* of the relation. The number of n-tuples of the relation is the *cardinality* of the relation. Each value d_i in the n-tuple is named the *component*.

A relational database is a collection of tables, where a table corresponds to a relation. The table rows correspond to the n-tuples of the relation. The table columns correspond to the components of the n-tuples. The table columns are often assigned unique names, the *attributes*. Tables rows are all different and univocally identified by the values of one or more attributes called the *key* of the relation.

A *functional dependency* exists between two single-valued attributes A_1 and A_2 of a relation r if and only if to every value of A_1 in r there corresponds only one value of A_2 in r (symbolically denoted by $A_1 \rightarrow A_2$).

The *relational schema* is composed of the relation name and the set of the relation attribute names. The *schema* of a relational database is the set of the relational schemas of all the relations defined for the representation of the entities and the relationships between entities of the database. An example of a relational database schema is reported in Figure 1.4, where the `Student` and `Course` entities and the 'many-to-many' relationship named `Enrolment` among them are modelled by three relations.

At the **instance** level, a table corresponds to each relational schema, and a record corresponds to each n-tuple of the relation.

Relevant constraints of the relational model are the following:

- *entity integrity constraint*, stating that no primary key value can be null;
- *referential integrity constraint*, between two relations, stating that an n-tuple in one relation that refers to another relation must refer to an existing n-tuple in that relation.

Figure 1.5 shows an example of an instance related to the schema shown in Figure 1.4.

As an example, we show a declaration statement in the SQL **data definition language** (Structured Query Language), defined for **relational** DBMSs. The declaration of the `Student` relation of Figure 1.4 has the following form in SQL:

```
Student ( StudNum, Name, Birthdate )
Course ( CourseCode, Teacher )
Enrolment ( StudNum, CourseCode, Grade )
```

Figure 1.4 Sample relational schema.

```
CREATE TABLE Student
(StudNum NUMBER (6.0) NOT NULL,
Name CHAR (20),
Birthdate DATE);
CREATE INDEX FOR Student ON Name
```

StudNum	Name	Birthdate
306318	Smith	05-24-65
316804	Johnson	12-28-64
305700	Bradford	11-3-65
319887	Partridge	11-9-65

Course Code	Teacher
CS 130	Brown
CS 104	Schoen
CS 125	Debson

StudNum	Course Code	Grade
306318	CS 130	A
305700	CS 130	B
319887	CS 130	A
319887	CS 104	C
319887	CS 125	D

Figure 1.5 An example of the schema of Figure 1.4.

This declaration means that a `Student` relation exists, with `StudNum`, `Name`, and `Birthdate` attributes for which the domain values are specified. In addition, a value must be always defined for `StudNum` in the database records, because this attribute is the relation key (primary key). The definition of an index on `Name` is useful for searching the student records more efficiently at the physical level.

As an example, the SQL instructions respectively corresponding to search, insert, delete and data update operations in the database are listed in the following **query and update** example:

```
SELECT Name FROM Student
WHERE StudNum > '316056' AND < '324567'
```

to find all the student names whose `StudNum` is in the given range of values.

```
INSERT INTO Student
VALUES (307240, 'Ferg', '09-25-65')
```

to insert into the `Student` table the new record corresponding to the student `Ferg` with the specified values.

```
DELETE FROM Student
WHERE StudNum = '306318'
```

to delete from the `Student` table a record with the specified key.

```
UPDATE Student
SET Birthdate = '11-28-63'
WHERE StudNum = '319887'
```

to update the `Birthdate` attribute of the record whose `StudNum` value is `319887` to the new value specified in the `SET` clause.

Since a formal basis is available (the concept of 'relation'), *query languages* can be defined for the relational model to express formally operations on the relations. The best known query language of this type is the **relational algebra** which is a procedural language with a set of operators whose operands are relations. All the relational algebra operators are applied on relations and give relations as a result. The main operators on sets of the relational algebra are:

- *Union*: applied to r and s, returns a new relation t defined as the set of the n-tuples belonging to r or to s;
- *Difference* $r - s$: applied to r and s returns a new relation t defined as the set of the n-tuples belonging to r and not to s;
- *Intersection*: applied to r and s returns a new relation t defined as the set of the n-tuples belonging to r and to s;
- *Cartesian product*: applied to r and s, returns a new relation t defined as the set of the n-tuples that are a combination of the n-tuples of r and s.

The basic relational algebra operators specifically defined for relations are:

- *Selection* σ: applied to a relation r, returns a new relation s formed of the n-tuples of r verifying a predicate (expressed through a formula involving constants, and comparison operators). It implements the horizontal decomposition of r;
- *Projection* π: applied to a relation r, returns a relation s containing the n-tuples of r defined on a subset of the attributes of r. It implements the vertical decomposition of r;
- *Natural join*, $r \bowtie s$: let $r(YX)$ and $s(XZ)$ be two relations, such that $YX \cap XZ = X$; the natural join of r and s is a relation t defined on $Y\ X\ Z$ consisting of the set of the n-tuples resulting from the concatenation of n-tuples in r with n-tuples in s that have identical values for the attributes X.

1.3 Security problems in databases

In database environments, the different applications and users of an organization refer to a unique integrated set of data through the DBMS. On the one hand, this solves problems like duplication, data inconsistency, or dependence between the programs and the data structures; on the other hand, security threats become a more serious and important issue in database environments.

Achieving security in a database environment means identifying the threats and choosing the proper *policies* ('what' the security system is expected to do) (Olson and Marshall, 1990) and *mechanisms* ('how' the security system should achieve the security goals) (Bell, 1990). It also involves the provision of *security system assurance* ('how well' the security system meets the protection requirements and executes the expected functions) (Andrews and MacEwen, 1990).

1.3.1 Threats to database security

A *threat* can be defined as a hostile agent that, either casually or by using a specialized technique, can disclose or modify the information managed by a system.

Violations to database security consist of improper readings, modifications or deletions of data. Events that bring violations to databases are called threats (Hinke, 1988). Consequences of violations can be grouped into three categories.

- *Improper release of information* caused by reading of data from intentional or accidental access by improper users. Included in this category are violations to secrecy deriving from authorized observation of data that could be used to infer unauthorized information.
- *Improper modification of data*. This involves all violations to data integrity through improper data handling or modifications. Improper modifications do not necessarily involve unauthorized reading, as data can be tampered with without being read.
- *Denial of service*. This involves those actions that could prevent users from accessing data or using resources.

Security threats can also be classified according to the way they can occur; that is, as *non-fraudulent* (accidental) and *fraudulent* (intentional) threats.

Non-fraudulent threats are casual accidents independent of a determined will to cause damage. These involve:

- *Natural or accidental disasters*, such as earthquakes, water damage or fire. These accidents can damage the system hardware and the stored data; they always cause an integrity violation or denial of service.
- *Errors or bugs in hardware or software*. This may lead to incorrect application of the security policies and, therefore, to unauthorized access, reading or modification of data, or to denial of access to authorized users.
- *Human errors* causing unintentional violations such as incorrect input or incorrect use of applications: the consequences are analogous to those caused by errors or bugs.

Fraudulent or intentional factors denote an explicit and determined fraudulent will to cause damage. Violations involve two classes of user:

(1) *Authorized users* who can abuse their privileges and authority.
(2) *Hostile agents*, namely, improper users (outsiders and insiders) executing actions of vandalism to the software and/or system hardware, or improperly reading or writing data. In both cases 'legal' tasks or 'legal' use of applications can mask the real fraudulent purpose. Typically, viruses, Trojan Horses and trapdoors are attacks of hostile agents. A virus is a code able to copy itself and to damage permanently and often irreparably the environment where it gets reproduced. A Trojan Horse is a program which, under an apparent utility, collects information for its own, possibly fraudulent, use. It can be software installed unwittingly by authorized users which, in addition to the desired functionalities, exploits the user's legitimate privileges to cause a security breach. A trapdoor is a code segment hidden within a program; a special input will start this segment and allow its owner to skip the protection mechanisms and to access the system resources beyond his or her privileges.

1.3.2 Database protection requirements

Protecting a database from possible threats means protecting resources, particularly stored data, from accidental or intentional unauthorized reading and/or updates (Fernandez *et al.*, 1981). Database protection requirements can be summarized as follows.

Protection from improper access

This is a primary problem to which this book devotes major attention. It consists of granting access to a database only to authorized users. Access requests have to be checked by the DBMS against the user's or application's authorizations. Access controls are more complex for databases than for files managed by an operating system. Controls need to apply to objects of a finer

granularity such as records, attributes and values. Additionally, data within a database is semantically related, thus allowing a user to come to know the value of a data item without accessing it directly, but by inferring it from known values.

Protection from inference

Inference denotes the possibility of obtaining confidential information from non-confidential data. In particular, inference problems affect statistical databases where users must be prevented from tracing back to information on individual entities starting from statistical aggregated information. For example, suppose a user first queries the database for the average salary of women employees, and later for the number of women employees. If this last value is 1, the user is enabled to draw (infer) the woman's salary using only statistical queries (average and counting).

Integrity of the database

This requirement concerns database protection from unauthorized access that could modify the contents of data, as well as from errors, viruses, sabotage or *failures* in the system that could damage stored data. This kind of protection is partly carried out by the DBMS through proper system controls, and various backup and recovery procedures, and partly through *ad hoc* security procedures.

Backup and recovery procedures are widely investigated in DBMS literature (Korth and Silberschatz, 1988). Hereafter, we summarize briefly the relevant concepts. In case of failure, the state of the database may no longer be consistent. To preserve consistency, we require that each transaction be atomic. Atomicity means that a transaction can only:

(1) Terminate correctly, modifying the accessed data;
(2) Terminate unsuccessfully without modifying the accessed data.

After a transaction has terminated correctly, the data modifications are made permanent (durability).

The *recovery system* uses a log journal, namely, a file containing a sequence of records stored into stable storage. For each transaction, the log journal records the operations that have been performed on data (read, write, insert, delete) as well as transaction control operations ('begin transaction', 'commit' – correct termination, 'abort' – unsuccessful termination, 'end transaction'), also the old value and the new value of the records involved.

Basically, the recovery system reads the log file to determine the transactions to be undone (i.e., all noncommitted transactions), and the transactions to be redone (i.e., transactions having a 'commit' in the log). To undo a transaction means to copy the old value of each operation in the involved record. To redo a transaction means to copy the new value of each operation in the record.

Ad-hoc security procedures aim to protect data from unauthorized modifications, alterations, insertions and erasures. Modelling, design and enforcement of these procedures is one of the goals of the logical security of a database, as illustrated extensively in this book.

Operational integrity of data

This requirement aims to ensure the logical consistency of data in a database during concurrent transactions. The **concurrency manager** is the DBMS subsystem that fulfils this requirement.

The concurrency manager ensures the *serializability* and *isolation* properties of transactions. Serializability means that the outcome of a concurrent run of a set of transactions is the same as the one produced by a strict sequence of these transactions. *Isolation* means mutual independence among transactions, thus avoiding 'domino effects', where an 'abort transaction' causes other transactions to abort in a cascade.

The problem of ensuring that concurrent access to the same data item by different transactions does not lead to data inconsistency is commonly solved through **locking** techniques.

Lock and unlock techniques consist, respectively, in blocking data items for the time needed to execute an operation, and in releasing the items once the operation has been completed. In this way, a transaction can lock a data item, making it inaccessible to other transactions. The item is accessible again at release time.

However, it is not possible to ensure serializability sufficiently by locking techniques alone: the *two-phases locking* protocol overcomes this problem. According to this protocol, lock and unlock operations must be executed to prevent a transaction from blocking other resources (that is, demanding further blocks) after releasing some item. This assures transaction serializability through a growing phase for lock acquisition and a decreasing phase for lock releasing. According to a second principle of this protocol, a transaction is prevented from releasing resources (that is, a lock) before commit, abort or writing operations have been executed for this transaction. Owing to this principle, commit or abort operations are executed by a transaction only when all the required resources are actually available.

Semantic integrity of data

The problem is to ensure the logical consistency of modified data by controlling data values in the allowed range. Restrictions on data values (for example, attribute values in a relational database) are expressed as *integrity constraints*.

Constraints can be defined for the whole database (conditions defining the correct state of a database), or for transitions (conditions to be verified in order to execute a modification to the database).

Accountability and auditing

This requirement consists of the possibility of recording all accesses to data, for both 'read' and 'write' operations (Bonium, 1988). Auditing and accountability are useful deterrent tools for data physical integrity, as well as for subsequent analysis of access sequences to a database. The granularity level of the registered operations could be a problem for a database; it might be useful to record operations involving the single value inside a record, but this is impractical from the viewpoint of time and cost.

User authentication

This requirement concerns the necessity of identifying uniquely the database users. User identification is the basis of every authorization mechanism. Users are allowed access to data when identified as 'authorized' users by the system.

Management and protection of sensitive data

Databases may contain *sensitive data* that should not to be made public; some databases contain only sensitive data (for example, military databases), while others are completely public (for example, library databases). Databases containing mixed data, that is, both sensitive and ordinary data, exhibit more complex protection problems. A data item is sensitive in various circumstances: by itself; when combined with other data; for being contained in records that are declared as sensitive or for being declared sensitive by the *Database Administrator (DBA)*.

Access control for databases containing mixed data consists primarily in protecting the confidentiality of sensitive data, allowing access only to authorized users (generally a small portion of the user population); these users are granted a set of operations on sensitive data and are prevented from propagating their privileges.

In addition, access control allows users who are authorized for sensitive data to work on non-sensitive data as well, together with other users, with no interferences.

Finally, situations may arise where authorized users can access a given set of sensitive data separately, although they cannot access all data concurrently.

Multilevel protection

'Multilevel protection' means a set of protection requirements. Information may need to be classified at various levels of protection: for example, in military databases, where a finer classification is needed than simply 'sensitive' and 'ordinary' data. In these environments, sensitive levels can be different even among items of the same record, or values of the same attribute. In this sense, multilevel protection is intended both as the assignment of classification levels to different information items and as the assignment of different accesses to single items according to their classification.

A further requirement in multilevel protection concerns the possibility of assigning a level to aggregated information to express that sensitivity is higher or lower for aggregated items than for the single items participating in the aggregation. Secrecy and integrity of multilevel information is met by assigning a clearance to users and by allowing a user to access data classified at the user clearance level.

Confinement

Confinement is intended as the necessity to avoid undesired *information transfer* between system programs: for example, transfer of 'critical' data to unauthorized programs. Information transfer occurs along **authorized**

channels, memory channels, and **covert channels.** Authorized channels supply output information via authorized actions: for example, editing or compiling a file. Memory channels are memory areas where data stored by a program can be read by other programs. A covert channel is a communication channel based on the use of system resources not normally intended for communication between the subjects (processes) of the system. For example, a program that varies its pagination rate when processing some critical data could transfer information to another program which can trace back the critical data by examining those variations. As another example, a covert channel between a 'high' subject and a 'low' subject can occur via Trojan Horses included respectively in 'high' and 'low' software, which leak information, unbeknown to the 'high' subject. The concern of covert channels is mainly with subjects (not users) which might contain infected code (Levin *et al.*, 1990).

1.4 Security controls

Database protection can be obtained through security measures for:

- Flow control
- Inference control
- Access control.

To these controls, *cryptographic techniques* can be added, which consist of coding stored data under a secret enciphering key. Through these techniques secrecy of information is assured by making data visible to anyone but only understandable to authorized users (Denning, 1982).

1.4.1 Flow control

Flow controls regulate the distribution (flow) of information among accessible objects (Denning, 1976, 1977, 1982). A flow between object X and object Y occurs when a statement reads values from X and writes values into Y. Flow controls check that information 'contained' in some objects (for example, reports) does not flow explicitly (through a copy) or implicitly (via groups of instructions involving intermediate objects) into less protected objects. Should this occur, a user could indirectly get in Y what he cannot get directly from X, thus leading to secrecy violation.

Copying data from X to Y is the typical example of information flow (from X to Y). Moreover, partial information flow occurs when part of the information in X is copied or when data moved to Y is not exactly data contained in X but can provide information about it. An example is a test operation on X: by observing the result of this test, information can be inferred on the value of X.

Flow-control policies require admissible flows to be listed or regulated. A flow violation occurs via a request for data transfer between two objects for

which, on the basis of admitted flows, that transfer is unauthorized. Control mechanisms should deny the execution of these requests.

Often, flow-control problems are faced by classifying the system elements: that is, objects and subjects. 'Read' and 'write' operations among these are authorized on the basis of relationships among the classes. Higher-class objects are more protected during 'read' access than lower-class objects: here flow controls prevent violations consisting of information transfer to lower levels, that is, to more accessible levels. Instead, transfer to higher levels, namely to more protected levels, is allowed. The risk is to over-classify information with more protection than is appropriate for its actual sensitivity. This problem in flow-control policies has been solved by defining operations that allow transfer of objects to lower levels, while keeping unaltered the sensitivity of data in these objects.

1.4.2 Inference control

Inference controls aim at protecting data from indirect detection (Denning and Schlorer, 1983). This occurs when a set X of data items to be read by a user can be used to obtain the set Y of data as $Y = f(X)$, that is, by applying a function f to X.

An **inference channel** is a channel where users can find an item X and then use X to derive Y as $Y = f(X)$.

The main inference channels in a system are:

(1) *Indirect access.* This occurs when an unauthorized user succeeds in knowing the set of data Y (for which he is not authorized) through a query on the set of data X visible to him, and undergoing conditions on Y. For example, the following query:

```
SELECT X FROM r WHERE Y = value
```

reveals that n-tuples may exist in the relation r verifying the condition Y=value.

Another example is the insertion of a record with an n-tuple having the same key of an already existing n-tuple not visible to the user performing the insertion. Upon system denial, the user infers the existence of an n-tuple that he cannot access, holding that specific key value. Indirect access is a flow violation, and problems concerning flow violations can sometimes be included in inference controls.

(2) *Correlated data.* Correlated data is the typical inference channel where visible X data is semantically connected to invisible Y data. Consequently, information about Y can be obtained by reading X. For example, in $z = t \times k$, t and k being visible, and z invisible, the value of z can be inferred from the existing 'times' arithmetical relation. Decreasing the uncertainty degree about a data value as a result of authorized requests can also be considered as inference.

(3) *Missing data.* A channel of missing data is an inference channel through which users can come to know the existence of a set of data X. Typically, a user can get an object name, although he or she may be prevented from access to the information contained therein. Another example is answering a user query about a relation by displaying, together with authorized information, null values which mask the sensitive inaccessible information. These null values make it possible to infer the existence of the masked item value (sometimes the value itself), or at least decrease the uncertainty about it.

Statistical inference is a further aspect involving deduction of data. The basic aspects of statistical inference are presented in Denning (1982) and Denning and Schlorer (1983): the issue is typical of databases releasing statistics about groups of individuals. In these databases, access to individual data is not allowed; instead, data is accessible only via *statistical functions*. However, traces of the original items contained in data obtained via statistics could be obtained by skilful users. Statistical attacks can be faced by two types of control:

(1) *Data perturbation.* This control is performed directly on the protected data. For example, it is achieved by grouping sensitive information and by replacing the original items by microstatistics so that the resulting global value is unchanged, but no single items are left in the database. As another method, information on single records can be biased through a proper perturbation function; this preserves data privacy without affecting the global statistics appreciably.

(2) *Query controls.* These are the most widely used. Most of them are based on the 'size' (number of records) of the query set (namely the set of records verifying the query). A well-known technique is the 'query-set size control' consisting in returning data only about sets sized between k and $n - k$, n being the total size of the database, and $k > 1$ a selected parameter. Query controls are also implemented by determining the sensitivity degree of information. To this purpose, the user's previous knowledge about data must be taken into account when processing a new query, in order to compute the total amount of information that the user would finally get and what damage to data secrecy could result from this. Despite satisfactory results, this kind of control proves to be expensive and difficult to manage. Quantifying the users' knowledge of information is difficult, since it depends on sources external to the system, or released formerly and no longer belonging to the system.

1.4.3 Access control

Access controls in information systems are responsible for ensuring that all direct accesses to the system objects occur exclusively according to the modes and rules fixed by protection policies. An *access control system* (Figure 1.6) includes subjects (users, processes) who access objects (data, programs) through operations ('read', 'write', 'run'). Functionally, it consists of two components:

(1) A *set of access policies and rules:* information stored in the system, stating the access modes to be followed by subjects upon access to objects.

(2) A *set of control procedures (security mechanisms)* that check the queries (access requests) against the stated rules (query validation process); queries may then be allowed, denied or modified, filtering out unauthorized data.

Security policies

The security policies of a system are high-level guidelines concerning design and management of authorization systems (Fernandez *et al.*, 1981). Generally, they express the basic choices taken by an organization for its own data security. The definition of security policies leads to the explicit formulation of security *strategies*, thus giving security its rightful relevance instead of the fragmentary and approximate consideration it is often given. In fact, a clear and complete description of security strategies of an organization allows critical points and possible conflicts to be discussed and solved, and the objectives to be analysed and correlated.

Security policies define the *principles* on which access is granted or denied. Sometimes besides 'if', they state 'how' an access should be granted, or that the queries can return partial results, filtering out unauthorized data.

Authorization rules (access rules) are the expression of security policies; they determine the system behaviour at run time. The security policies should also state how the set of authorization rules (insertion, modification) is administered. Examples of security policies follow.

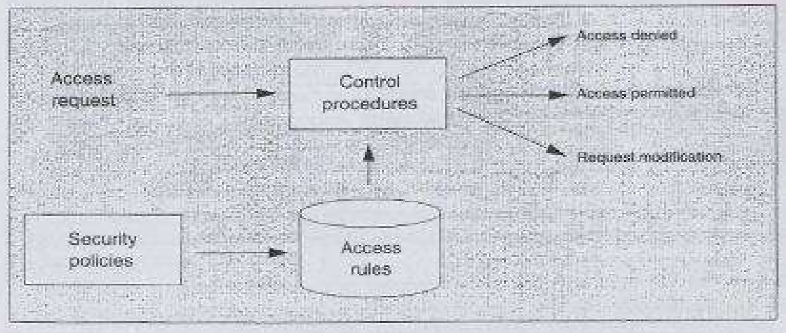

Figure 1.6 Access control system.

The question, 'How much information is accessible to each subject?' involves the problem of **access limitation**. Two basic policies exist:

(1) *Minimum privilege policy*, also called 'need-to-know' policy. According to this policy, system subjects should use the minimum quantity of information needed for their activity. This assumes a definable minimum that is sometimes hard to evaluate. A drawback of this policy is that overlimiting may lead to strong and useless restrictions for innocuous subjects.

(2) *Maximum privilege policy*, based on the principle of the 'maximum availability' of data in a database, so that sharing is maximized. This policy is adequate for environments such as universities or research centres, where strict protection is not particularly needed, because of both the reliability of users and the data-exchange requirements.

In a **closed system** only explicitly authorized accesses are allowed. In an **open system** accesses that are not explicitly forbidden are allowed. A closed system policy states that, for each subject, authorization rules exist specifying the access privileges of that subject on the system objects. These will be the only rights this subject is granted by the control mechanism. An open system policy states that, for each subject, authorization rules exist specifying the privileges that subject does not hold on the system objects. These will be the only rights this subject will be denied by the control mechanism.

Open and closed systems are mutually exclusive. When deciding upon security strategies, the choice depends on the features and requirements of the database environment, users, applications, organizational aspects, and so on. A closed system enforces the minimum privilege policy, whereas an open system enforces maximized sharing. Protection is higher in closed systems: errors such as a missing rule can deny authorized access but cause no damage, whereas in open systems this same event can grant unauthorized access.

Closed systems allow an easier evaluation of the authorization state of a system, that is, the privileges held by all users, and, for this reason, are often the preferred choice among this type of system. However, the choice also depends on the kind of environment and protection requirements; for example, for maximized sharing, where access grant may overcome access denial, closed systems are hard to manage and authorization rules onerous to insert.

Access controls, according to closed- and open-systems policies, are shown in Figures 1.7 and 1.8 respectively.

In a security system, the definition of **policies for authorization management**, that is, 'who' can grant or revoke access rights, is a relevant procedure. Grant and revocation are not always the prerogative of an authorizer or of a single security officer (centralized authorization). Sometimes, authorization management requires decentralization to different authorizers. This is typical of distributed systems, where various local systems are often managed autonomously. It also happens in large information systems, where portions of the same database are 'logically' partitioned into different databases, each managed by a local DBA.

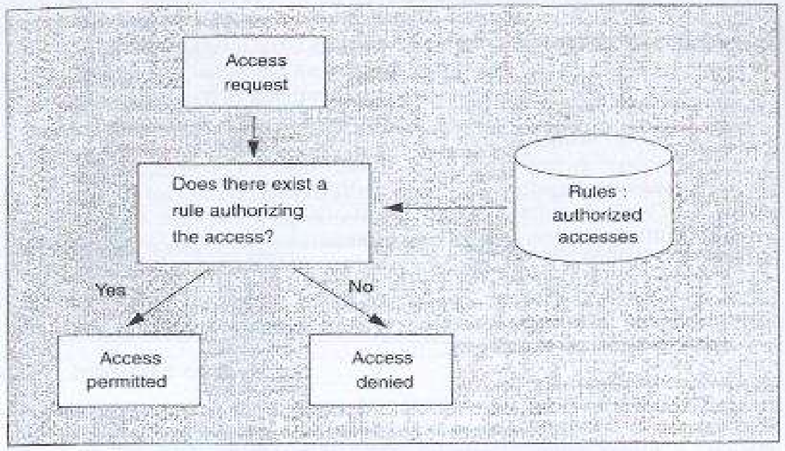
Access request
Does there exist a rule authorizing the access?
Rules : authorized accesses
Yes
No
Access permitted
Access denied

Figure 1.7 Access control: closed system.

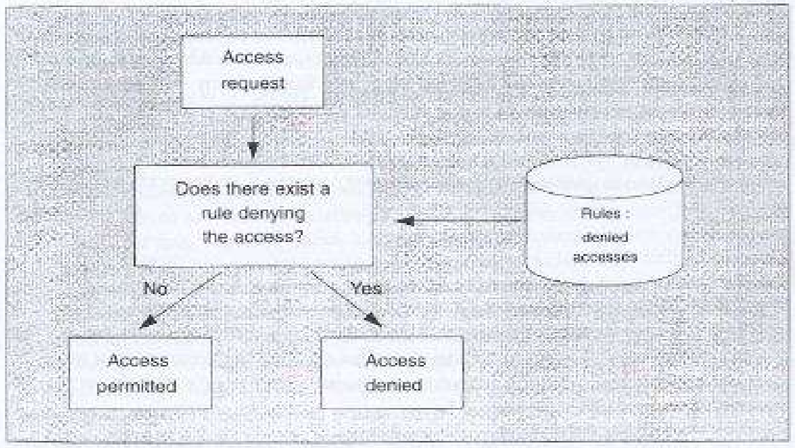

Figure 1.8 Access control: open system.

The choice between centralized or decentralized administration is a security policy. Intermediate policies are, however, possible. For example:

- *Hierarchical decentralized authorization*, where a central authorizer is responsible for distributing administrative responsibilities of a database among dependent subadministrators. The central authorizer thus nominates and revokes subauthorizers, for example, according to the hierarchy of his or her organization.
- *Ownership*. Upon object creation (for example, a table in a relational database), the user creating the object becomes its default owner; he or she can thus grant or revoke access to that object. Sometimes a central administrator's approval may be required.
- *Cooperative authorization*. Special rights on certain resources cannot be authorized by a single user but only by a predefined group of users who must all agree on access granted.

Access-control policies state if, and how, system subjects and objects can be grouped, in order to share access modes according to given authorizations and rules. Moreover, the policies state if, and how, access rights can be transferred.

Grouping or classifying users who hold some privileges, or resources that share common protection requirements, greatly simplifies the specification of the security policies and the implementation of the security mechanisms. Therefore, various grouping criteria have been proposed.

Problems related to group management comprise, at the design level, the consequences of a user being a member of different groups and, at the implementation level, how to manage changing group membership.

Membership is implemented by default in various systems to simplify design and enforcement problems. Classification in hierarchical levels (totally or partially ordered) for groups of subjects and groups of objects is a widespread procedure; access controls map into controls of information flow among the different levels. This procedure is largely used in **multilevel security systems** for military environments, where access control policies are essentially control policies for information flow.

Multilevel systems have been successful because of their underlying formal security model; for these models, theoretical research has been carried out on the general properties of security systems. Access policies for multilevel systems can be either mandatory or discretionary. Mandatory Access Controls (MAC) restrict the access of subjects to objects on the basis of *security labels*. Discretionary Access Controls (DAC) allow access rights to be propagated from one subject to another.

A **mandatory policy** for access control applies to large amounts of information requiring strong protection in environments where the system data can be classified and the users cleared. A mandatory policy can also be defined as a flow-control policy because it prevents information flow towards objects of a lower classification. Access to data is determined by a mandatory policy through definition of subject and object security classes. The two main features

of an object security class are: the *classification level*, which reflects the information it contains, and the category (application area) to which object information refers. Classification levels are, for example:

```
0 = Unclassified
1 = Confidential
2 = Secret
3 = Top secret
```

Categories tend to reflect the system areas or departments of the organization. In military environments, for example, the following could be areas:

```
Nuclear - Nato - Intelligence
```

while in industrial environments areas could be:

```
Production - Personnel - Engineering - Administration
```

For *m* system areas, possible categories derivable from their associations would be 2^m.

A relation of partial order is defined on the Security Classes *SC* defined as:

$SC = (A,C)$

A being the classification level, and *C* a category. This relation states that, given two security classes, $SC = (A, C)$ and $SC' = (A', C')$:

$SC \leq SC'$

if, and only if, the following conditions are verified:

$A \leq A'$ and $C \subseteq C'$

Thus, for example, the relation:

(2, *Nuclear*) ≤ (3, (*Nuclear, Nato*))

is verified, while:

(2, (*Nuclear, Nato*)) ≤ (3, *Nato*)

is not verified.

Each subject and each object is assigned a security class comprising a sensitivity level and a set of categories. These two components correspond, both for subjects and objects, to their role in the system.

A subject classification reflects the degree of trust that can be assigned to that subject, and the application area where it operates. An object classification reflects the sensitivity of the information contained in the object, i.e., the potential damage that could result from unauthorized disclosure of information.

A set of axioms determines the relations to be verified between the subject class and the object class in order to allow subjects to access objects according to security criteria. These relations depend on the access mode.

With regard to access right transfer, assigned rights cannot be changed and modifications are permitted only to the authorizer. This means that full control on the authorization system is kept by the authorizer. Access control through mandatory policies is depicted in Figure 1.9.

Discretionary policies require that, for each *subject*, *authorization rules* be defined specifying the *privileges* owned on the system *objects*. Access requests are checked by the discretionary control mechanism and access is granted only to subjects for which an authorization rule exists and is verified (Figure 1.10).

Discretionary policies are based on the identification of the user requesting the access. 'Discretionary' means that the possibility exists for users to grant and revoke access rights on some objects. This implies the decentralization of the administrative control through ownership. However, a discretionary policy is also well suited for centralized administration. In this case, authorizations will be managed by the system administrator: decentralized administration implies discretionary control policies, while the contrary does not normally hold. Discretionary policies need more complex authorization mechanisms, also aimed at avoiding loss of control on right *propagation* from the authorizer or other responsible persons.

Revocation of propagated rights is a further problem. For each revoked right, the user(s) who subsequently granted or received it must be identified by the system. Various revocation policies exist for this purpose. DAC has some

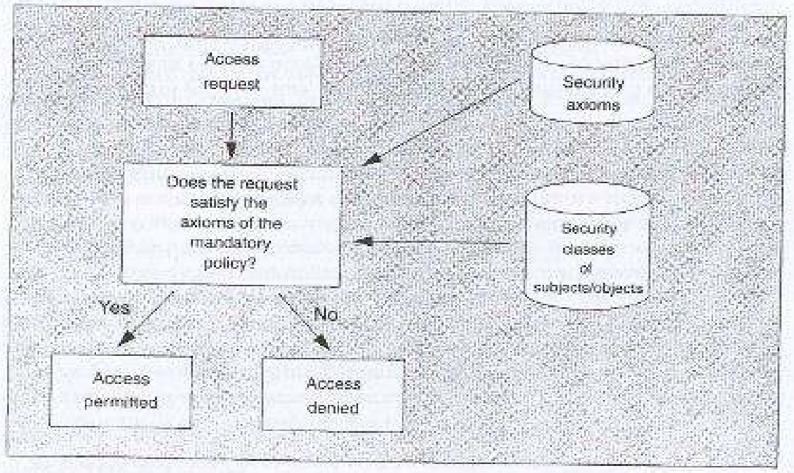
Access request
Security axioms
Does the request satisfy the axioms of the mandatory policy?
Security classes of subjects/objects
Yes
No
Access permitted
Access denied

Figure 1.9 Mandatory access control.

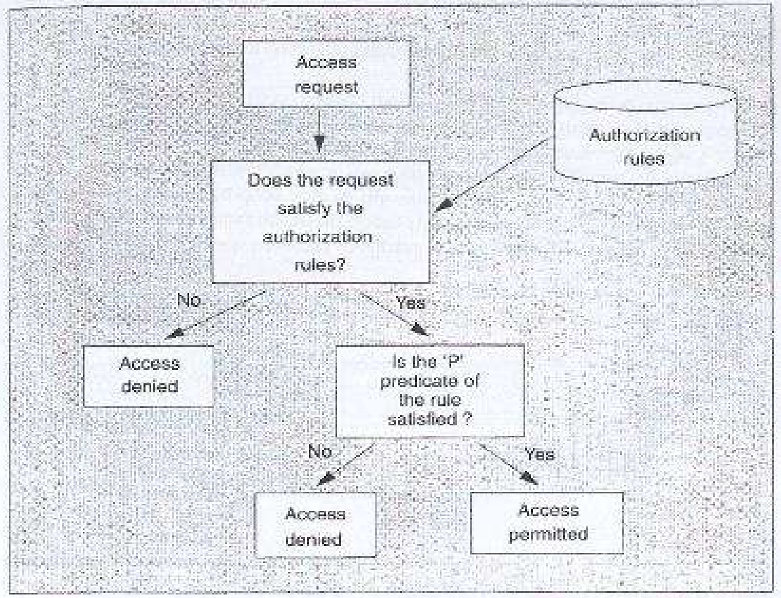
Access request
Authorization rules
Does the request satisfy the authorization rules?
No
Yes
Access denied
Is the 'P' predicate of the rule satisfied ?
No
Yes
Access denied
Access permitted

Figure 1.10 Discretionary access control.

inherent weaknesses: it allows information from an object that can be read to flow to any other object that can be written by the subject; even if the user is trusted not to do that deliberately, it is still possible for Trojan Horses to copy information from one object to another.

Mandatory policies and discretionary policies are not mutually exclusive. They can be combined: the mandatory policy applies to the authorization control, while the discretionary policy applies to access control. If an access query meets the discretionary controls, for example, if authorizations exist for this query, it will also meet the mandatory policy axioms for the controlled authorizations (Figure 1.11).

Authorization rules

As described before, translating requirements and security policies into a set of authorization rules is the authorizer's task. Normally, security requirements are both determined by the organization and identified by the users, on the basis of their own experience.

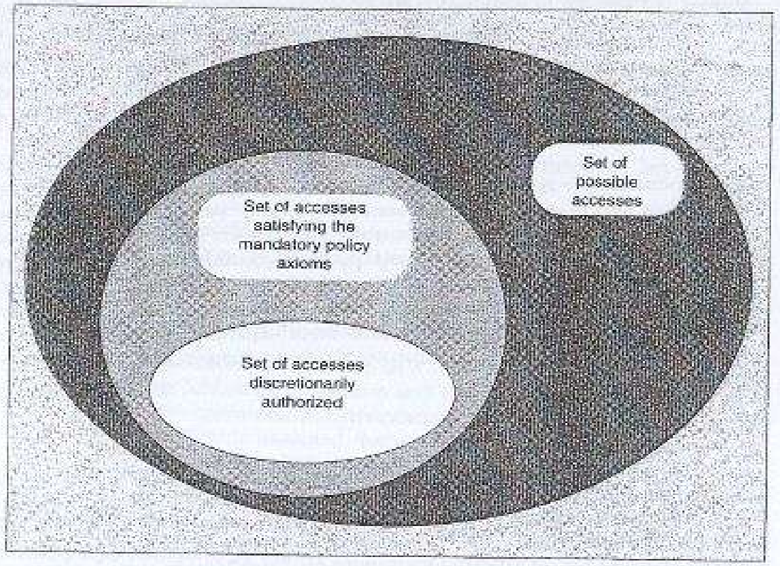
Set of possible accesses
Set of accesses satisfying the mandatory policy axioms
Set of accesses discretionarily authorized

Figure 1.11 Authorized rights in contemporaneous application of mandatory and discretionary policies.

Authorization rules are expressed in compliance with the hardware/software environments of the protection system, and the adopted security policies. The design process of a security system must supply a model supporting the authorizer when mapping requirements into the rules, according to the security policies that have been considered (Figure 1.12). Since single entities of the system are taken into consideration at the requirement specification level (for example, user `'Smith'`, file `'filename.ext'`, program library of application `'A1'`), the amount of associated requirements and access rules tends to be large. Different security models exist (see Chapter 2) supporting the representation of access rules as a structured and formalized design activity (Landwehr, 1981; Millen and Cerniglia, 1984). As an example, in the access matrix model, the set of security rules of a system is represented as a matrix A called *Authorization, or Access Matrix*, whose rows represent system subjects and columns represent system objects. An entry $A[i,j]$ represents the type of access allowed for subject s_i to object o_j.

An example of authorization matrix is depicted in Table 1.1, where R = Read, W = Write, EXEC = Execute, CR = Create, and DEL = Delete. This is a simple case of **access control** based on the object name, called *name-dependent* access.

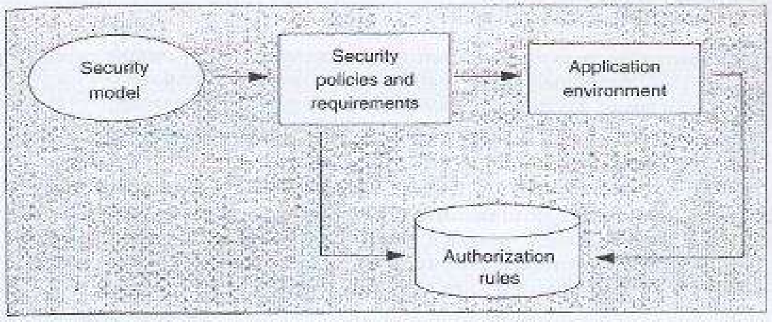

Figure 1.12 Design of authorization rules.

$A[i,j]$ may contain more complex security rules specifying access constraints between subjects and objects (for example, depending on the contents of j).

The following rule: 'User 1 can read record Y only if the third field of Y is < 100' is an example of *content-dependent control*. In this case, the matrix entries need to be 'extended' through a predicate expressing a constraint on the privilege. A security rule can be represented as a quadruple (Fernandez *et al.*, 1981; Bussolati and Martella, 1982):

$\langle s,o,t,p \rangle$

where s = subject, o = object, t = type of access right and p = (optional) predicate.

A predicate may also refer to some system variables, such as the date, time and query source, thus establishing the basis for *context-dependent control*. An example of context-dependent control is: 'classified information cannot be accessed via a remote login', or 'salary information can be updated only at the end of the year'. Content-dependent controls are beyond the scope of operating systems and are provided by the DBMS. Context-dependent controls can be partially provided by the operating system and partially by the DBMS. More sophisticated controls (such as controls based on the past history of accesses) require database support.

Table 1.1 Authorization matrix.

	Objects		
Subjects	File F1	File F2	File F3
User 1	R, W	EXEC	EXEC
User 2	–	–	CR, DEL
Program P1	R, W	R	–

Security rules should also specify non-accessible aggregations of data, taking into consideration that data sensitivity increases via aggregation. For instance, users can be granted reading of both name and salary values of the employees separately, but be forbidden to read the `'name-salary'` aggregation: that is, they cannot correlate salaries to individual employees.

Access to programs handling data, such as some system programs or applications, should also be controlled. These are represented as objects in the columns of an authorization matrix. Therefore, problems and mechanisms typically concerning data protection are extended to deal with the more complex problem of the logical protection of the whole of the system resources. With discretionary policies, where grant or revocation of access rights is demanded to several authorizers, a security rule can be a 6-tuple:

$\langle a, s, o, t, p, f\rangle$

where a is an authorizer subject who granted s the right $\langle o, t, p\rangle$, while f is a copy flag describing the possibility for s to further transfer $\langle o, t, p\rangle$ to other subjects. The security policies **(access rules management)** concerning access right transfer determine the presence of this flag and its use. For example, in some systems the flag is reset upon n subsequent grants, thus allowing an n-deep privilege transfer.

Sometimes, a set of Auxiliary Procedures (AP) is associated to a security rule, to define which actions can be possibly undertaken by the security system, should an access query partially meet a rule. For each AP_i belonging to a set AP, a condition C_i is specified which can be a function of other components of the rule, or of system variables. A more complete form of an authorization rule is then the following:

$\langle a,s,t,o,p,f, \{C_i, AP_i\}\rangle$

Security mechanisms

An access control system relies on security mechanisms that are functions implementing the security rules and policies (Denning, 1982). Security mechanisms concern the *prevention* of improper access (access control mechanisms), and the *detection* of improper access (auditing and intrusion detection mechanisms). Good prevention and detection require good authentication mechanisms. Access control mechanisms are more fundamental because prevention is preferred. However, sometimes, detection is the only option: for example, accountability in proper use of privileges or against modification of messages in a network.

Security mechanisms can be implemented via hardware, software or through administrative procedures (Landwehr 1983, Myers 1990). During the development of a security system, policies and mechanisms should be separated, making it possible to:

- Discuss access rules independently of the implementation mechanisms. This allows designers to focus on the correctness of the security requirements being captured and on the consistency of the security policies;

- Compare different access control policies or different implementation mechanisms for the same policy;
- Design mechanisms with the ability to implement different policies. This is necessary when policies need dynamic changes to accord with modifications to the application environment, and consequently with protection requirements. Should security policies be strictly related to implementation mechanisms, changes to policies must be adequate for the implementation of the control system.

The achievement of mechanisms complying with the designed policies is a crucial issue. In fact, the incorrect implementation of a security policy leads to incorrect access rules or to insufficient support of protection policies. Two basic types of system fault can derive from incorrect implementation:

(1) *Denial of allowed access.*
(2) *Grant of forbidden access.*

External mechanisms

These consist of administrative and physical control measures able to prevent undesired access to the physical resources (rooms, terminals, devices), so that only authorized accesses are allowed. Devices providing protection against accidental threats like short circuits, fire, earthquakes or environment conditions can also be included among external protection mechanisms. However, full protection cannot be assured, particularly in those environments where accidental attacks or violations can hardly be foreseen. The target is then to minimize possible damages. This means:

- Minimize possible violations;
- Minimize possible consequent damages;
- Provide recovery procedures.

Internal mechanisms

These are protection measures to be applied once a user bypasses, or receives authorized access by, the external controls. Authentication of user identity and verification of the legitimacy of the required actions according to the user authorizations are the basic actions. Internal protection consists of three principal mechanisms:

(1) *Authentication.* This mechanism prevents unauthorized users from using a system by checking their identity through:
    - Something a user is acquainted with (passwords, codes, etc.);
    - Something the user owns (magnetic cards, badges, etc.);
    - Physical characteristics of the user (fingerprints, signature, voice, etc.).
(2) *Access controls.* Upon successful authentication, queries entered by users can be answered only according to existing authorizations for these users.

(3) *Auditing mechanisms*. These monitor the utilization of the system resources from its users. Auditing mechanisms consist of two phases:
- A *logging phase*, where all the access queries and related answers (both authorized and denied) are recorded;
- A *reporting phase*, where reports from the previous phase are checked to detect possible violations or attacks.

Auditing mechanisms are relevant for data protection because they support:
- The evaluation of the *system reactions* in the face of some types of threat; this also helps to detect system leaks or inadequacies;
- The *detection of violation attempts* executed through sequences of queries.

Moreover, attempts or threats are discouraged because of the users' consciousness of auditing procedures monitoring every operation.

An overall view of the architecture of a DBMS including security features appears in Figure 1.13, where modules and users are shown (the figure is an elaboration of that contained in Fernandez *et al.* (1981) of a reference secure DBMS architecture). It is assumed that protected data is accessible only via DBMS functions. After user login and authentication, each subsequent access query to the database (generally made through an application program) is mediated by the Authorization System (AS) procedures, which consult the files of the authorization rules to check query compliance with these rules. Access is allowed upon query-rules match. Otherwise, an error message can be sent to the user, and/or violations can be registered in a log file together with references (for example, date, time, user) by the AS. This log file is periodically examined by the auditor, to detect possible suspect behaviours or to verify recurring types of violations.

A specialist, the security administrator, is responsible for defining the authorization rules derived from the security requirements of the organization. The authorizer may be the same auditor and/or the DBA.

Authorized access queries are translated into program calls from the library of applications, processed by the transaction manager and then transformed into data access requests (processed by the data manager). Further controls can be made by the operating system (such as the control of file access) and the hardware, to ensure that data is exclusively transferred into the work area of a requesting user. Crypto-techniques and backup copies are the usual means for the protection of physical data and the program storage system.

The secure DBMS module manages all the queries. It includes the *authorization rules* (for discretionary controls) and the *security axioms* (for mandatory controls); one, or both, is used by the AS, depending on the system protection policies. In the same module, the DB schemas are included since these are also protected objects.

The following roles are involved in the management of a security system:
- The *application manager*, responsible for the development and maintenance of library programs;

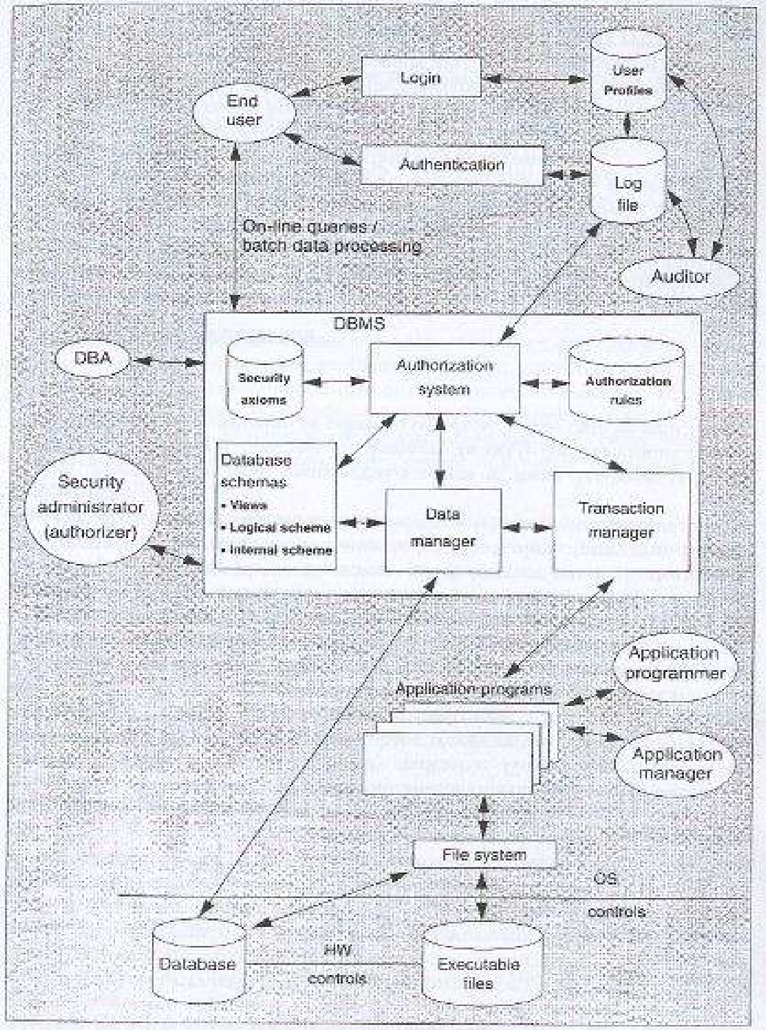

Figure 1.13 Architecture of a DBMS including security features.

- The *DBA*, who manages the conceptual and internal schemas of the database;
- The *security officer*, who defines access authorizations and/or security axioms through rules in a proper language (sometimes, the DDL or DML);
- The *auditor*, responsible for checking sequences of connection requests and of access queries, in order to detect possible authorization violations.

1.5 Designing database security

As we have seen, database security includes:

(1) An *external level*, meaning control of physical access to the processing system, and protection of the processing system from natural, man-made, or machine disasters;

(2) An *internal level*, against possible attacks on the system from dishonest or disgruntled insiders, and errors and omissions by insiders, and from outsiders.

These levels are referred to as *physical security* and *logical security*. In the past few years, physical security has received great attention because of the general view of protection as a *'locking process'* of the system resources within a secure physical environment. Security cannot be assured by relying solely on physical protection. In fact, fraudulent access to data can be performed by authorized users, who can take unfair advantage of their privileges. Therefore, access to 'sensitive' information should be granted to selected users, according to selected access modes and to limited sets of data items.

In general, **protection requirements** of a system are strictly connected to the environment where the system is used, and to economical considerations. Security features are recognized to constitute additional costs and cause downgrading of performance. They also lead to a higher system complexity; a loss in flexibility; the need for human resources for design, management, and maintenance; and the need for additional hardware and software. However, it is also recognized that a lack of security measures, by exposing the system to risks, might bring about much heavier costs upon system failure. The risks should be accurately estimated on the basis of the typology of the environment and of its users. For example, a distinction can be made between security requirements of private/commercial information systems and those of government departments.

1.5.1 Databases in government departments

Analysis of database security problems in some countries has led to some databases being classified according to their contents: *vital information*, that is, necessary to national security, and *non-vital information*, which should be granted under proper controls or authorizations. Databases containing these types of information are called **classified databases**. Data therein is classified at

different security levels (secret, confidential, etc.) according to its level of sensitivity. Access is granted to users and transactions properly *cleared* for a given correct security level. Databases belonging to the Department of Defense, State Department, Department of Treasury, and Department of Industry and Justice are usually included in this category (Landwehr, 1988).

In these environments, penetration attempts are hardly detectable. Individuals wishing to infiltrate the database would be highly motivated and would use sophisticated traceless tools. Information integrity and denial of service (which means preventing authorized users from employing the system resources) are relevant problems in this type of database.

Currently, software environments and DBMSs offering *provable* protection to classified databases seem to be hardly realizable. Great research efforts are being devoted to achieve *trusted* database management software. Hale and Coates (1985) describe the basic requirements of such software, while Schell and Denning (1986) illustrate the integrity aspects of trusted databases.

A viable approach for classified databases is to make access authorizations dependent on the personal identity and certifiable reliability of users.

The described type of protection has long since been recognized to be inappropriate for public, unclassified databases (Lipner, 1982). Their contents refer, for example, to energy plants, to census, social, fiscal or criminal information, to commercial data, such as economic indicators, budget, forecast and development plans of great interest to companies applying for public contracts and competitions.

Cost quantification from unauthorized access to these databases is hardly estimable: for example, how can the damage caused to the individual derived from disclosure and diffusion of personal (medical, fiscal) data be evaluated?

1.5.2 Commercial databases

At first sight, damage estimation is easier for *information systems* of commercial organizations. As a matter of fact, sensitivity degrees of data are directly stated by the organization (for example, industry, financial or insurance body), by distinguishing between data that is vital for the organization, and ordinary data requiring low protection. So far, security design has rarely been given primary consideration in commercial databases and the related security problems have received limited attention in the literature.

Within these environments, security problems arise from authorized users; in fact, a preliminary check on the users' reliability seems almost impossible to achieve methodically. Consequently the procedures for authorization grant are often inadequate, and control techniques and tools for checking that users access only data and programs that they are authorized for are rather poor.

Moreover, the complexity of security problems depends on the *semantic nature* of a database. The entity to be protected is not a physical resource or device, such as a file or a terminal, but rather a set of items representing information with a global meaning. Therefore, protection requirements are not only *selective*, that is, dependent on the user identity or on the access mode, but also sensitive to the contents of data, or dependent on elements such as system parameters, access history and data aggregation. This makes the design, interpretation and enforcement of an authorization model for databases a complex task.

Finally, consider that access grant and revocation should be able to follow the dynamics of an organization: to this purpose, the consistency of the set of authorizations in the system not only needs continuous checks against the protection requirements, but also needs possible modifications.

The degree of security provided by current commercial DBMS technology is rather low. Practically, databases appear to be extremely vulnerable even to simple attacks, let alone sophisticated attacks like Trojan Horses, inference threats, worms, trackers or trapdoors. Note that all these terms are typical of the literature on computer security – see, for example, Denning (1990, 1982).

Protection techniques for both unclassified public databases and commercial databases are substantially similar, being based on hardware/software tools and on organizational measures. Attempts are being made to adapt trusted DBMS techniques to commercial environments (see, for example, Turn (1982) for a discussion). However, they often appear inflexible and inadequate; moreover, they turn out to be totally inadequate in the face of threats like Trojan Horses and trapdoors embedded in application software (Hinke, 1988).

Specialized DBMS architectures, called *multilevel secure DBMS architectures*, have been proposed to meet multilevel protection requirements. Some multilevel architectures are the Integrity Lock architecture, the Kernelized architecture, the Replicated architecture (as proposed in the Woods Hole Study performed in Summer 1982), and the Trusted Subject architecture. The Replicated architecture exists as a research prototype developed at NRL (Naval Research Laboratory) and no commercial systems are currently available. The Integrity Lock architecture has been studied at the Mitre Corporation and is available as a commercial product (Trudata). The Kernelized architecture has been studied at SRI (Stanford Research Institute) and is embedded in one secure version of Oracle. Most of the commercially secure DBMS products (for example, Sybase, Rubix, Informix, Oracle, Dec) implement the trusted subject architecture as developed in a research prototype (ASD-TRW). Some techniques used in these prototypes are: the partitioning of multilevel databases into various one-level databases (thus producing redundancy and inconsistency); the application of crypto-techniques on sensitive data; the *integrity lock* techniques specifying sensitivity level and checking unauthorized modifications (through *checksum* techniques); and mechanisms based on 'sensitivity keys', which are a combination of an identifier with a sensitivity level of the data item. Further techniques introduce a front end between the user and the DBMS, so that multilevel protection is managed without altering the DBMS protection features. Alternatively, a user/DBMS filter is employed, or views are defined, containing only the information a user is authorized for.

1.6 Concluding remarks

Access control in a system is guided by *access policies* stating who can access which objects of that system. Access policies should be independent of the mechanisms that will implement the physical control of access: this allows control mechanisms to be available for different polices. Access policies specify *access requirements*: that is, detailed access modes can be expressed for access by a subject (or group of subjects) to an object (or group of objects).

Requirements are then translated into access rules on the basis of the adopted policies. This is a crucial phase in secure system development. Correctness and completeness of the rules and of the corresponding implementation mechanisms are involved. A mapping process should be performed using modelling techniques for security requirements and policies: a model allows the designers to clearly represent and verify the security properties of the system.

The variety of possible threats to database secrecy and integrity makes database protection a complex problem. Therefore, protection needs a set of measures involving humans, software and hardware. Weakness in any one of these might compromise the whole system security. Moreover, data protection, as a part of a more complex security system, raises issues of system reliability.

In summary, in secure system development one needs to consider some crucial aspects, such as:

- The features of the actual storage and processing environment. A careful analysis is essential to state the degree of protection offered and required by the system: these are the security requirements;
- The protection mechanisms external to the processing environment. These are the administrative and physical control measures that contribute to the effectiveness of operational security measures;
- The internal DB protection mechanisms. These act once the controls (login and authentication) have been passed successfully by the user;
- The physical organization of stored information;
- The security features provided by the operating system and by the hardware;
- The reliability of hardware and software;
- The administrative, human and organizational aspects.

References

Andrews D.J. and MacEwen G. (1990). *A Review of Tools and Methods for System Assurance*. Andyne Computing Ltd

Atzeni P. and De Antonellis V. (1993). *Relational Database Theory*. Benjamin Cummings

Bell D.E. (1990). Lattices, policies, and implementations. In *Proc. 13th National Computer Security Conf.* October 1990

4.2.2 The System R authorization model

To illustrate how some of the previous issues have been implemented in one of the very first DBMSs, we present the authorization model defined by Griffiths and Wade (1976) and later revised by Fagin (1978), in the framework of the relational DBMS System R, developed at the IBM Research Laboratory in San Jose.

The authorization model of System R considers, as objects to be protected, tables of the database. These can be either base tables or view tables (see Chapter 1). Subjects of the model are the users who can access the database. The privileges considered by the model are the access modes applicable to the tables of the database. In particular, the following access modes are considered:

Read	to read tuples (rows) from a table. The authorization for the read privilege entitles a user owning it also to define views on the table.
Insert	to add tuples to a table.
Delete	to delete tuples from a table.
Update	to modify existing tuples in a table.
Drop	to delete an entire table from the system.

All the access modes refer to a table as a whole with the exception of the update privilege which can refer to specific columns inside a table.

The model supports a decentralized administration of authorizations. In particular, any database user may be authorized to create a new table. When a user creates a table, he or she is solely and fully authorized to execute privileges on the table (this is not completely true if the table is a view, as we will see later on). As owner, the user is also the only one entitled to drop the view. The owner can grant other users the privileges on the table. When a user grants another user a privilege on a table a new authorization is inserted among the set of authorizations held in the system. Privileges can be granted with the grant option, meaning the receiver is allowed to grant privileges, and the grant option, to other users. Each authorization can be characterized as a tuple ⟨*s, p, t, ts, g, go*⟩ where:

s is the subject (user) to whom the authorization is granted (that is, the *grantee*);
p is the privilege (access mode);
t is the table to which the authorization refers;
ts is the time at which the authorization was granted;
g is the user who granted the authorization (that is, the *grantor*);
go ∈ {yes, no} indicates whether *s* has the grant option for *p* on *t*.

If the privilege in the authorization is 'update', the columns on which the privilege can be executed must also be indicated. The grant option is similar to the copy flag of the access matrix model. If a user holds the authorization for a privilege on a table with the grant option, the user can grant other users the privilege on the table as well as the grant option on it.

For example, tuple <B, select, T, 10, A, yes> indicates that user B can select tuples from table T, and grant other users authorizations to select tuples from table T, and that this privilege was granted to B by user A at time 10. Tuple <C, select, T, 20, B, no> indicates that user C can select tuples from table T and that this privilege was granted to C by user B at time 20. The authorization does not entitle user C to grant other users the select privilege on T.

The reason why, in each authorization, the grantor and the time the authorization was granted are indicated, is to enforce a special type of revoke procedure as we will illustrate later.

The data definition, manipulation and control language of System R, named SQL, has been extended to the consideration of statements allowing the user to require execution of grant and revoke operations.

The grant command of SQL has the form:

$$\text{GRANT}\left\{\begin{array}{l}\texttt{ALL RIGHTS}\\ \langle\texttt{privileges}\rangle\\ \texttt{ALL BUT}\ \langle\texttt{privileges}\rangle\end{array}\right\}\texttt{ON}\ \langle\texttt{table}\rangle\ \texttt{TO}\ \langle\texttt{user-list}\rangle\ [\texttt{WITH GRANT OPTION}]$$

The user granting a privilege on a table may also specify the keyword PUBLIC instead of the ⟨user-list⟩ to which the privilege is being authorized. In such a case, all database users are granted the privilege on the table.

Every user who has the authorization for a privilege on a table with the grant option can also revoke the privilege on the table. However, a user can revoke

only those authorizations that were granted by him. In particular, the revocation of a privilege on a table from the revokee by the revoker does not have any effect on possible authorizations for the privilege on the table that the revokee may have received from users different from the revoker.

The revoke command of SQL has the form:

$$\text{REVOKE}\left\{\begin{array}{l}\texttt{ALL RIGHTS}\\ \langle\texttt{privileges}\rangle\end{array}\right\}\texttt{ON}\ \langle\texttt{table}\rangle\ \texttt{FROM}\ \langle\texttt{user-list}\rangle$$

The next section discusses the revoke mechanism.

Revocation of authorizations

The System R authorization model enforces recursive (or *cascading*) revocation. The semantics of the recursive revocation of privilege *p* on table *t* from user *y* by user *x* is defined to be as if all the authorizations for *p* on *t* granted by *x* to *y* had never been granted. Therefore, all the effects brought about by the presence of the authorizations being revoked have to be eliminated. In particular, all the authorizations for *p* on *t* which could not have been granted if the revokee had never received any authorization for the privilege on the table from the revoker will also have to be deleted.

As an example, consider the sequence of grant operation for privilege *p* on table *t* illustrated in Figure 4.1(a), where every node represents a user and an arc between node u_1 and node u_2 indicates that u_1 granted the privilege on the table to u_2. The label of the arc indicates the time the privilege was granted: that is, the timestamp of the authorization. Suppose now that user *B* revokes the privilege on the table from user *D*. According to the semantics of recursive revocation, the resulting authorization state has to be as if *D* had never received the authorization from *B*. If *D* had never received the authorization from *B*, he could never have granted the authorization to *E* (his request would have been rejected by the system). Analogously, *E* could not have granted the authorization

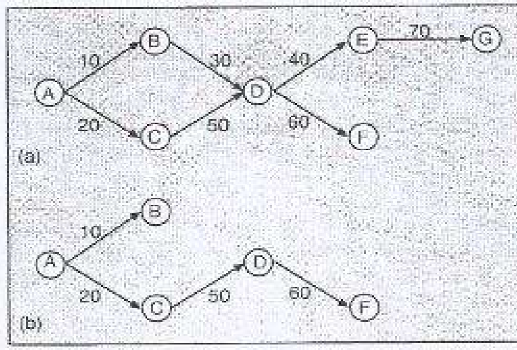

Figure 4.1 An example of a revoke operation: (a) B has granted a privilege to D, who has passed it to E, who has passed it to G; (b) authorization state after B revokes D's privilege.

to G. Therefore, the authorization granted by D to E and by E to G must also be deleted upon the revocation. By contrast, the authorization granted by D to F does not have to be deleted. Indeed, this authorization could have been granted anyway even if D had never received the grant option for the privilege from B. Then, after the revocation, the set of authorizations holding in the system should be as if the sequence of grant operations had been as in Figure 4.1(b).

The revocation algorithm works as follows (Fagin, 1978). Suppose user x revokes privilege p from user y. If there is no authorization for the privilege on the table granted by x to y, then the revoke operation is ignored. Otherwise, every authorization for privilege p on table t granted by x to y is deleted. If at least one of the deleted authorizations was with the grant option, then the authorizations granted by y may also need to be revoked. Let ts be the minimum timestamp of y's remaining authorizations for privilege p on table t with the grant option (if no such authorization still appears, then let $ts = \infty$). Then, each grant by y of privilege p on table t, with timestamp smaller than ts, is deleted from the authorization table. The procedure is repeated for each user from which any authorization is revoked.

Views

System R allows users to define views on top of base tables and other views. Views, which are defined in terms of queries on one or more base tables or views, represent a single and effective mechanism to support content-dependent authorizations. For example, suppose user A creates table T and wants to give user B the authorization to read the tuples of T for which the value of attribute a_1 is greater than 1000. Then, A can define a view of the form "select * from T where $a_1 > 1000$" on top of T, and grant B the read authorization on the view.

The user defining a view is the view owner and is therefore entitled to drop the view. However, he may not be authorized to exercise all privileges on the view. The authorizations that the owner of a view owns on the view depend on the view semantics (certain operations may not be executable on the view) and on the authorizations the user has on the tables directly referenced by the view. In particular, in order to be authorized for a privilege on a view, the owner must own an authorization for the privilege on all the tables directly referenced by the view. Therefore, if a view is defined on a single table, the user can be authorized on the view for all privileges he owns on the table. If the view is defined on a set of tables, the user is authorized for all privileges he owns on every table directly referenced by the view. The privileges on the view, however, may be further restricted depending on the view semantics (see Chapter 1).

The privileges of the view owner on the view are determined at the time of the view definition. For every privilege the user owns on all the tables directly referenced by the view, the corresponding authorization on the view is defined. Although the user may have more authorizations for a privilege on a table, only one authorization for the privilege on the view will be derived. If the user defining the view is authorized for a privilege on all the underlying tables with the grant option, the grant option for the privilege on the view will be given to the user. The grantor of the authorizations derived for the owner of a view on the w is the view owner himself. The timestamp is the time of the view definition.

For example, suppose user B, who is authorized for the read privilege on tables T_1 and T_2 with the grant option, defines view V on top of the two tables. User B receives the authorization for the read privilege with the grant option on the view.

If a user receives the authorization for a privilege on a view with the grant option, he can grant other users the privilege, possibly with the grant option, on the view. In order for these users to use the privilege on the view, it is not necessary to be authorized for the privilege on the tables directly referenced by the view. For instance, with reference to the example just mentioned, if user B grants C the read privilege on the view, C can select tuples from V even if he or she does not have the read authorization on tables T_1 and T_2.

If, after having defined a view, the view owner receives additional privileges on the underlying tables, these will not be passed on to the view: that is, the user will not be authorized for them on the view. By contrast, if after having defined a view, the view owner is revoked a privilege on any of the underlying tables, the privilege may need to be revoked also on the view. For instance, with reference to the example just mentioned, if B is revoked the read privilege on T_1, B's authorization on V and, as a consequence, the authorization on V granted by B to C will also have to be deleted.

Implementation of the model

Information of users' authorizations to access database tables are stored in two relations named SYSAUTH and SYSCOLAUTH. Relation SYSAUTH has the following attributes:

Userid	indicates the user to which the authorizations are referred.
Tname	indicates the table to which the authorizations are referred.
Type	indicates the type of table *Tname*. This attribute has value "R" if the table is a base table and "V" if the table is a view table.
Grantor	indicates the user who granted the authorizations.
Read	indicates the time at which the grantor granted the grantee the read privilege on the table. If there is no authorization given by the grantor to the grantee for the read privilege on the table, the attribute has value 0.
Insert	indicates the time at which the grantor granted the grantee the insert privilege on the table. If there is no authorization given by the grantor to the grantee for the insert privilege on the table, the attribute has value 0.
Delete	indicates the time at which the grantor granted the grantee the update privilege on the table. If there is no authorization given by the grantor to the grantee for the update privilege on the table, the attribute has value 0.
Update	indicates the columns on which the update privilege is granted. It can have the value "All", meaning all columns can be updated, "None", meaning no update privilege, or "Some", meaning the privilege can be exercised only on some columns of the table.
Grantopt	indicates whether the privileges indicated have been granted with the grant option.

Hence, each tuple in SYSAUTH may correspond to more authorizations. In particular, each tuple defines all authorizations on a table granted by the grantor to the revokee. An example of a SYSAUTH table is illustrated in Figure 4.2. The authorizations in the table correspond to the authorizations resulting from the sequence of grant operations illustrated in Figure 4.1(a), supposing that the grant operations are referred to the read privilege on table EMPLOYEE. Note that, in the original version of the model, if a privilege on a table is granted by the same grantor to the same grantee twice, then the later grant operation is ignored: that is, the earlier time stamp is maintained in the column of the corresponding privilege in table SYSAUTH. However, in the revised model (Fagin, 1978) all grant operations are taken into consideration and therefore more authorizations with the same privilege, table, grantee and grantor, but a different timestamp, may exist. This means that more values may need to be associated with columns corresponding to privileges in table SYSAUTH indicating the different times at which privileges have been granted by the grantor to the grantee.

Table SYSCOLAUTH stores information on the columns to which the authorizations for the update privilege indicated in table SYSAUTH can be applied. In particular, if *some* is specified in a tuple of SYSAUTH, then, for each updatable column, a tuple is placed in table SYSCOLAUTH. Relation SYSCOLAUTH has the following attributes:

Userid	indicates the user to which the update privilege has been granted.
Table	indicates the table on which the update privilege has been granted.
Column	indicates the column of Table on which the update privilege has been granted.
Grantor	indicates the user who granted the privilege.
Grantopt	indicates whether the privilege has been granted with the grant option.

Userid	Tname	Type	Grantor	Read	Insert	Delete	Update	Grantopt
B	Employee	R	A	10	0	0	0	yes
C	Employee	R	A	20	0	0	0	yes
D	Employee	R	B	30	0	0	0	yes
E	Employee	R	D	40	0	0	0	yes
D	Employee	R	C	50	0	0	0	yes
F	Employee	R	D	60	0	0	0	yes
G	Employee	R	E	70	0	0	0	yes

Figure 4.2 An example of the SYSAUTH table.