Database Security*

Sabrina De Capitani di Vimercati, Dip. Elettronica, Università di Brescia, 25123 Brescia, Italy

Pierangela Samarati, Dip. di Tecnologie dell'Informazione, Università di Milano, 26013 Crema, Italy

Sushil Jajodia, Dept. of Information & Software Engineering, George Mason University, Fairfax, VA 22030-4444

1 Introduction

All organizations—public, governmental or private, small or large—depend on computerized information systems for carrying out their daily activity. At the heart of each such information system, there is a *database*. At a very general level, we can define a *database* as a persistent collection of related data, where data are facts that have an implicit meaning. For instance, an employee's name, social security number, or date of birth are all facts that can be recorded in a database. Typically, a database is built to store logically interrelated data representing some aspects of the real world, which must be collected, processed, and made accessible to a given user population. The database is constructed according to a *data model* which defines the way in which data and interrelationships between them can be represented. The collection of software programs that provide the functionalities for defining, maintaining, and accessing data stored in a database is called a *database management system* (DBMS).

A database can be seen at different abstraction levels. Typically a three-level view is adopted (see Figure 1) containing an *internal level*, describing the physical storage of the database; a *conceptual* (or *logical level*) providing the users with a high level description of the real world that the database represents; and an *external level* describing the views that different users or applications have on the stored data. The internal level maps the logical objects supported by the data model to the physical objects (files) of the underlying operating system (see Figure 2).

Beside access and processing functionalities, each DBMS must also provide security functionalities to ensure the *secrecy*, *integrity*, and *availability* of the stored data [3]. Providing secrecy means

*The work of Sabrina De Capitani di Vimercati and Pierangela Samarati was supported in part by the European Community within the FASTER Project in the Fifth (EC) Framework Programme under contract IST-1999-11791.

ensuring that data will not be disclosed to unauthorized users. Providing integrity means ensuring that data will not be modified in an unauthorized or *improper* way. In particular, integrity ensures that the stored data correctly reflect the real world. Providing availability means ensuring that the database will always be accessible by legitimate users for the accesses they are authorized for.

Since ultimately a database is mapped to (i.e., stored as) files of the underlying operating system, one may think that a DBMS does not need to deal with security as security functionalities of the operating system would suffice. This is not true, however, since at the operating system level the data interrelationships and their semantics are lost and therefore security restrictions exploiting concepts of the data model cannot be enforced. Some of the differences between databases and operating systems that make it necessary for a DBMS to support security features are as follows.

- *Protection level* A DBMS usually needs to protect data at a fine granularity level (e.g., a record of a file), while an operating system protects data at the file level.
- *Object differences* There is a greater variety of object types in a DBMS than in an operating system. The typical object type in an operating system is a file; in a DBMS there can be relations (tables), tuples (rows within a table), attributes (columns within a table), indexes, metadata, and others.
- *Data interrelationships* A database may include many logical objects with complex semantic interrelationships that must be protected. By contrast, the number of physical objects that the operating system protects is less and no semantic interrelationships are supported.
- *Dynamic versus static objects* Data objects in a DBMS can be obtained by dynamically aggregating data from different physical objects in an operating system. By contrast, files tend to be more static making their protection easier.
- *Lifetime of data* The lifetime and frequency of access of data in a DBMS is quite different than the lifetime of data stored as files in an operating system.
- *User views of data* While in an operating system, users are either granted or denied access to data (files), in a DBMS it is possible to give access to a portion of an object by defining different views for different users.

Because of these differences, it is clear that some security requirements must be supported by the DBMS itself. Of course, the DBMS can rely on basic security services provided by the

underlying operating system. Typical security services provided by the operating system that can be exploited by the DBMS are *physical security controls*, *authentication* and *auditing*. Physical security protects againsts intentional or accidental threats, like fire or natural disasters. Physical security measures also control the physical access to the computer system on which the database is hosted. Examples of physical measures are the use of locks, security guards, badges, and alarms. Authentication is a means of verifying the identity of a party to another, and is a prerequisite for DBMS security controls to ensure that the correct identity of users is being considered (i.e., users are who they claim to be). The simplest form of authentication is based on the use of passwords: users state their identity with a *login identifier* and provide a secret *password*. Finally, auditing is the post facto evaluation of a system's activities, which must therefore be properly logged. Auditing services can be used to perform off-line analysis to determine possible security violations and to recover the correct state of the database in the case integrity has been compromised.

The overall DBMS/OS architecture is depicted in Figure 3. In this chapter, we mainly concentrate on basic security services that are available to users in commercial DBMSs for *access control* and *integrity constraints* enforcement. Since these controls cannot cope with Trojan horse attacks, we include a brief description of the multilevel secure DBMSs. We conclude this chapter with a short discussion of the inference problem.

2 Access control policies

Access control policies define the rules according to which access to the database objects is regulated. The most popular class of access control policies is represented by discretionary access control (DAC) policies, where the word *discretionary* characterizes the fact that users can be given the ability of passing their privileges to others. Discretionary access control policies are based on authorizations rules. An authorization rule states that a *subject* has the privilege to exercise a given *action* on a given *object*. The kind (and granularity) of subjects, objects, and actions that can be referenced in authorizations may be different in different systems.

Subjects Subjects are the entities to which authorizations can be granted. Typically, subjects are users (i.e., identifiers corresponding to human entities). User *groups* can also be defined to which authorizations can be granted; authorizations granted to a group can be enjoyed by all its members. Discretionary access control can be extended with role-based capabilities allowing the definition of *roles* to which privileges can be granted. Roles are granted to users, and users can dynamically

activate and deactivate the roles received, thereby turning on and off the corresponding privileges. Intuitively, a role identifies a task, and corresponding privileges, that users need to execute to perform organizational activities. While groups are set of users, roles are set of privileges. Note the difference between groups and roles. Groups are static: users cannot enable and disable group memberships (and corresponding privileges) at their will. By contrast, roles are dynamic and can be activated and deactivated upon explicit request by users.

Objects Objects are the entities to be protected. Typically, objects correspond to information container (tables or portion of it) or procedures. In DBMS systems, different granularity levels can be supported spanning from the whole database to the single element (e.g., a specific employee's salary) in it.

Actions Actions define the specific operations that subjects can execute on objects. Actions to be supported include the operations corresponding to the basic *read*, *write*, *delete*, *create*, and *execute*, which can take on different names in relational database systems (for instance, read operations correspond to `SELECT` actions).

Authorizations Authorizations define which accesses are to be allowed. The simplest form of authorization is a triple `(subject,object,action)` specifying that `subject` is authorized to exercise `action` on `object`.

3 Relational databases and SQL

Since DBMS security constraints can be specified at the logical level (authorizations refer to objects of the data model), we need first to define a reference data model. Although different models have been proposed in the literature, the *relational model* is in fact the dominant model and is used in most commercial DBMSs. We therefore limit our discussion to *relational DBMSs* and the related *SQL* language.

3.1 Relational databases

In the relational model, data are organized as a collection of *relations* [4]. Intuitively, a relation can be seen as a table; the columns of the table are called *attributes* and the rows are called *tuples*. Figure 4 illustrates an example of a relational database composed of two relations, `Employee`

and `Department`, containing information about the employees and the departments of an organization. The corresponding *relation schemes*, which show the structure of the database, are `Employee(name,office,salary,dept)` and `Department(dept,location,phone,budget)`. Each relation must have a *primary key*, that is, a minimal set of attributes such that the relation may not have two distinct tuples with the same values for the key. For instance, with respect to the `Employee` relation just described, if we expect hat duplicate names can never occur, attribute `name` can be designated as the primary key. A primary key can also be used to link relations together. As an example, consider the relations in Figure 4. The `dept` attribute of the `Employee` relation is said to be a *foreign key* from the `Employee` relation to the `Department` relation, and its values must match those of the primary key `dept` in the `Department` relation. Note that the relation containing the primary key and the relation containing the foreign key may be the same. For instance, relation `Employee` could be extended to include attribute `supervisor` referencing the relation's primary key `name`.

Primary and foreign keys must satisfy the following *integrity constraints*.

- *Entity integrity*: A primary key can never have null (i.e., undefined) value.
- *Referential integrity*: If a foreign key of a relation R_1 is the primary key of another relation R_2, then, for each tuple in R_1, the value of the foreign key must be either null or equal to the value of the primary key of a tuple in R_2.

Entity integrity guarantees that the value of the primary key can uniquely identify each tuple. Referential integrity guarantees that attributes assume only legitimate values, existing in the real world. For instance, with reference to foreign key `dept` discussed above, the referential integrity constraint avoids employees from being assigned to a non existent department.

4 SQL privileges

An SQL privilege states that a given authorization identifier is authorized for a set of actions on a given SQL object. The actions that can be executed on SQL objects are summarized in Figure 5. The specific set of actions applicable to an object depend on the type of the object. For instance, the `UPDATE` action is applicable to tables, while the `EXECUTE` action is applicable to routines.

Each privilege, which corresponds to the classical notion of an authorization tuple, is represented by a privilege descriptor that includes the following elements:

- the object on which the privilege is granted;
- the authorization identifier of the grantor of the privilege (it is special value `_SYSTEM` for privileges automatically granted to the creator of an object);
- the authorization identifier of the grantee of the privilege;
- the action that the privilege allows;
- an indication of whether or not the privilege is grantable to others;
- an indication of whether or not the privilege has the `WITH HIERARCHY OPTION` specified (see Section 4.1).

An authorization identifier can execute all actions allowed by the set of applicable privileges for it. In particular, the set of user privileges for a user identifier consists of all privileges defined by the privilege descriptors whose grantee is either that user identifier or `PUBLIC`. Analogously, the set of role privileges for a role name consists of all privileges defined by the privilege descriptors whose grantee is either that role name, `PUBLIC`, or one of the applicable roles of that role name.

4.1 Privilege management

The creator of an object in a database is its owner and can perform any action on the object (the particular cases where the object is a view or a routine are treated in Section 5). Also, it can grant and revoke to others the privileges on the object as well as delegate to others the capability of granting and revoking the privilege (and its administration).

Privilege granting Privileges are granted by using the `GRANT` statement, whose syntax is illustrated in Figure 6. Value `ALL PRIVILEGES` can be used in place of a specific action to denote all of the actions on <object name> for which the grantor has grantable privilege. If the `WITH HIERARCHY OPTION` clause is present, then the action must be `SELECT` and the object must be a table. This clause provides the grantee with `SELECT` privileges on all subtables (either already existing or that may be defined at a later time) of the table on which the privilege is granted. With the `WITH GRANT OPTION` clause the grantee receives, beside the privileges, the authority to grant the privilege (with or without grant option) to others.

As an example, suppose that `Alice` is the owner of the tables in Figure 4. She can grant any action on these tables to any authorization identifier in the SQL environment. A possible sequence of `GRANT` statements is as follows:

By Alice: `GRANT UPDATE` (salary) `ON TABLE` Employee `TO` Bob

By Alice: `GRANT SELECT ON TABLE` Department `TO` Eve `WITH GRANT OPTION`

By Eve: `GRANT SELECT ON TABLE` Department `TO` Dave `WITH GRANT OPTION`

By Eve: `GRANT SELECT ON TABLE` Department `TO` Peggy

According to these statements, `Alice` grants the authorization to modify the `salary` column of the `Employee` table to `Bob`. `Alice` also grants the authorization to retrieve rows of the `Department` table with the grant option to `Eve`. Since `Eve` holds the grant option, she can grant the privilege to others. She grants the `SELECT` privilege on `Department` to `Dave` and to `Peggy`. The grant option associated with the grant statement for `Dave` allows him to further propagate the privilege.

Privileges granted through grant statements are represented by privilege descriptors as discussed in Section 4. The fact that a privilege descriptor is used to allow the granting of another descriptor (as in the case where `Eve` grants the privilege to `Dave` and `Peggy`) introduces a dependency relationship between descriptors. A privilege descriptor p_2 is said to *directly depend* on a privilege descriptor p_1 if all the following conditions hold: *i)* the privilege represented by p_1 is grantable; *ii)* p_1 and p_2 have the same action and object; and *iii)* the grantee of p_1 is either the same as the grantor of p_2 or is `PUBLIC`, or, if the grantor of p_2 is a role name, the grantee of p_1 belongs to the set of applicable roles of the grantor of p_2. Depending on the type of the involved objects, SQL:1999 defines also other notions of dependency; the in-depth discussion of these notions is far beyond the scope of this chapter (we refer the reader to [1] for a more comprehensive treatment). Privilege descriptors, and dependencies between them, can be represented as a graph called *privilege dependency graph*. The privilege dependency graph is a directed graph such that each node represent a privilege descriptor and each arc from node p_1 to node p_2 represents the fact that p_2 directly depends on p_1. Figure 7(a) illustrates an example of privilege dependency graph for the `SELECT` privilege on table `Department`, where, for the sake of clarity, nodes are labeled with the grantee identifier.

Privilege revocation The `REVOKE` statement allows authorization identifiers to revoke privileges they previously granted (note that an authorization identifier can revoke only privileges granted by itself). The SQL syntax for the `REVOKE` statement is illustrated in Figure 6. A `REVOKE` statement

identifies the set of all privilege descriptors where the action and object are equal to those specified (explicitly or implicitly) in <action> and <object name>, respectively, the grantor is <grantor>, if specified, or coincides with the non-null element in the current pair $\langle uid, rid \rangle$, and the grantee is <grantee>. For instance, the `REVOKE` statement

```
REVOKE SELECT ON Department
FROM Mallory, Walter
CASCADE
```

executed by `Dave` identifies the privilege descriptors corresponding to the nodes labeled `Mallory` and `Walter` in Figure 7(a). The privilege dependency graph after revocation is as illustrated in Figure 7(b).

5 Views and routines

In addition to using the privileges described above to provide specific types of access to entire objects, there is also a mechanism within SQL to provide content-based access restrictions. This mechanism is represented by *views*, which define a convenient way to refer to specific portions of tables. A view is a virtual table derived from base tables and/or other views. A view is specified by a *view definition* that is written in SQL. The database stores the view definitions and materializes the views as needed.

A view can contain a subset of the data (columns and/or rows) in the tables on which the view is defined. A view can also contain data derived from them or representing an aggregation of them, using the *aggregate functions* supported by SQL (e.g., `SUM`, `MIN`, `MAX`). Examples of view creation statements are as follows:

```
CREATE VIEW R&D(name,salary) AS
  SELECT   name, salary
  FROM     Employee
  WHERE    dept = 'Research & Development'
```

```
CREATE VIEW MaxSalary(dept,salary) AS
  SELECT     dept,MAX(salary)
  FROM       Employee
  GROUP BY   dept
```

View `R&D` is defined to represent `name` and `office` attributes of the tuples in the `Employee` table that satisfy `dept` = "Research & Development" predicate. By assuming table `Employee` as in Figure 4, the view will contain the single tuple `(Alice,15)`. View `MaxSalary` uses data aggregation functions to report the list of departments having employees and the maximum salary paid by each department.

A user/role can create a view only if it has permission to access *all* the tables (views or base tables) directly referenced by the view (i.e., appearing in its definition). The creator receives on the view the privileges that it holds on all the tables directly refenced. Also, it receives the grant option for a privilege only if it has the grant option for the privilege on all the tables directly referenced by the view. If it holds a privilege on the view with the grant option, the creator can grant the privilege (and the grant option) to others. The grantees of such privileges need not hold the privileges on the underlying tables to access the view; access to the view only requires the existence of privileges for the view. Views provide therefore a convenient way to enforce selective access on tables not accessible directly.

For retrieving purposes, there is no distinction between views and base tables. Therefore, views provide a powerful mechanism for controlling what information can be retrieved. When updates are considered, views and base tables must be treated differently. In general, users cannot directly update views, particularly when they are created from the combination (join) of two or more base tables. The updatable views can instead appear as targets of statements that change SQL data. The results of changes expressed in this way are defined in terms of corresponding changes to base tables.

Like views, routines also provide an effective way to enforce access restrictions. The creator of a routine is granted the `EXECUTE` privilege on the routine only if it holds all the privileges necessary for the routine to run successfully. The `EXECUTE` privilege is granted with the grant option if the creator has the grant option for all the privileges necessary for the routine to run. If the creator of a routine holds the `EXECUTE` privilege with the grant option, it can grant to others the privilege (and the grant option on it). Like for views, the grantees need not hold the privileges necessary for the routine to run in order to execute it (intuitively these accesses are checked against the privileges of the routine creator).

6 Integrity constraints

Providing integrity means ensuring that the data stored in the database correctly reflect the real world, that is, they have not been modified by unauthorized users or in an improper way. Integrity requires the enforcement of different controls and principles, including treatment of *transactions*. A transaction is a sequence of database actions for which the following *ACID* properties must be ensured:

- *Atomicity*: a transaction is either performed in its entirety or not performed at all;
- *Consistency*: a transaction must preserve the consistency of the database;
- *Isolation*: a transaction should not make its updates visible to other transactions until it is committed;
- *Durability*: changes made by a transaction that has committed must never be lost because of subsequent failures.

Satisfaction of the properties above requires application of specific controls (e.g., *concurrency control*) in transaction processing by the DBMS.

In this section we focus of *integrity constraints* that define what it means for a database to be *consistent* and that can be specified through the DBMS manipulation language. Integrity constraints, generally referred to simply as constraints, define the valid states of data in the database by constraining the values in the base tables. The checking of constraints can be performed at the end of an SQL statement or deferred until the end of transaction. If a constraint is not satisfied, an exception is raised and the SQL statement that caused the constraint to be checked has no effect. SQL:1999 defines different types of constraints: *table constraints*, *domain constraints*, and *assertions*. Figure 8 illustrates the general syntax of the SQL statements used to define constraints.

Table constraints A table constraint is either a *unique constraint*, a *referential constraint* or a *table check constraint* which may be specified for tables.

- A *unique constraint* specifies one or more columns of the table as unique columns, that is, no two rows in a table have the same non-null values in the unique columns. To illustrate, consider the following table definition

```
CREATE TABLE Employee
(name    CHAR(20)
office   SMALLINT
salary   DECIMAL(9,2)
dept     CHAR(20))
UNIQUE (office)
PRIMARY KEY (name))
```

The unique constraint associated with attribute `office` states that no value can be present more than once in the `office` column (intuitively, offices cannot be shared). Note that, as discussed in Section 3.1 the primary key definition implicitely imposes a unicity constraint on the primary key attribute.

- A *referential constraint* specifies one or more columns (called *foreign key*) in a table T as referencing columns and corresponding referenced columns in some (not necessarily distinct) base table S. For instance, the SQL statement:

  ```
  ALTER TABLE Employee
  ADD FOREIGN KEY (dept)
  REFERENCES Department(dept))
  ```

  modifies the definition of the `Employee` table given above by adding a referential constraint stating that the `dept` column in the `Employee` table is a foreign key referencing the `dept` column in the `Department` table.

  As already discussed in Section 3.1, if the foreign key for a row of T is non null, then a row in S must exist such that all the corresponding columns match. If null values are present, satisfaction of the referential constraint depends on the treatment specified for nulls (known as the *match type*).

- A *table check constraint* specifies a search condition which all rows in the table must satisfy. For instance, attribute office in the `Employee` table above could be defined with associated constraint

  ```
  CHECK (office BETWEEN 10 AND 100)
  ```

  restricting to the range 10 to 100 the possible values for attribute office.

  A table check constraint is considered violated if the result of the search condition is false for any row of the table. Note that the constraint is violated if the result is unknown.

Domain constraints A domain constraint is a constraint that is specified for a domain. It is similar to table check constraint above but is defined on a domain instead of a specific column. The constraint is then enforced on all the columns defined on the domains, and to all values cast to that domain.

Assertions An assertion is a named constraint that may relate to the content of individual rows of a table, to the entire contents of a table, or to a state required to exist among a number of tables. Examples of assertions are as follows.

```
CREATE ASSERTION max_employee
CHECK ((SELECT COUNT(*)
        FROM Employee ) < 10000)

CREATE ASSERTION max_salary_employee
CHECK ((SELECT SUM(salary)
        FROM Employee) < 0.8*(SELECT SUM(budget)
                              FROM Department))
```

The first assertion states that the total number of employees must be less that 10000 while the second one states that the total sum of employees' salary must be less that 80% of the budget.

In every SQL-session, every constraint has a constraint mode that is a property of that SQL-session. Any type of constraint can be declared to be `DEFERRABLE` or `NOT DEFERRABLE`; if it is `DEFERRABLE`, it can further be declared to be `INITIALLY DEFERRED` or `INITIALLY IMMEDIATE`, which defined its state at the beginning of transactions (<constraint characteristics>). The constraint state can be set by the SQL statement "`SET CONSTRAINTS` <constraint name list> { `DEFERRED` | `IMMEDIATE` }", provided the constraint is deferrable. When a transaction is initiated, the constraint mode of each constraint is set to its default. On completion of execution of every SQL-statement, every constraint is checked whose constraint mode is immediate.

7 Multilevel Secure Databases

Discretionary access controls have the limitation that they can be easily circumvented by malicious users. TO illustrate, if the user `Alice` has been granted the `SELECT` privilege without the grant option on a particular relation, then she should not be able to grant this privilege to other users. She can easily subvert this intent by making a copy of the relation in question, and as owner can give `SELECT` privilege on the copy to others.

Even if users are trusted not to violate the security in this way, a malicious user can imbed Trojan horses that can do so [3]. Multilevel secure databases enforce mandatory access controls to help eliminate these problems.

In a multilevel system, access controls are based on the security labels associated with each user and each data item. Security labels have two components: a hierarchical component (such as TOP SECRET, SECRET, CONFIDENTIAL, and UNCLASSIFIED, listed in decreasing order of sensitivity) and a set of categories or compartments (such as NUCLEAR, NATO, CONVENTIONAL)

which could be empty. The security labels are partially ordered as follows: A label X dominates label Y if the hiearchical component of X is greater than or equal to that of Y and if the categories of X contain the categories of Y.

The security policy requires that a user may have access to a data item if the security label of the user dominates that of the data item. Thus, a user with a SECRET clearance can have access to a relation with CONFIDENTIAL data, but not a relation with TOP SECRET data.

There are multitudes of problems that must be addressed to make multilevel secure DBMSs commercially viable. One of the major difficulties is that in a mulilevel secure system, data values may be *polyinstantiated*, allowing users with different clearance levels to have different views of the same object. Refer to Abrams et al. [2] for additional details.

8 Inference Problems

An inference problem arises when users can derive some unauthorized information from the information they legally obtain from the database. Many of the inference problems that arise are due to combining data retrieved from the database with external knowledge of the users.

The US Census Bureau is particularly concerned about the inference problem since by law it must release to the public aggregate statistics on groups of individuals without releasing information about particular individuals [5]. Another commonly cited example is the US Department of Defense phone book; the entire phone book is considered sensitive but individual numbers are not.

References

[1] Database Language SQL – Part 2: Foundation (SQL/Foundation). ISO International Standard, ISO/IEC 9075:1999, 1999.

[2] Marshall D. Abrams, Sushil Jajodia, and Harold J. Podell, eds. *Information Security: An Integrated Collection of Essays.* IEEE Computer Society Press, Los Alamitos, CA, 1995.

[3] S. Castano, M.G. Fugini, G. Martella, and P. Samarati. *Database Security.* Addison Wesley Publishing Company, 1995.

[4] E.F. Codd. A Relational Model for Large Shared Data Banks. *Communications of the ACM*, 13(6):377–387, June 1970.

[5] Dorothy E. Denning. *Cryptography and Data Security.* Addison-Wesley, Reading, MA, 1982.

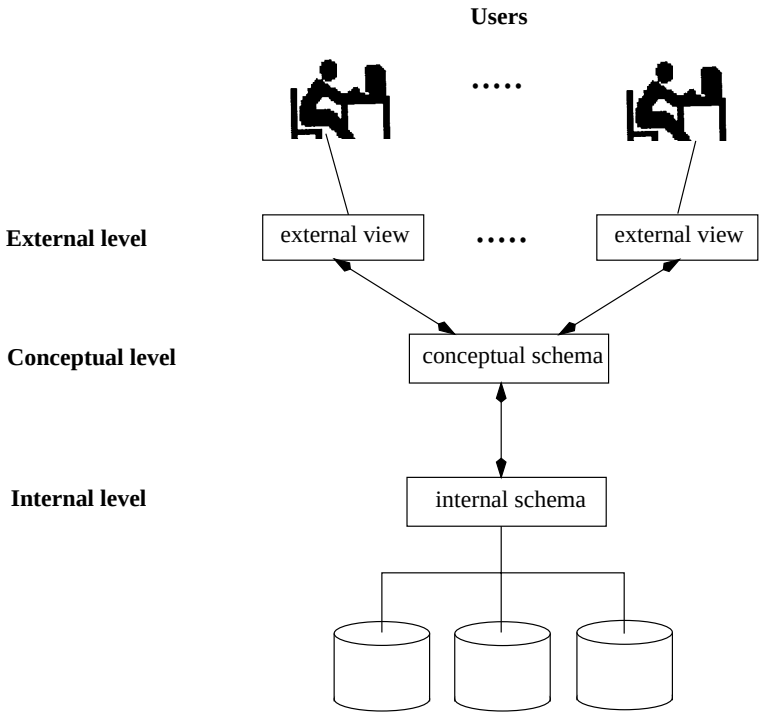

Figure 1: Three-level architecture

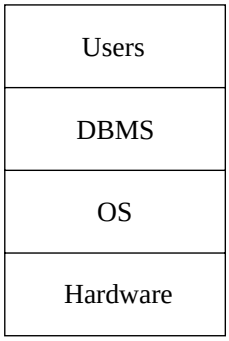

Figure 2: DBMS/OS high-level architecture

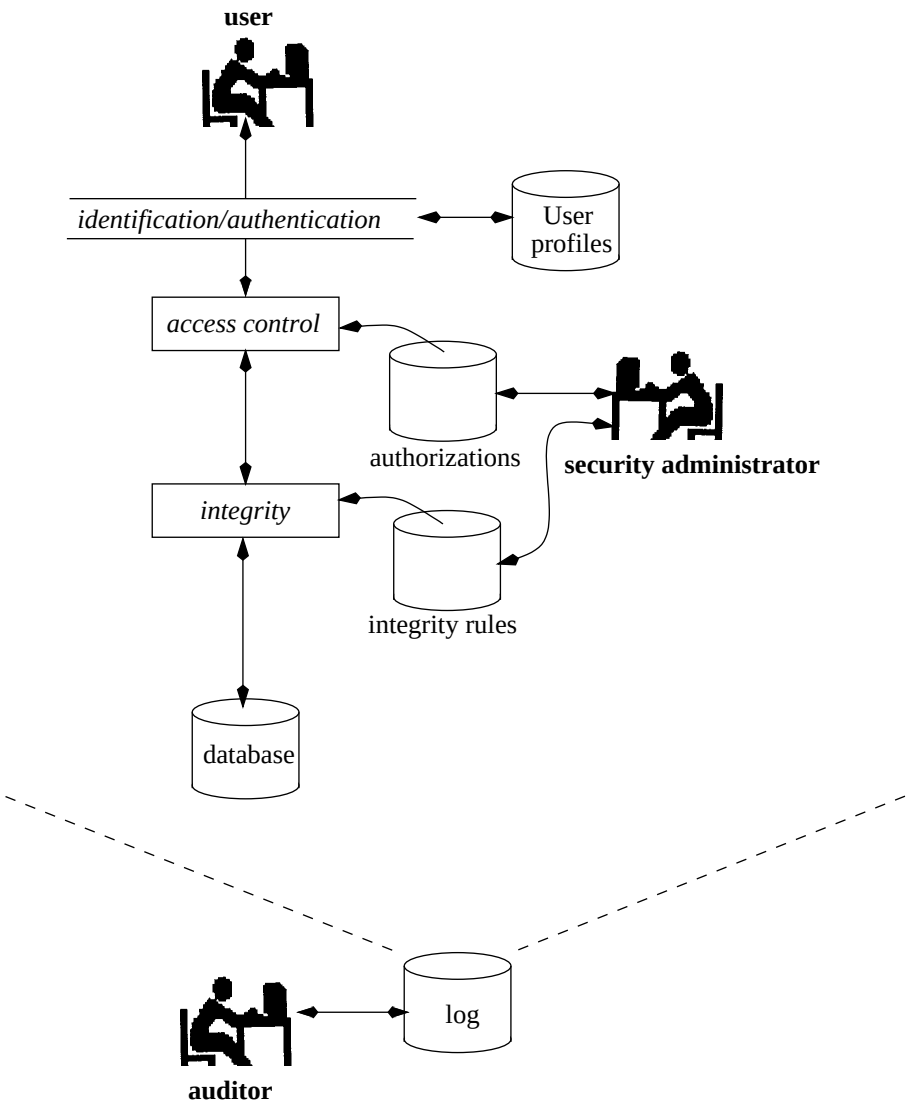

Figure 3: Security mechanisms

Employee

name	office	salary	dept
Alice	15	$70,000	Research & Development
Eve	5	$50,000	Computer Science
Dave	22	$65,000	Electrical Engineering

Department

dept	location	phone	budget
Research & Development	South Street	555-789-123	$555,000
Computer Science	Main Street	555-456-789	$1,500,000
Electrical Engineering	Park Street	555-908-345	$350,000

Figure 4: A simple example of a relational database

Privilege	Object	Description
SELECT	table column name list, or privilege method list	retrieve values from the specified object
INSERT	table or column name list	insert rows in the specified object
UPDATE	table or column name list	modify the data within the specified object
DELETE	table	delete rows from the specified object
TRIGGER	table	define a trigger for the specified object
REFERENCES	table or column name list	specify a foreign key reference from the specified object schema to the foreign key of another object
USAGE	domain, user-defined type, character set, collation, or translation	allow the usage of the specified object
UNDER	structured type	define a subtype of the specified object
EXECUTE	SQL-invoked routine	execute the specified object

Figure 5: SQL privileges

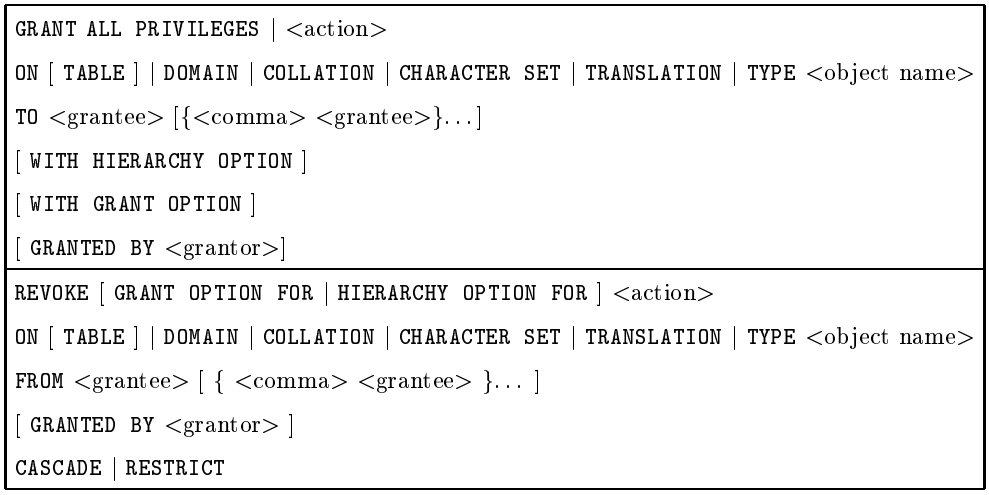

Figure 6: Syntax of the SQL statements for assigning privileges

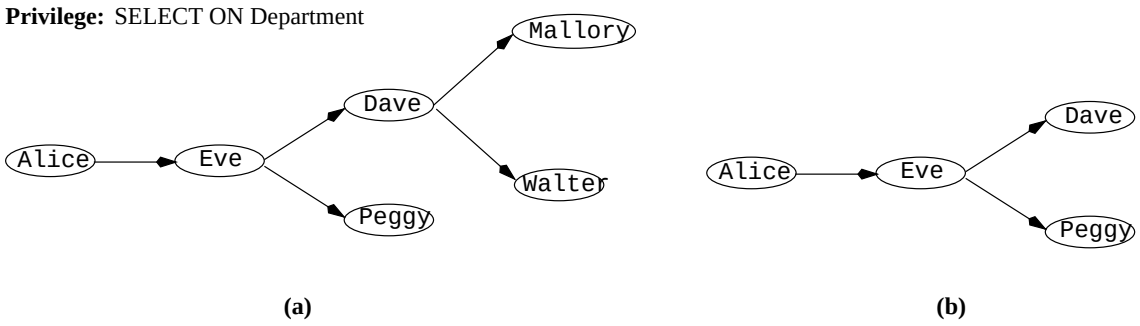

Figure 7: An example of privilege dependency graph (a) and the graph after a `REVOKE` statement

Table	**Unique**	[`CONSTRAINT` <constraint name>] `UNIQUE` | `PRIMARY KEY` (<unique column list>) | `UNIQUE` (`VALUE`) [<constraint characteristics>]
	Referential	[`CONSTRAINT` <constraint name>] `FOREIGN KEY` (<referencing columns>) `REFERENCES` <referenced table and columns> [`MATCH` <match type>] [<referential triggered action>] [<constraint characteristics>]
	Table check	[`CONSTRAINT` <constraint name>] `CHECK` (<search condition>) [<constraint characteristics>]
Domain		[`CONSTRAINT` <constraint name>] `CHECK` (<search condition>) [<constraint characteristics>]
Assertion		`CREATE ASSERTION` <constraint name> `CHECK` (<search condition>) [<constraint characteristics>]

Figure 8: Syntax of the SQL statements for defining constraints