CHAPTER

7 SQL: Data Definition

Chapter Objectives

In this chapter you will learn:

- The data types supported by the SQL standard.
- The purpose of the integrity enhancement feature of SQL.
- How to define integrity constraints using SQL, including:
  - required data;
  - domain constraints;
  - entity integrity;
  - referential integrity;
  - general constraints.
- How to use the integrity enhancement feature in the CREATE and ALTER TABLE statements.
- The purpose of views.
- How to create and delete views using SQL.
- How the DBMS performs operations on views.
- Under what conditions views are updatable.
- The advantages and disadvantages of views.
- How the ISO transaction model works.
- How to use the GRANT and REVOKE statements as a level of security.

In the previous chapter we discussed in some detail SQL and, in particular, the SQL data manipulation facilities. In this chapter we continue our presentation of SQL and examine the main SQL data definition facilities.

Structure of this Chapter In Section 7.1 we examine the ISO SQL data types. The 1989 ISO standard introduced an Integrity Enhancement Feature (IEF), which provides facilities for defining referential integrity and other constraints (ISO, 1989). Prior to this standard, it was the responsibility of each application program to ensure compliance with these constraints. The provision of an IEF greatly enhances the functionality of SQL and allows constraint checking to be centralized and standardized. We consider the IEF in Section 7.2 and the main SQL data definition facilities in Section 7.3.

In Section 7.4 we show how views can be created using SQL, and how the DBMS converts operations on views into equivalent operations on the base tables. We also discuss the restrictions that the ISO SQL standard places on views in order for them to be updatable. In Section 7.5, we briefly describe the ISO SQL transaction model.

Views provide a certain degree of database security. SQL also provides a separate access control subsystem, containing facilities to allow users to share database objects or, alternatively, to restrict access to database objects. We discuss the access control subsystem in Section 7.6.

In Section 29.4 we examine in some detail the features that have recently been added to the SQL specification to support object-oriented data management. In Appendix I we discuss how SQL can be embedded in high-level programming languages to access constructs that until recently were not available in SQL. As in the previous chapter, we present the features of SQL using examples drawn from the *DreamHome* case study. We use the same notation for specifying the format of SQL statements as defined in Section 6.2.

7.1 The ISO SQL Data Types

In this section we introduce the data types defined in the SQL standard. We start by defining what constitutes a valid identifier in SQL.

7.1.1 SQL Identifiers

SQL identifiers are used to identify objects in the database, such as table names, view names, and columns. The characters that can be used in a user-defined SQL identifier must appear in a **character set**. The ISO standard provides a default character set, which consists of the uppercase letters A . . . Z, the lowercase letters a . . . z, the digits 0 . . . 9, and the underscore (_) character. It is also possible to specify an alternative character set. The following restrictions are imposed on an identifier:

- an identifier can be no longer than 128 characters (most dialects have a much lower limit than this);
- an identifier must start with a letter;
- an identifier cannot contain spaces.

TABLE 7.1 ISO SQL data types.

DATA TYPE	DECLARATIONS			
boolean	BOOLEAN			
character	CHAR	VARCHAR		
bit†	BIT	BIT VARYING		
exact numeric	NUMERIC	DECIMAL	INTEGER	SMALLINT
approximate numeric	FLOAT	REAL	DOUBLE PRECISION	
datetime	DATE	TIME	TIMESTAMP	
interval	INTERVAL			
large objects	CHARACTER LARGE OBJECT		BINARY LARGE OBJECT	

†BIT and BIT VARYING have been removed from the SQL:2003 standard.

© CourseSmart

7.1.2 SQL Scalar Data Types

Table 7.1 shows the SQL scalar data types defined in the ISO standard. Sometimes, for manipulation and conversion purposes, the data types *character* and *bit* are collectively referred to as **string** data types, and *exact numeric* and *approximate numeric* are referred to as **numeric** data types, as they share similar properties. The SQL standard now also defines both character large objects and binary large objects, although we defer discussion of these data types until Section 29.4.

Boolean data

Boolean data consists of the distinct truth values TRUE and FALSE. Unless prohibited by a NOT NULL constraint, boolean data also supports the UNKNOWN truth value as the NULL value. All boolean data type values and SQL truth values are mutually comparable and assignable. The value TRUE is greater than the value FALSE, and any comparison involving the NULL value or an UNKNOWN truth value returns an UNKNOWN result.

Character data

Character data consists of a sequence of characters from an implementation-defined character set, that is, it is defined by the vendor of the particular SQL dialect. Thus, the exact characters that can appear as data values in a character type column will vary. ASCII and EBCDIC are two sets in common use today. The format for specifying a character data type is:

CHARACTER [VARYING] [length]
CHARACTER can be abbreviated to **CHAR** and
CHARACTER VARYING to **VARCHAR**.

When a character string column is defined, a length can be specified to indicate the maximum number of characters that the column can hold (default length is 1). A character string may be defined as having a **fixed** or **varying** length. If the string

is defined to be a fixed length and we enter a string with fewer characters than this length, the string is padded with blanks on the right to make up the required size. If the string is defined to be of a varying length and we enter a string with fewer characters than this length, only those characters entered are stored, thereby using less space. For example, the branch number column branchNo of the Branch table, which has a fixed length of four characters, is declared as:

branchNo **CHAR**(4)

The column address of the PrivateOwner table, which has a variable number of characters up to a maximum of 30, is declared as:

address **VARCHAR**(30)

Bit data

The bit data type is used to define bit strings, that is, a sequence of binary digits (bits), each having either the value 0 or 1. The format for specifying the bit data type is similar to that of the character data type:

BIT [**VARYING**] [length]

For example, to hold the fixed length binary string "0011", we declare a column *bitString*, as:

bitString **BIT**(4)

7.1.3 Exact Numeric Data

The exact numeric data type is used to define numbers with an exact representation. The number consists of digits, an optional decimal point, and an optional sign. An exact numeric data type consists of a **precision** and a **scale**. The precision gives the total number of significant decimal digits, that is, the total number of digits, including decimal places but excluding the point itself. The scale gives the total number of decimal places. For example, the exact numeric value –12.345 has precision 5 and scale 3. A special case of exact numeric occurs with integers. There are several ways of specifying an exact numeric data type:

NUMERIC [precision [, scale]]
DECIMAL [precision [, scale]]
INTEGER
SMALLINT

INTEGER can be abbreviated to **INT** and **DECIMAL** to **DEC**

NUMERIC and DECIMAL store numbers in decimal notation. The default scale is always 0; the default precision is implementation-defined. INTEGER is used for large positive or negative whole numbers. SMALLINT is used for small positive or negative whole numbers. By specifying this data type, less storage space can be reserved for the data. For example, the maximum absolute value that can be stored with SMALLINT might be 32 767. The column rooms of the PropertyForRent table, which represents the number of rooms in a property, is obviously a small integer and can be declared as:

rooms **SMALLINT**

The column salary of the Staff table can be declared as:

salary **DECIMAL**(7,2)

which can handle a value up to 99,999.99.

Approximate numeric data

The approximate numeric data type is used for defining numbers that do not have an exact representation, such as real numbers. Approximate numeric, or floating point, notation is similar to scientific notation, in which a number is written as a *mantissa* times some power of ten (the *exponent*). For example, 10E3, +5.2E6, −0.2E–4. There are several ways of specifying an approximate numeric data type:

FLOAT [precision]
REAL
DOUBLE PRECISION

The *precision* controls the precision of the mantissa. The precision of REAL and DOUBLE PRECISION is implementation-defined.

Datetime data

The datetime data type is used to define points in time to a certain degree of accuracy. Examples are dates, times, and times of day. The ISO standard subdivides the datetime data type into YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, TIMEZONE_HOUR, and TIMEZONE_MINUTE. The latter two fields specify the hour and minute part of the time zone offset from Universal Coordinated Time (which used to be called Greenwich Mean Time). Three types of datetime data type are supported:

DATE
TIME [timePrecision] [**WITH TIME ZONE**]
TIMESTAMP [timePrecision] [**WITH TIME ZONE**]

DATE is used to store calendar dates using the YEAR, MONTH, and DAY fields. TIME is used to store time using the HOUR, MINUTE, and SECOND fields. TIMESTAMP is used to store date and times. The *timePrecision* is the number of decimal places of accuracy to which the SECOND field is kept. If not specified, TIME defaults to a precision of 0 (that is, whole seconds), and TIMESTAMP defaults to 6 (that is, microseconds). The WITH TIME ZONE keyword controls the presence of the TIMEZONE_HOUR and TIMEZONE_MINUTE fields. For example, the column date of the Viewing table, which represents the date (year, month, day) that a client viewed a property, is declared as:

viewDate **DATE**

Interval data

The interval data type is used to represent periods of time. Every interval data type consists of a contiguous subset of the fields: YEAR, MONTH, DAY, HOUR,

TABLE 7.2 ISO SQL scalar operators.

OPERATOR	MEANING
BIT_LENGTH	Returns the length of a string in bits. For example, **BIT_LENGTH**(X'FFFF') returns 16.
OCTET_LENGTH	Returns the length of a string in octets (bit length divided by 8). For example, **OCTET_LENGTH**(X'FFFF') returns 2.
CHAR_LENGTH	Returns the length of a string in characters (or octets, if the string is a bit string). For example, **CHAR_LENGTH**('Beech') returns 5.
CAST	Converts a value expression of one data type into a value in another data type. For example, **CAST**(5.2E6 AS INTEGER).
||	Concatenates two character strings or bit strings. For example, fName || lName.
CURRENT_USER or **USER**	Returns a character string representing the current authorization identifier (informally, the current user name).
SESSION_USER	Returns a character string representing the SQL-session authorization identifier.
SYSTEM_USER	Returns a character string representing the identifier of the user who invoked the current module.
LOWER	Converts uppercase letters to lowercase. For example, **LOWER**(**SELECT** fName **FROM** Staff **WHERE** staffNo = 'SL21') returns 'john'.
UPPER	Converts lower-case letters to upper-case. For example, **UPPER**(**SELECT** fName **FROM** Staff **WHERE** staffNo = 'SL21') returns 'JOHN'.
TRIM	Removes leading (**LEADING**), trailing (**TRAILING**), or both leading and trailing (**BOTH**) characters from a string. For example, **TRIM**(**BOTH** '*' **FROM** '*** Hello World ***') returns 'Hello World'.
POSITION	Returns the position of one string within another string. For example, **POSITION**('ee' **IN** 'Beech') returns 2.
SUBSTRING	Returns a substring selected from within a string. For example, **SUBSTRING**('Beech' **FROM** 1 **TO** 3) returns the string 'Bee'.
CASE	Returns one of a specified set of values, based on some condition. For example, **CASE** type **WHEN** 'House' **THEN** 1 **WHEN** 'Flat' **THEN** 2 **ELSE** 0 **END**
CURRENT_DATE	Returns the current date in the time zone that is local to the user.
CURRENT_TIME	Returns the current time in the time zone that is the current default for the session. For example, **CURRENT_TIME**(6) gives time to microseconds precision.
CURRENT_TIMESTAMP	Returns the current date and time in the time zone that is the current default for the session. For example, **CURRENT_TIMESTAMP**(0) gives time to seconds precision.
EXTRACT	Returns the value of a specified field from a datetime or interval value. For example, **EXTRACT**(**YEAR FROM** Registration.dateJoined).

MINUTE, SECOND. There are two classes of interval data type: **year–month** intervals and **day–time** intervals. The year–month class may contain only the YEAR and/or the MONTH fields; the day–time class may contain only a contiguous selection from DAY, HOUR, MINUTE, SECOND. The format for specifying the interval data type is:

INTERVAL {{startField **TO** endField} singleDatetimeField}
startField = **YEAR | MONTH | DAY | HOUR | MINUTE**
[(intervalLeadingFieldPrecision)]
endField = **YEAR | MONTH | DAY | HOUR | MINUTE | SECOND**
[(fractionalSecondsPrecision)]
singleDatetimeField = startField | **SECOND**
[(intervalLeadingFieldPrecision [, fractionalSecondsPrecision])]

In all cases, *startField* has a leading field precision that defaults to 2. For example:

INTERVAL YEAR(2) **TO MONTH**

represents an interval of time with a value between 0 years 0 months, and 99 years 11 months. In addition,

INTERVAL HOUR TO SECOND(4)

represents an interval of time with a value between 0 hours 0 minutes 0 seconds and 99 hours 59 minutes 59.9999 seconds (the fractional precision of second is 4).

Scalar operators

SQL provides a number of built-in scalar operators and functions that can be used to construct a scalar expression, that is, an expression that evaluates to a scalar value. Apart from the obvious arithmetic operators (+, –, *, /), the operators shown in Table 7.2 are available.

7.2 Integrity Enhancement Feature

In this section, we examine the facilities provided by the SQL standard for integrity control. Integrity control consists of constraints that we wish to impose in order to protect the database from becoming inconsistent. We consider five types of integrity constraint (see Section 4.3):

- required data;
- domain constraints;
- entity integrity;
- referential integrity;
- general constraints.

These constraints can be defined in the CREATE and ALTER TABLE statements, as we will explain shortly.

7.2.1 Required Data

Some columns must contain a valid value; they are not allowed to contain nulls. A null is distinct from blank or zero, and is used to represent data that is either not available, missing, or not applicable (see Section 4.3.1). For example, every member of staff must have an associated job position (for example, Manager, Assistant, and so on). The ISO standard provides the **NOT NULL** column specifier in the CREATE and ALTER TABLE statements to provide this type of constraint. When NOT NULL is specified, the system rejects any attempt to insert a null in the column. If NULL is specified, the system accepts nulls. The ISO default is NULL. For example, to specify that the column position of the Staff table cannot be null, we define the column as:

position **VARCHAR**(10) **NOT NULL**

7.2.2 Domain Constraints

Every column has a domain; in other words, a set of legal values (see Section 4.2.1). For example, the sex of a member of staff is either 'M' or 'F', so the domain of the column sex of the Staff table is a single character string consisting of either 'M' or 'F'. The ISO standard provides two mechanisms for specifying domains in the CREATE and ALTER TABLE statements. The first is the **CHECK** clause, which allows a constraint to be defined on a column or the entire table. The format of the CHECK clause is:

CHECK (searchCondition)

In a column constraint, the CHECK clause can reference only the column being defined. Thus, to ensure that the column sex can be specified only as 'M' or 'F', we could define the column as:

sex **CHAR NOT NULL CHECK** (sex **IN** ('M', 'F'))

However, the ISO standard allows domains to be defined more explicitly using the **CREATE DOMAIN** statement:

CREATE DOMAIN DomainName [**AS**] dataType
[**DEFAULT** defaultOption]
[**CHECK** (searchCondition)]

A domain is given a name, *DomainName*, a data type (as described in Section 7.1.2), an optional default value, and an optional CHECK constraint. This is not the complete definition, but it is sufficient to demonstrate the basic concept. Thus, for the previous example, we could define a domain for sex as:

CREATE DOMAIN SexType **AS CHAR**
DEFAULT 'M'
CHECK (**VALUE IN** ('M', 'F'));

This definition creates a domain SexType that consists of a single character with either the value 'M' or 'F'. When defining the column sex, we can now use the domain name SexType in place of the data type CHAR:

sex SexType **NOT NULL**

The *searchCondition* can involve a table lookup. For example, we can create a domain BranchNumber to ensure that the values entered correspond to an existing branch number in the Branch table, using the statement:

```
CREATE DOMAIN BranchNumber AS CHAR(4)
    CHECK (VALUE IN (SELECT branchNo FROM Branch));
```

The preferred method of defining domain constraints is using the CREATE DOMAIN statement. Domains can be removed from the database using the DROP DOMAIN statement:

```
DROP DOMAIN DomainName [RESTRICT | CASCADE]
```

The drop behavior, RESTRICT or CASCADE, specifies the action to be taken if the domain is currently being used. If RESTRICT is specified and the domain is used in an existing table, view, or assertion definition (see Section 7.2.5), the drop will fail. In the case of CASCADE, any table column that is based on the domain is automatically changed to use the domain's underlying data type, and any constraint or default clause for the domain is replaced by a column constraint or column default clause, if appropriate.

7.2.3 Entity Integrity

The primary key of a table must contain a unique, nonnull value for each row (see Section 4.3.2). For example, each row of the PropertyForRent table has a unique value for the property number propertyNo, which uniquely identifies the property represented by that row. The ISO standard supports entity integrity with the PRIMARY KEY clause in the CREATE and ALTER TABLE statements. For example, to define the primary key of the PropertyForRent table, we include the following clause:

```
PRIMARY KEY(propertyNo)
```

To define a composite primary key, we specify multiple column names in the PRIMARY KEY clause, separating each by a comma. For example, to define the primary key of the Viewing table, which consists of the columns clientNo and propertyNo, we include the clause:

```
PRIMARY KEY(clientNo, propertyNo)
```

The PRIMARY KEY clause can be specified only once per table. However, it is still possible to ensure uniqueness for any alternate keys in the table using the keyword **UNIQUE**. Every column that appears in a UNIQUE clause must also be declared as NOT NULL. There may be as many UNIQUE clauses per table as required. SQL rejects any INSERT or UPDATE operation that attempts to create a duplicate value within each candidate key (that is, primary key or alternate key). For example, with the Viewing table we could also have written:

```
clientNo        VARCHAR(5)      NOT NULL,
propertyNo      VARCHAR(5)      NOT NULL,
UNIQUE (clientNo, propertyNo)
```

7.2.4 Referential Integrity

A foreign key is a column, or set of columns, that links each row in the child table containing the foreign key to the row of the parent table containing the matching candidate key value. Referential integrity means that, if the foreign key contains a value, that value must refer to an existing, valid row in the parent table (see Section 4.3.3). For example, the branch number column branchNo in the PropertyForRent table links the property to that row in the Branch table where the property is assigned. If the branch number is not null, it must contain a valid value from the column branchNo of the Branch table, or the property is assigned to an invalid branch office.

The ISO standard supports the definition of foreign keys with the FOREIGN KEY clause in the CREATE and ALTER TABLE statements. For example, to define the foreign key branchNo of the PropertyForRent table, we include the clause:

FOREIGN KEY(branchNo) **REFERENCES** Branch

SQL rejects any INSERT or UPDATE operation that attempts to create a foreign key value in a child table without a matching candidate key value in the parent table. The action SQL takes for any UPDATE or DELETE operation that attempts to update or delete a candidate key value in the parent table that has some matching rows in the child table is dependent on the **referential action** specified using the ON UPDATE and ON DELETE subclauses of the FOREIGN KEY clause. When the user attempts to delete a row from a parent table, and there are one or more matching rows in the child table, SQL supports four options regarding the action to be taken:

- CASCADE: Delete the row from the parent table and automatically delete the matching rows in the child table. Because these deleted rows may themselves have a candidate key that is used as a foreign key in another table, the foreign key rules for these tables are triggered, and so on in a cascading manner.
- SET NULL: Delete the row from the parent table and set the foreign key value(s) in the child table to NULL. This option is valid only if the foreign key columns do not have the NOT NULL qualifier specified.
- SET DEFAULT: Delete the row from the parent table and set each component of the foreign key in the child table to the specified default value. This option is valid only if the foreign key columns have a DEFAULT value specified (see Section 7.3.2).
- NO ACTION: Reject the delete operation from the parent table. This is the default setting if the ON DELETE rule is omitted.

SQL supports the same options when the candidate key in the parent table is updated. With CASCADE, the foreign key value(s) in the child table are set to the new value(s) of the candidate key in the parent table. In the same way, the updates cascade if the updated column(s) in the child table reference foreign keys in another table.

For example, in the PropertyForRent table the staff number staffNo is a foreign key referencing the Staff table. We can specify a deletion rule such that if a staff record is deleted from the Staff table, the values of the corresponding staffNo column in the PropertyForRent table are set to NULL:

FOREIGN KEY (staffNo) **REFERENCES** Staff **ON DELETE SET NULL**

Similarly, the owner number ownerNo in the PropertyForRent table is a foreign key referencing the PrivateOwner table. We can specify an update rule such that if an owner number is updated in the PrivateOwner table, the corresponding column(s) in the PropertyForRent table are set to the new value:

FOREIGN KEY (ownerNo) **REFERENCES** PrivateOwner **ON UPDATE CASCADE**

7.2.5 General Constraints

Updates to tables may be constrained by enterprise rules governing the real-world transactions that are represented by the updates (see Section 4.3.4). For example, *DreamHome* may have a rule that prevents a member of staff from managing more than 100 properties at the same time. The ISO standard allows general constraints to be specified using the CHECK and UNIQUE clauses of the CREATE and ALTER TABLE statements and the **CREATE ASSERTION** statement. We discussed the CHECK and UNIQUE clauses earlier in this section. The CREATE ASSERTION statement is an integrity constraint that is not directly linked with a table definition. The format of the statement is:

CREATE ASSERTION AssertionName
CHECK (searchCondition)

This statement is very similar to the CHECK clause discussed earlier. However, when a general constraint involves more than one table, it may be preferable to use an ASSERTION rather than duplicate the check in each table or place the constraint in an arbitrary table. For example, to define the general constraint that prevents a member of staff from managing more than 100 properties at the same time, we could write:

```
CREATE ASSERTION StaffNotHandlingTooMuch
    CHECK (NOT EXISTS (SELECT staffNo
                       FROM PropertyForRent
                       GROUP BY staffNo
                       HAVING COUNT(*) > 100))
```

We show how to use these integrity features in the following section when we examine the CREATE and ALTER TABLE statements.

7.3 Data Definition

The SQL DDL allows database objects such as schemas, domains, tables, views, and indexes to be created and destroyed. In this section, we briefly examine how to create and destroy schemas, tables, and indexes. We discuss how to create and destroy views in the next section. The ISO standard also allows the creation of character sets, collations, and translations. However, we will not consider these database objects in this book. The interested reader is referred to Cannan and Otten, 1993.

The main SQL data definition language statements are:

CREATE SCHEMA		DROP SCHEMA
CREATE DOMAIN	ALTER DOMAIN	DROP DOMAIN
CREATE TABLE	ALTER TABLE	DROP TABLE
CREATE VIEW		DROP VIEW

These statements are used to create, change, and destroy the structures that make up the conceptual schema. Although not covered by the SQL standard, the following two statements are provided by many DBMSs:

CREATE INDEX	DROP INDEX

Additional commands are available to the DBA to specify the physical details of data storage; however, we do not discuss these commands here, as they are system-specific.

7.3.1 Creating a Database

The process of creating a database differs significantly from product to product. In multi-user systems, the authority to create a database is usually reserved for the DBA. In a single-user system, a default database may be established when the system is installed and configured and others can be created by the user as and when required. The ISO standard does not specify how databases are created, and each dialect generally has a different approach.

According to the ISO standard, relations and other database objects exist in an **environment**. Among other things, each environment consists of one or more **catalogs**, and each catalog consists of a set of **schemas**. A schema is a named collection of database objects that are in some way related to one another (all the objects in the database are described in one schema or another). The objects in a schema can be tables, views, domains, assertions, collations, translations, and character sets. All the objects in a schema have the same owner and share a number of defaults.

The standard leaves the mechanism for creating and destroying catalogs as implementation-defined, but provides mechanisms for creating and destroying schemas. The schema definition statement has the following (simplified) form:

CREATE SCHEMA [Name | **AUTHORIZATION** CreatorIdentifier]

Therefore, if the creator of a schema SqlTests is Smith, the SQL statement is:

CREATE SCHEMA SqlTests **AUTHORIZATION** Smith;

The ISO standard also indicates that it should be possible to specify within this statement the range of facilities available to the users of the schema, but the details of how these privileges are specified are implementation-dependent.

A schema can be destroyed using the DROP SCHEMA statement, which has the following form:

DROP SCHEMA Name [**RESTRICT** | **CASCADE**]

If RESTRICT is specified, which is the default if neither qualifier is specified, the schema must be empty or the operation fails. If CASCADE is specified, the operation cascades to drop all objects associated with the schema in the order defined previously. If any of these drop operations fail, the DROP SCHEMA fails. The total

effect of a DROP SCHEMA with CASCADE can be very extensive and should be carried out only with extreme caution. It should be noted, however, that the CREATE and DROP SCHEMA statements are not always supported.

7.3.2 Creating a Table (CREATE TABLE)

Having created the database structure, we may now create the table structures for the base relations to be stored in the database. This task is achieved using the CREATE TABLE statement, which has the following basic syntax:

CREATE TABLE TableName
{(columnName dataType [**NOT NULL**] [**UNIQUE**]
[**DEFAULT** defaultOption] [**CHECK** (searchCondition)] [, . . .]}
[**PRIMARY KEY** (listOfColumns),]
{[**UNIQUE** (listOfColumns)] [, . . .]}
{[**FOREIGN KEY** (listOfForeignKeyColumns)
REFERENCES ParentTableName [(listOfCandidateKeyColumns)]
[**MATCH** {**PARTIAL** | **FULL**}
[**ON UPDATE** referentialAction]
[**ON DELETE** referentialAction]] [, . . .]}
{[**CHECK** (searchCondition)] [, . . .]})

As we discussed in the previous section, this version of the CREATE TABLE statement incorporates facilities for defining referential integrity and other constraints. There is significant variation in the support provided by different dialects for this version of the statement. However, when it is supported, the facilities should be used.

The CREATE TABLE statement creates a table called TableName consisting of one or more columns of the specified *dataType*. The set of permissible data types is described in Section 7.1.2. The optional **DEFAULT** clause can be specified to provide a default value for a particular column. SQL uses this default value whenever an INSERT statement fails to specify a value for the column. Among other values, the *defaultOption* includes literals. The NOT NULL, UNIQUE, and CHECK clauses were discussed in the previous section. The remaining clauses are known as **table constraints** and can optionally be preceded with the clause:

CONSTRAINT ConstraintName

which allows the constraint to be dropped by name using the ALTER TABLE statement (see following).

The **PRIMARY KEY** clause specifies the column or columns that form the primary key for the table. If this clause is available, it should be specified for every table created. By default, NOT NULL is assumed for each column that comprises the primary key. Only one PRIMARY KEY clause is allowed per table. SQL rejects any INSERT or UPDATE operation that attempts to create a duplicate row within the PRIMARY KEY column(s). In this way, SQL guarantees the uniqueness of the primary key.

The **FOREIGN KEY** clause specifies a foreign key in the (child) table and the relationship it has to another (parent) table. This clause implements referential integrity constraints. The clause specifies the following:

- A *listOfForeignKeyColumns*, the column or columns from the table being created that form the foreign key.

- A REFERENCES subclause, giving the parent table, that is, the table holding the matching candidate key. If the *listOfCandidateKeyColumns* is omitted, the foreign key is assumed to match the primary key of the parent table. In this case, the parent table must have a PRIMARY KEY clause in its CREATE TABLE statement.
- An optional update rule (ON UPDATE) for the relationship that specifies the action to be taken when a candidate key is updated in the parent table that matches a foreign key in the child table. The **referentialAction** can be CASCADE, SET NULL, SET DEFAULT, or NO ACTION. If the ON UPDATE clause is omitted, the default NO ACTION is assumed (see Section 7.2).
- An optional delete rule (ON DELETE) for the relationship that specifies the action to be taken when a row is deleted from the parent table that has a candidate key that matches a foreign key in the child table. The **referentialAction** is the same as for the ON UPDATE rule.
- By default, the referential constraint is satisfied if any component of the foreign key is null or there is a matching row in the parent table. The MATCH option provides additional constraints relating to nulls within the foreign key. If MATCH FULL is specified, the foreign key components must all be null or must all have values. If MATCH PARTIAL is specified, the foreign key components must all be null, or there must be at least one row in the parent table that could satisfy the constraint if the other nulls were correctly substituted. Some authors argue that referential integrity should imply MATCH FULL.

There can be as many FOREIGN KEY clauses as required. The **CHECK** and **CONSTRAINT** clauses allow additional constraints to be defined. If used as a column constraint, the CHECK clause can reference only the column being defined. Constraints are in effect checked after every SQL statement has been executed, although this check can be deferred until the end of the enclosing transaction (see Section 7.5). Example 7.1 demonstrates the potential of this version of the CREATE TABLE statement.

EXAMPLE 7.1 CREATE TABLE

Create the PropertyForRent *table using the available features of the CREATE TABLE statement.*

```
CREATE DOMAIN OwnerNumber AS VARCHAR(5)
        CHECK (VALUE IN (SELECT ownerNo FROM PrivateOwner));
CREATE DOMAIN StaffNumber AS VARCHAR(5)
        CHECK (VALUE IN (SELECT staffNo FROM Staff));
CREATE DOMAIN BranchNumber AS CHAR(4)
        CHECK (VALUE IN (SELECT branchNo FROM Branch));
CREATE DOMAIN PropertyNumber AS VARCHAR(5);
CREATE DOMAIN Street AS VARCHAR(25);
CREATE DOMAIN City AS VARCHAR(15);
CREATE DOMAIN Postcode AS VARCHAR(8);
CREATE DOMAIN PropertyType AS CHAR(1)
        CHECK(VALUE IN ('B', 'C', 'D', 'E', 'F', 'M', 'S'));
CREATE DOMAIN PropertyRooms AS SMALLINT;
        CHECK(VALUE BETWEEN 1 AND 15);
CREATE DOMAIN PropertyRent AS DECIMAL(6,2)
        CHECK(VALUE BETWEEN 0 AND 9999.99);
```

```
CREATE TABLE PropertyForRent(
    propertyNo      PropertyNumber      NOT NULL,
    street          Street              NOT NULL,
    city            City                NOT NULL,
    postcode        PostCode,
    type            PropertyType        NOT NULL DEFAULT 'F',
    rooms           PropertyRooms       NOT NULL DEFAULT 4,
    rent            PropertyRent        NOT NULL DEFAULT 600,
    ownerNo         OwnerNumber         NOT NULL,
    staffNo         StaffNumber
                    CONSTRAINT StaffNotHandlingTooMuch
                      CHECK (NOT EXISTS (SELECT staffNo
                                         FROM PropertyForRent
                                         GROUP BY staffNo
                                         HAVING COUNT(*) > 100)),
    branchNo        BranchNumber        NOT NULL,
    PRIMARY KEY (propertyNo),
    FOREIGN KEY (staffNo) REFERENCES Staff ON DELETE SET NULL
                                ON UPDATE CASCADE,
    FOREIGN KEY (ownerNo) REFERENCES PrivateOwner ON DELETE NO
                                ACTION ON UPDATE CASCADE,
    FOREIGN KEY (branchNo) REFERENCES Branch ON DELETE NO
                                ACTION ON UPDATE CASCADE);
```

A default value of 'F' for 'Flat' has been assigned to the property type column type. A CONSTRAINT for the staff number column has been specified to ensure that a member of staff does not handle too many properties. The constraint checks whether the number of properties the staff member currently handles is more than than 100.

The primary key is the property number, propertyNo. SQL automatically enforces uniqueness on this column. The staff number, staffNo, is a foreign key referencing the Staff table. A deletion rule has been specified such that, if a record is deleted from the Staff table, the corresponding values of the staffNo column in the PropertyForRent table are set to NULL. Additionally, an update rule has been specified such that if a staff number is updated in the Staff table, the corresponding values in the staffNo column in the PropertyForRent table are updated accordingly. The owner number, ownerNo, is a foreign key referencing the PrivateOwner table. A deletion rule of NO ACTION has been specified to prevent deletions from the PrivateOwner table if there are matching ownerNo values in the PropertyForRent table. An update rule of CASCADE has been specified such that, if an owner number is updated, the corresponding values in the ownerNo column in the PropertyForRent table are set to the new value. The same rules have been specified for the branchNo column. In all FOREIGN KEY constraints, because the *listOfCandidateKeyColumns* has been omitted, SQL assumes that the foreign keys match the primary keys of the respective parent tables.

Note, we have not specified NOT NULL for the staff number column staffNo, because there may be periods of time when there is no member of staff allocated to manage the property (for example, when the property is first registered). However, the other foreign key columns—ownerNo (the owner number) and branchNo (the branch number)—must be specified at all times.

7.3.3 Changing a Table Definition (ALTER TABLE)

The ISO standard provides an ALTER TABLE statement for changing the structure of a table once it has been created. The definition of the ALTER TABLE statement in the ISO standard consists of six options to:

- add a new column to a table;
- drop a column from a table;
- add a new table constraint;
- drop a table constraint;
- set a default for a column;
- drop a default for a column.

The basic format of the statement is:

ALTER TABLE TableName
[**ADD** [**COLUMN**] columnName dataType [**NOT NULL**] [**UNIQUE**]
[**DEFAULT** defaultOption] [**CHECK** (searchCondition)]]
[**DROP** [**COLUMN**] columnName [**RESTRICT** | **CASCADE**]]
[**ADD** [**CONSTRAINT** [ConstraintName]] tableConstraintDefinition]
[**DROP CONSTRAINT** ConstraintName [**RESTRICT** | **CASCADE**]]
[**ALTER** [**COLUMN**] **SET DEFAULT** defaultOption]
[**ALTER** [**COLUMN**] **DROP DEFAULT**]

where the parameters are as defined for the CREATE TABLE statement in the previous section. A *tableConstraintDefinition* is one of the clauses: PRIMARY KEY, UNIQUE, FOREIGN KEY, or CHECK. The ADD COLUMN clause is similar to the definition of a column in the CREATE TABLE statement. The DROP COLUMN clause specifies the name of the column to be dropped from the table definition, and has an optional qualifier that specifies whether the DROP action is to cascade or not:

- RESTRICT: The DROP operation is rejected if the column is referenced by another database object (for example, by a view definition). This is the default setting.
- CASCADE: The DROP operation proceeds and automatically drops the column from any database objects it is referenced by. This operation cascades, so that if a column is dropped from a referencing object, SQL checks whether *that* column is referenced by any other object and drops it from there if it is, and so on.

EXAMPLE 7.2 ALTER TABLE

(a) Change the Staff *table by removing the default of 'Assistant' for the position column and setting the default for the sex column to female ('F').*

ALTER TABLE Staff
    ALTER position **DROP DEFAULT**;
ALTER TABLE Staff
    ALTER sex **SET DEFAULT** 'F';

(b) Change the PropertyForRent *table by removing the constraint that staff are not allowed to handle more than 100 properties at a time. Change the* Client *table by adding a new column representing the preferred number of rooms.*

```
ALTER TABLE PropertyForRent
    DROP CONSTRAINT StaffNotHandlingTooMuch;
ALTER TABLE Client
    ADD prefNoRooms PropertyRooms;
```

The ALTER TABLE statement is not available in all dialects of SQL. In some dialects, the ALTER TABLE statement cannot be used to remove an existing column from a table. In such cases, if a column is no longer required, the column could simply be ignored but kept in the table definition. If, however, you wish to remove the column from the table, you must:

- upload all the data from the table;
- remove the table definition using the DROP TABLE statement;
- redefine the new table using the CREATE TABLE statement;
- reload the data back into the new table.

The upload and reload steps are typically performed with special-purpose utility programs supplied with the DBMS. However, it is possible to create a temporary table and use the INSERT . . . SELECT statement to load the data from the old table into the temporary table and then from the temporary table into the new table.

7.3.4 Removing a Table (DROP TABLE)

Over time, the structure of a database will change; new tables will be created and some tables will no longer be needed. We can remove a redundant table from the database using the DROP TABLE statement, which has the format:

DROP TABLE TableName [**RESTRICT** | **CASCADE**]

For example, to remove the PropertyForRent table we use the command:

DROP TABLE PropertyForRent;

Note, however, that this command removes not only the named table, but also all the rows within it. To simply remove the rows from the table but retain the table structure, use the DELETE statement instead (see Section 6.3.10). The DROP TABLE statement allows you to specify whether the DROP action is to be cascaded:

- RESTRICT: The DROP operation is rejected if there are any other objects that depend for their existence upon the continued existence of the table to be dropped.
- CASCADE: The DROP operation proceeds and SQL automatically drops all dependent objects (and objects dependent on these objects).

The total effect of a DROP TABLE with CASCADE can be very extensive and should be carried out only with extreme caution. One common use of DROP TABLE is to correct mistakes made when creating a table. If a table is created with an incorrect structure, DROP TABLE can be used to delete the newly created table and start again.

7.3.5 Creating an Index (CREATE INDEX)

An index is a structure that provides accelerated access to the rows of a table based on the values of one or more columns (see Appendix F for a discussion of indexes and how they may be used to improve the efficiency of data retrievals). The presence of an index can significantly improve the performance of a query. However, as indexes may be updated by the system every time the underlying tables are updated, additional overheads may be incurred. Indexes are usually created to satisfy particular search criteria after the table has been in use for some time and has grown in size. The creation of indexes is *not* standard SQL. However, most dialects support at least the following capabilities:

CREATE [UNIQUE] INDEX IndexName
ON TableName (columnName [**ASC** | **DESC**] [, . . .])

The specified columns constitute the index key and should be listed in major to minor order. Indexes can be created only on base tables *not* on views. If the UNIQUE clause is used, uniqueness of the indexed column or combination of columns will be enforced by the DBMS. This is certainly required for the primary key and possibly for other columns as well (for example, for alternate keys). Although indexes can be created at any time, we may have a problem if we try to create a unique index on a table with records in it, because the values stored for the indexed column(s) may already contain duplicates. Therefore, it is good practice to create unique indexes, at least for primary key columns, when the base table is created and the DBMS does not automatically enforce primary key uniqueness.

For the Staff and PropertyForRent tables, we may want to create at least the following indexes:

CREATE UNIQUE INDEX StaffNoInd **ON** Staff (staffNo);
CREATE UNIQUE INDEX PropertyNoInd **ON** PropertyForRent (propertyNo);

For each column, we may specify that the order is ascending (ASC) or descending (DESC), with ASC being the default setting. For example, if we create an index on the PropertyForRent table as:

CREATE INDEX RentInd **ON** PropertyForRent (city, rent);

then an index called RentInd is created for the PropertyForRent table. Entries will be in alphabetical order by city and then by rent within each city.

7.3.6 Removing an Index (DROP INDEX)

If we create an index for a base table and later decide that it is no longer needed, we can use the DROP INDEX statement to remove the index from the database. DROP INDEX has the following format:

DROP INDEX IndexName

The following statement will remove the index created in the previous example:

DROP INDEX RentInd;

7.4 Views

Recall from Section 4.4 the definition of a view:

View The dynamic result of one or more relational operations operating on the base relations to produce another relation. A view is a *virtual relation* that does not necessarily exist in the database but can be produced upon request by a particular user, at the time of request.

To the database user, a view appears just like a real table, with a set of named columns and rows of data. However, unlike a base table, a view does not necessarily exist in the database as a stored set of data values. Instead, a view is defined as a query on one or more base tables or views. The DBMS stores the definition of the view in the database. When the DBMS encounters a reference to a view, one approach is to look up this definition and translate the request into an equivalent request against the source tables of the view and then perform the equivalent request. This merging process, called **view resolution**, is discussed in Section 7.4.3. An alternative approach, called **view materialization**, stores the view as a temporary table in the database and maintains the currency of the view as the underlying base tables are updated. We discuss view materialization in Section 7.4.8. First, we examine how to create and use views.

7.4.1 Creating a View (CREATE VIEW)

The format of the CREATE VIEW statement is:

CREATE VIEW ViewName [(newColumnName [, . . .])]
AS subselect [**WITH** [**CASCADED** | **LOCAL**] **CHECK OPTION**]

A view is defined by specifying an SQL SELECT statement. A name may optionally be assigned to each column in the view. If a list of column names is specified, it must have the same number of items as the number of columns produced by the *subselect*. If the list of column names is omitted, each column in the view takes the name of the corresponding column in the *subselect* statement. The list of column names must be specified if there is any ambiguity in the name for a column. This may occur if the *subselect* includes calculated columns, and the AS subclause has not been used to name such columns, or it produces two columns with identical names as the result of a join.

The *subselect* is known as the **defining query**. If WITH CHECK OPTION is specified, SQL ensures that if a row fails to satisfy the WHERE clause of the defining query of the view, it is not added to the underlying base table of the view (see Section 7.4.6). It should be noted that to create a view successfully, you must have SELECT privilege on all the tables referenced in the subselect and USAGE privilege on any domains used in referenced columns. These privileges are discussed further in Section 7.6. Although all views are created in the same way, in practice different types of views are used for different purposes. We illustrate the different types of views with examples.

EXAMPLE 7.3 Create a horizontal view

Create a view so that the manager at branch B003 can see the details only for staff who work in his or her branch office.

A horizontal view restricts a user's access to selected rows of one or more tables.

```
CREATE VIEW Manager3Staff
AS SELECT *
    FROM Staff
    WHERE branchNo = 'B003';
```

This creates a view called Manager3Staff with the same column names as the Staff table but containing only those rows where the branch number is B003. (Strictly speaking, the branchNo column is unnecessary and could have been omitted from the definition of the view, as all entries have branchNo = 'B003'.) If we now execute this statement:

```
SELECT * FROM Manager3Staff;
```

we get the result table shown in Table 7.3. To ensure that the branch manager can see only these rows, the manager should not be given access to the base table Staff. Instead, the manager should be given access permission to the view Manager3Staff. This, in effect, gives the branch manager a customized view of the Staff table, showing only the staff at his or her own branch. We discuss access permissions in Section 7.6.

TABLE 7.3 Data for view Manager3Staff.

staffNo	fName	lName	position	sex	DOB	salary	branchNo
SG37	Ann	Beech	Assistant	F	10-Nov-60	12000.00	B003
SG14	David	Ford	Supervisor	M	24-Mar-58	18000.00	B003
SG5	Susan	Brand	Manager	F	3-Jun-40	24000.00	B003

EXAMPLE 7.4 Create a vertical view

Create a view of the staff details at branch B003 that excludes salary information, so that only managers can access the salary details for staff who work at their branch.

A vertical view restricts a user's access to selected columns of one or more tables.

```
CREATE VIEW Staff3
AS SELECT staffNo, fName, lName, position, sex
    FROM Staff
    WHERE branchNo = 'B003';
```

Note that we could rewrite this statement to use the Manager3Staff view instead of the Staff table, thus:

```
CREATE VIEW Staff3
AS SELECT staffNo, fName, lName, position, sex
    FROM Manager3Staff;
```

Either way, this creates a view called Staff3 with the same columns as the Staff table, but excluding the salary, DOB, and branchNo columns. If we list this view, we get the result

table shown in Table 7.4. To ensure that only the branch manager can see the salary details, staff at branch B003 should not be given access to the base table Staff or the view Manager3Staff. Instead, they should be given access permission to the view Staff3, thereby denying them access to sensitive salary data.

Vertical views are commonly used where the data stored in a table is used by various users or groups of users. They provide a private table for these users composed only of the columns they need.

TABLE 7.4 Data for view Staff3.

staffNo	fName	lName	position	sex
SG37	Ann	Beech	Assistant	F
SG14	David	Ford	Supervisor	M
SG5	Susan	Brand	Manager	F

EXAMPLE 7.5 Grouped and joined views

Create a view of staff who manage properties for rent, which includes the branch number they work at, their staff number, and the number of properties they manage (see Example 6.27).

```
CREATE VIEW StaffPropCnt (branchNo, staffNo, cnt)
AS SELECT s.branchNo, s.staffNo, COUNT(*)
    FROM Staff s, PropertyForRent p
    WHERE s.staffNo = p.staffNo
    GROUP BY s.branchNo, s.staffNo;
```

This example gives the data shown in Table 7.5. It illustrates the use of a subselect containing a GROUP BY clause (giving a view called a **grouped view**), and containing multiple tables (giving a view called a **joined view**). One of the most frequent reasons for using views is to simplify multi-table queries. Once a joined view has been defined, we can often use a simple single-table query against the view for queries that would otherwise require a multi-table join. Note that we have to name the columns in the definition of the view because of the use of the unqualified aggregate function COUNT in the subselect.

TABLE 7.5 Data for view StaffPropCnt.

branchNo	staffNo	cnt
B003	SG14	1
B003	SG37	2
B005	SL41	1
B007	SA9	1

7.4.2 Removing a View (DROP VIEW)

A view is removed from the database with the DROP VIEW statement:

```
DROP VIEW ViewName [RESTRICT | CASCADE]
```

DROP VIEW causes the definition of the view to be deleted from the database. For example, we could remove the Manager3Staff view using the following statement:

```
DROP VIEW Manager3Staff;
```

If CASCADE is specified, DROP VIEW deletes all related dependent objects; in other words, all objects that reference the view. This means that DROP VIEW also deletes any views that are defined on the view being dropped. If RESTRICT is specified and there are any other objects that depend for their existence on the continued existence of the view being dropped, the command is rejected. The default setting is RESTRICT.

7.4.3 View Resolution

Having considered how to create and use views, we now look more closely at how a query on a view is handled. To illustrate the process of **view resolution**, consider the following query, which counts the number of properties managed by each member of staff at branch office B003. This query is based on the StaffPropCnt view of Example 7.5:

```
SELECT staffNo, cnt
FROM StaffPropCnt
WHERE branchNo = 'B003'
ORDER BY staffNo;
```

View resolution merges the example query with the defining query of the StaffPropCnt view as follows:

(1) The view column names in the SELECT list are translated into their corresponding column names in the defining query. This gives:

```
SELECT s.staffNo AS staffNo, COUNT(*) AS cnt
```

(2) View names in the FROM clause are replaced with the corresponding FROM lists of the defining query:

```
FROM Staff s, PropertyForRent p
```

(3) The WHERE clause from the user query is combined with the WHERE clause of the defining query using the logical operator AND, thus:

```
WHERE s.staffNo = p.staffNo AND branchNo = 'B003'
```

(4) The GROUP BY and HAVING clauses are copied from the defining query. In this example, we have only a GROUP BY clause:

```
GROUP BY s.branchNo, s.staffNo
```

(5) Finally, the ORDER BY clause is copied from the user query with the view column name translated into the defining query column name:

```
ORDER BY s.staffNo
```

(6) The final merged query becomes:

```
SELECT s.staffNo AS staffNo, COUNT(*) AS cnt
FROM Staff s, PropertyForRent p
```

© CourseSmart

```
WHERE s.staffNo = p.staffNo AND branchNo = 'B003'
GROUP BY s.branchNo, s.staffNo
ORDER BY s.staffNo;
```

This gives the result table shown in Table 7.6.

TABLE 7.6 Result table after view resolution.

staffNo	cnt
SG14	1
SG37	2

7.4.4 Restrictions on Views

The ISO standard imposes several important restrictions on the creation and use of views, although there is considerable variation among dialects.

- If a column in the view is based on an aggregate function, then the column may appear only in SELECT and ORDER BY clauses of queries that access the view. In particular, such a column may not be used in a WHERE clause and may not be an argument to an aggregate function in any query based on the view. For example, consider the view StaffPropCnt of Example 7.5, which has a column cnt based on the aggregate function COUNT. The following query would fail:

  ```
  SELECT COUNT(cnt)
  FROM StaffPropCnt;
  ```

  because we are using an aggregate function on the column cnt, which is itself based on an aggregate function. Similarly, the following query would also fail:

  ```
  SELECT *
  FROM StaffPropCnt
  WHERE cnt > 2;
  ```

  because we are using the view column, cnt, derived from an aggregate function, on the left-hand side of a WHERE clause.
- A grouped view may never be joined with a base table or a view. For example, the StaffPropCnt view is a grouped view, so any attempt to join this view with another table or view fails.

7.4.5 View Updatability

All updates to a base table are immediately reflected in all views that encompass that base table. Similarly, we may expect that if a view is updated, the base table(s) will reflect that change. However, consider again the view StaffPropCnt of Example 7.5. Consider what would happen if we tried to insert a record that showed that at branch B003, staff member SG5 manages two properties, using the following insert statement:

```
INSERT INTO StaffPropCnt
VALUES ('B003', 'SG5', 2);
```

We have to insert two records into the PropertyForRent table showing which properties staff member SG5 manages. However, we do not know which properties they

are; all we know is that this member of staff manages two properties. In other words, we do not know the corresponding primary key values for the PropertyForRent table. If we change the definition of the view and replace the count with the actual property numbers as follows:

```
CREATE VIEW StaffPropList (branchNo, staffNo, propertyNo)
AS SELECT s.branchNo, s.staffNo, p.propertyNo
   FROM Staff s, PropertyForRent p
   WHERE s.staffNo = p.staffNo;
```

and we try to insert the record:

```
INSERT INTO StaffPropList
VALUES ('B003', 'SG5', 'PG19');
```

there is still a problem with this insertion, because we specified in the definition of the PropertyForRent table that all columns except postcode and staffNo were not allowed to have nulls (see Example 7.1). However, as the StaffPropList view excludes all columns from the PropertyForRent table except the property number, we have no way of providing the remaining nonnull columns with values.

The ISO standard specifies the views that must be updatable in a system that conforms to the standard. The definition given in the ISO standard is that a view is updatable if and only if:

- DISTINCT is not specified; that is, duplicate rows must not be eliminated from the query results.
- Every element in the SELECT list of the defining query is a column name (rather than a constant, expression, or aggregate function) and no column name appears more than once.
- The FROM clause specifies only one table; that is, the view must have a single source table for which the user has the required privileges. If the source table is itself a view, then that view must satisfy these conditions. This, therefore, excludes any views based on a join, union (UNION), intersection (INTERSECT), or difference (EXCEPT).
- The WHERE clause does not include any nested SELECTs that reference the table in the FROM clause.
- There is no GROUP BY or HAVING clause in the defining query.

In addition, every row that is added through the view must not violate the integrity constraints of the base table. For example, if a new row is added through a view, columns that are not included in the view are set to null, but this must not violate a NOT NULL integrity constraint in the base table. The basic concept behind these restrictions is as follows:

Updatable view — For a view to be updatable, the DBMS must be able to trace any row or column back to its row or column in the source table.

7.4.6 WITH CHECK OPTION

Rows exist in a view, because they satisfy the WHERE condition of the defining query. If a row is altered such that it no longer satisfies this condition, then it will

disappear from the view. Similarly, new rows will appear within the view when an insert or update on the view causes them to satisfy the WHERE condition. The rows that enter or leave a view are called **migrating rows**.

Generally, the WITH CHECK OPTION clause of the CREATE VIEW statement prohibits a row from migrating out of the view. The optional qualifiers LOCAL/CASCADED are applicable to view hierarchies, that is, a view that is derived from another view. In this case, if WITH LOCAL CHECK OPTION is specified, then any row insert or update on this view, and on any view directly or indirectly defined on this view, must not cause the row to disappear from the view, unless the row also disappears from the underlying derived view/table. If the WITH CASCADED CHECK OPTION is specified (the default setting), then any row insert or update on this view and on any view directly or indirectly defined on this view must not cause the row to disappear from the view.

This feature is so useful that it can make working with views more attractive than working with the base tables. When an INSERT or UPDATE statement on the view violates the WHERE condition of the defining query, the operation is rejected. This behavior enforces constraints on the database and helps preserve database integrity. The WITH CHECK OPTION can be specified only for an updatable view, as defined in the previous section.

EXAMPLE 7.6 WITH CHECK OPTION

Consider again the view created in Example 7.3:

```
CREATE VIEW Manager3Staff
AS SELECT *
   FROM Staff
   WHERE branchNo = 'B003'
WITH CHECK OPTION;
```

with the virtual table shown in Table 7.3. If we now attempt to update the branch number of one of the rows from B003 to B005, for example:

```
UPDATE Manager3Staff
SET branchNo = 'B005'
WHERE staffNo = 'SG37';
```

then the specification of the WITH CHECK OPTION clause in the definition of the view prevents this from happening, as it would cause the row to migrate from this horizontal view. Similarly, if we attempt to insert the following row through the view:

```
INSERT INTO Manager3Staff
VALUES('SL15', 'Mary', 'Black', 'Assistant', 'F', DATE'1967-06-21', 8000, 'B002');
```

then the specification of WITH CHECK OPTION would prevent the row from being inserted into the underlying Staff table and immediately disappearing from this view (as branch B002 is not part of the view).

Now consider the situation where Manager3Staff is defined not on Staff directly but on another view of Staff:

```
CREATE VIEW LowSalary
AS SELECT *
   FROM Staff
   WHERE salary > 9000;

CREATE VIEW HighSalary
AS SELECT *
   FROM LowSalary
   WHERE salary > 10000
WITH LOCAL CHECK OPTION;

CREATE VIEW Manager3Staff
AS SELECT *
   FROM HighSalary
   WHERE branchNo = 'B003';
```

If we now attempt the following update on Manager3Staff:

```
UPDATE Manager3Staff
SET salary = 9500
WHERE staffNo = 'SG37';
```

then this update would fail: although the update would cause the row to disappear from the view HighSalary, the row would not disappear from the table LowSalary that HighSalary is derived from. However, if instead the update tried to set the salary to 8000, then the update would succeed, as the row would no longer be part of LowSalary. Alternatively, if the view HighSalary had specified WITH CASCADED CHECK OPTION, then setting the salary to either 9500 or 8000 would be rejected, because the row would disappear from HighSalary. Therefore, to ensure that anomalies like this do not arise, each view should normally be created using the WITH CASCADED CHECK OPTION.

7.4.7 Advantages and Disadvantages of Views

Restricting some users' access to views has potential advantages over allowing users direct access to the base tables. Unfortunately, views in SQL also have disadvantages. In this section we briefly review the advantages and disadvantages of views in SQL as summarized in Table 7.7.

TABLE 7.7 Summary of advantages/disadvantages of views in SQL.

ADVANTAGES	DISADVANTAGES
Data independence	Update restriction
Currency	Structure restriction
Improved security	Performance
Reduced complexity	
Convenience	
Customization	
Data integrity	

Advantages

In the case of a DBMS running on a standalone PC, views are usually a convenience, defined to simplify database requests. However, in a multi-user DBMS, views play a central role in defining the structure of the database and enforcing security and integrity. The major advantages of views are described next.

Data independence A view can present a consistent, unchanging picture of the structure of the database, even if the underlying source tables are changed (for example, if columns added or removed, relationships changed, tables split,

restructured, or renamed). If columns are added or removed from a table, and these columns are not required by the view, then the definition of the view need not change. If an existing table is rearranged or split up, a view may be defined so that users can continue to see the old table. In the case of splitting a table, the old table can be recreated by defining a view from the join of the new tables, provided that the split is done in such a way that the original table can be reconstructed. We can ensure that this is possible by placing the primary key in both of the new tables. Thus, if we originally had a Client table of the following form:

Client (clientNo, fName, lName, telNo, prefType, maxRent, eMail)

we could reorganize it into two new tables:

ClientDetails (clientNo, fName, lName, telNo, eMail)
ClientReqts (clientNo, prefType, maxRent)

Users and applications could still access the data using the old table structure, which would be recreated by defining a view called Client as the natural join of ClientDetails and ClientReqts, with clientNo as the join column:

```
CREATE VIEW Client
AS SELECT cd.clientNo, fName, lName, telNo, prefType, maxRent, eMail
   FROM ClientDetails cd, ClientReqts cr
   WHERE cd.clientNo = cr.clientNo;
```

Currency Changes to any of the base tables in the defining query are immediately reflected in the view.

Improved security Each user can be given the privilege to access the database only through a small set of views that contain the data appropriate for that user, thus restricting and controlling each user's access to the database.

Reduced complexity A view can simplify queries, by drawing data from several tables into a single table, thereby transforming multi-table queries into single-table queries.

Convenience Views can provide greater convenience to users as users are presented with only that part of the database that they need to see. This also reduces the complexity from the user's point of view.

Customization Views provide a method to customize the appearance of the database, so that the same underlying base tables can be seen by different users in different ways.

Data integrity If the WITH CHECK OPTION clause of the CREATE VIEW statement is used, then SQL ensures that no row that fails to satisfy the WHERE clause of the defining query is ever added to any of the underlying base table(s) through the view, thereby ensuring the integrity of the view.

Disadvantages

Although views provide many significant benefits, there are also some disadvantages with SQL views.

Update restriction In Section 7.4.5 we showed that, in some cases, a view cannot be updated.

Structure restriction The structure of a view is determined at the time of its creation. If the defining query was of the form SELECT * FROM . . . , then the * refers to the columns of the base table present when the view is created. If columns are subsequently added to the base table, then these columns will not appear in the view, unless the view is dropped and recreated.

Performance There is a performance penalty to be paid when using a view. In some cases, this will be negligible; in other cases, it may be more problematic. For example, a view defined by a complex, multi-table query may take a long time to process, as the view resolution must join the tables together *every time the view is accessed.* View resolution requires additional computer resources. In the next section we briefly discuss an alternative approach to maintaining views that attempts to overcome this disadvantage.

7.4.8 View Materialization

In Section 7.4.3 we discussed one approach to handling queries based on a view, in which the query is modified into a query on the underlying base tables. One disadvantage with this approach is the time taken to perform the view resolution, particularly if the view is accessed frequently. An alternative approach, called **view materialization**, is to store the view as a temporary table in the database when the view is first queried. Thereafter, queries based on the materialized view can be much faster than recomputing the view each time. The speed difference may be critical in applications where the query rate is high and the views are complex, so it is not practical to recompute the view for every query.

Materialized views are useful in new applications such as data warehousing, replication servers, data visualization, and mobile systems. Integrity constraint checking and query optimization can also benefit from materialized views. The difficulty with this approach is maintaining the currency of the view while the base table(s) are being updated. The process of updating a materialized view in response to changes to the underlying data is called **view maintenance**. The basic aim of view maintenance is to apply only those changes necessary to the view to keep it current. As an indication of the issues involved, consider the following view:

```
CREATE VIEW StaffPropRent (staffNo)
AS SELECT DISTINCT staffNo
   FROM PropertyForRent
   WHERE branchNo = 'B003' AND rent > 400;
```

TABLE 7.8 Data for view StaffPropRent.

staffNo
SG37
SG14

with the data shown in Table 7.8. If we were to insert a row into the PropertyForRent table with a rent ≤ 400, then the view would be unchanged. If we were to insert the row ('PG24', . . . , 550, 'CO40', 'SG19', 'B003') into the PropertyForRent table, then the row should also appear within the materialized view. However, if we were to insert the row ('PG54', . . . , 450, 'CO89', 'SG37', 'B003') into the PropertyForRent table, then no new row need be added to the materialized view, because there is a row for SG37 already. Note that in these three cases the decision whether to insert

the row into the materialized view can be made without access to the underlying PropertyForRent table.

If we now wished to delete the new row ('PG24', . . . , 550, 'CO40', 'SG19', 'B003') from the PropertyForRent table, then the row should also be deleted from the materialized view. However, if we wished to delete the new row ('PG54', . . . , 450, 'CO89', 'SG37', 'B003') from the PropertyForRent table, then the row corresponding to SG37 should not be deleted from the materialized view, owing to the existence of the underlying base row corresponding to property PG21. In these two cases, the decision on whether to delete or retain the row in the materialized view requires access to the underlying base table PropertyForRent. For a more complete discussion of materialized views, the interested reader is referred to Gupta and Mumick, 1999.

7.5 Transactions

The ISO standard defines a transaction model based on two SQL statements: COMMIT and ROLLBACK. Most, but not all, commercial implementations of SQL conform to this model, which is based on IBM's DB2 DBMS. A transaction is a logical unit of work consisting of one or more SQL statements that is guaranteed to be atomic with respect to recovery. The standard specifies that an SQL transaction automatically begins with a **transaction-initiating** SQL statement executed by a user or program (for example, SELECT, INSERT, UPDATE). Changes made by a transaction are not visible to other concurrently executing transactions until the transaction completes. A transaction can complete in one of four ways:

- A COMMIT statement ends the transaction successfully, making the database changes permanent. A new transaction starts after COMMIT with the next transaction-initiating statement.
- A ROLLBACK statement aborts the transaction, backing out any changes made by the transaction. A new transaction starts after ROLLBACK with the next transaction-initiating statement.
- For programmatic SQL (see Appendix I), successful program termination ends the final transaction successfully, even if a COMMIT statement has not been executed.
- For programmatic SQL, abnormal program termination aborts the transaction.

SQL transactions cannot be nested (see Section 22.4). The SET TRANSACTION statement allows the user to configure certain aspects of the transaction. The basic format of the statement is:

```
SET TRANSACTION
[READ ONLY | READ WRITE] |
[ISOLATION LEVEL READ UNCOMMITTED | READ COMMITTED |
REPEATABLE READ | SERIALIZABLE]
```

The READ ONLY and READ WRITE qualifiers indicate whether the transaction is read-only or involves both read and write operations. The default is READ WRITE if neither qualifier is specified (unless the isolation level is READ UNCOMMITTED). Perhaps confusingly, READ ONLY allows a transaction to issue INSERT, UPDATE, and DELETE statements against temporary tables (but only temporary tables).

© CourseSmart

TABLE 7.9 Violations of serializability permitted by isolation levels.

ISOLATION LEVEL	DIRTY READ	NONREPEATABLE READ	PHANTOM READ
READ UNCOMMITTED	Y	Y	Y
READ COMMITTED	N	Y	Y
REPEATABLE READ	N	N	Y
SERIALIZABLE	N	N	N

The *isolation level* indicates the degree of interaction that is allowed from other transactions during the execution of the transaction. Table 7.9 shows the violations of serializability allowed by each isolation level against the following three preventable phenomena:

- *Dirty read:* A transaction reads data that has been written by another as yet uncommitted transaction.
- *Nonrepeatable read:* A transaction rereads data that it has previously read, but another committed transaction has modified or deleted the data in the intervening period.
- *Phantom read:* A transaction executes a query that retrieves a set of rows satisfying a certain search condition. When the transaction re-executes the query at a later time, additional rows are returned that have been inserted by another committed transaction in the intervening period.

Only the SERIALIZABLE isolation level is safe, that is, generates serializable schedules. The remaining isolation levels require a mechanism to be provided by the DBMS that can be used by the programmer to ensure serializability. Chapter 22 provides additional information on transactions and serializability.

7.5.1 Immediate and Deferred Integrity Constraints

In some situations, we do not want integrity constraints to be checked immediately—that is, after every SQL statement has been executed—but instead at transaction commit. A constraint may be defined as INITIALLY IMMEDIATE or INITIALLY DEFERRED, indicating which mode the constraint assumes at the start of each transaction. In the former case, it is also possible to specify whether the mode can be changed subsequently using the qualifier [NOT] DEFERRABLE. The default mode is INITIALLY IMMEDIATE.

The SET CONSTRAINTS statement is used to set the mode for specified constraints for the current transaction. The format of this statement is:

SET CONSTRAINTS
{**ALL** | constraintName [, . . .]} {**DEFERRED** | **IMMEDIATE**}

7.6 Discretionary Access Control

In Section 2.4 we stated that a DBMS should provide a mechanism to ensure that only authorized users can access the database. Modern DBMSs typically provide one or both of the following authorization mechanisms:

- *Discretionary access control:* Each user is given appropriate access rights (or *privileges*) on specific database objects. Typically users obtain certain privileges when they create an object and can pass some or all of these privileges to other users at their discretion. Although flexible, this type of authorization mechanism can be circumvented by a devious unauthorized user tricking an authorized user into revealing sensitive data.
- *Mandatory access control:* Each database object is assigned a certain *classification level* (for example, Top Secret, Secret, Confidential, Unclassified) and each *subject* (for example, users or programs) is given a designated *clearance level*. The classification levels form a strict ordering (Top Secret > Secret > Confidential > Unclassified) and a subject requires the necessary clearance to read or write a database object. This type of multilevel security mechanism is important for certain government, military, and corporate applications. The most commonly used mandatory access control model is known as Bell–LaPadula (Bell and LaPadula, 1974), which we discuss further in Chapter 20.

SQL supports only discretionary access control through the GRANT and REVOKE statements. The mechanism is based on the concepts **of authorization identifiers**, **ownership**, and **privileges**, as we now discuss.

Authorization identifiers and ownership

An authorization identifier is a normal SQL identifier that is used to establish the identity of a user. Each database user is assigned an authorization identifier by the DBA. Usually, the identifier has an associated password, for obvious security reasons. Every SQL statement that is executed by the DBMS is performed on behalf of a specific user. The authorization identifier is used to determine which database objects the user may reference and what operations may be performed on those objects.

Each object that is created in SQL has an owner. The owner is identified by the authorization identifier defined in the AUTHORIZATION clause of the schema to which the object belongs (see Section 7.3.1). The owner is initially the only person who may know of the existence of the object and, consequently, perform any operations on the object.

Privileges

Privileges are the actions that a user is permitted to carry out on a given base table or view. The privileges defined by the ISO standard are:

- SELECT—the privilege to retrieve data from a table;
- INSERT—the privilege to insert new rows into a table;
- UPDATE—the privilege to modify rows of data in a table;
- DELETE—the privilege to delete rows of data from a table;
- REFERENCES—the privilege to reference columns of a named table in integrity constraints;
- USAGE—the privilege to use domains, collations, character sets, and translations. We do not discuss collations, character sets, and translations in this book; the interested reader is referred to Cannan and Otten, 1993.

The INSERT and UPDATE privileges can be restricted to specific columns of the table, allowing changes to these columns but disallowing changes to any other column.

Similarly, the REFERENCES privilege can be restricted to specific columns of the table, allowing these columns to be referenced in constraints, such as check constraints and foreign key constraints, when creating another table, but disallowing others from being referenced.

When a user creates a table using the CREATE TABLE statement, he or she automatically becomes the owner of the table and receives full privileges for the table. Other users initially have no privileges on the newly created table. To give them access to the table, the owner must explicitly grant them the necessary privileges using the GRANT statement.

When a user creates a view with the CREATE VIEW statement, he or she automatically becomes the owner of the view, but does not necessarily receive full privileges on the view. To create the view, a user must have SELECT privilege on all the tables that make up the view and REFERENCES privilege on the named columns of the view. However, the view owner gets INSERT, UPDATE, and DELETE privileges only if he or she holds these privileges for every table in the view.

7.6.1 Granting Privileges to Other Users (GRANT)

The GRANT statement is used to grant privileges on database objects to specific users. Normally the GRANT statement is used by the owner of a table to give other users access to the data. The format of the GRANT statement is:

```
GRANT   {PrivilegeList | ALL PRIVILEGES}
ON      ObjectName
TO      {AuthorizationIdList | PUBLIC}
[WITH GRANT OPTION]
```

PrivilegeList consists of one or more of the following privileges, separated by commas:

```
SELECT
DELETE
INSERT      [(columnName [, . . . ])]
UPDATE      [(columnName [, . . . ])]
REFERENCES  [(columnName [, . . . ])]
USAGE
```

For convenience, the GRANT statement allows the keyword ALL PRIVILEGES to be used to grant all privileges to a user instead of having to specify the six privileges individually. It also provides the keyword PUBLIC to allow access to be granted to all present and future authorized users, not just to the users currently known to the DBMS. *ObjectName* can be the name of a base table, view, domain, character set, collation, or translation.

The WITH GRANT OPTION clause allows the user(s) in *AuthorizationIdList* to pass the privileges they have been given for the named object on to other users. If these users pass a privilege on specifying WITH GRANT OPTION, the users receiving the privilege may in turn grant it to still other users. If this keyword is not specified, the receiving user(s) will not be able to pass the privileges on to other users. In this way, the owner of the object maintains very tight control over who has permission to use the object and what forms of access are allowed.

EXAMPLE 7.7 GRANT all privileges

Give the user with authorization identifier Manager *all privileges on the* Staff *table.*

```
GRANT ALL PRIVILEGES
ON Staff
TO Manager WITH GRANT OPTION;
```

The user identified as Manager can now retrieve rows from the Staff table, and also insert, update, and delete data from this table. Manager can also reference the Staff table, and all the Staff columns in any table that he or she creates subsequently. We also specified the keyword WITH GRANT OPTION, so that Manager can pass these privileges on to other users.

EXAMPLE 7.8 GRANT specific privileges

Give users Personnel *and* Director *the privileges SELECT and UPDATE on column* salary *of the* Staff *table.*

```
GRANT SELECT, UPDATE (salary)
ON Staff
TO Personnel, Director;
```

We have omitted the keyword WITH GRANT OPTION, so that users Personnel and Director cannot pass either of these privileges on to other users.

EXAMPLE 7.9 GRANT specific privileges to PUBLIC

Give all users the privilege SELECT on the Branch *table.*

```
GRANT SELECT
ON Branch
TO PUBLIC;
```

The use of the keyword PUBLIC means that all users (now and in the future) are able to retrieve all the data in the Branch table. Note that it does not make sense to use WITH GRANT OPTION in this case: as every user has access to the table, there is no need to pass the privilege on to other users.

7.6.2 Revoking Privileges from Users (REVOKE)

The REVOKE statement is used to take away privileges that were granted with the GRANT statement. A REVOKE statement can take away all or some of the privileges that were previously granted to a user. The format of the statement is:

```
REVOKE [GRANT OPTION FOR] {PrivilegeList | ALL PRIVILEGES}
ON      ObjectName
FROM    {AuthorizationIdList | PUBLIC} [RESTRICT | CASCADE]
```

The keyword ALL PRIVILEGES refers to all the privileges granted to a user by the user revoking the privileges. The optional GRANT OPTION FOR clause allows privileges passed on via the WITH GRANT OPTION of the GRANT statement to be revoked separately from the privileges themselves.

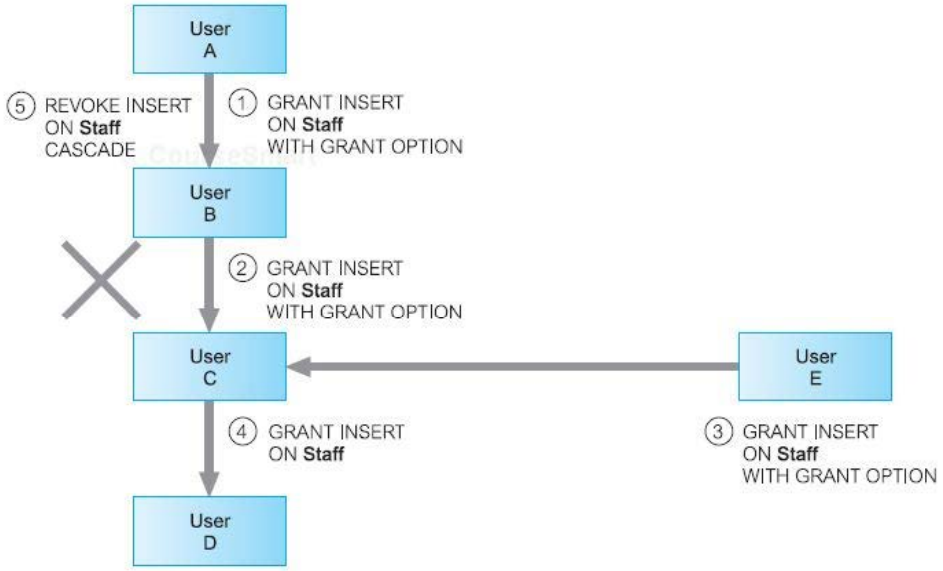

Figure 7.1 Effects of REVOKE.

The RESTRICT and CASCADE qualifiers operate exactly as in the DROP TABLE statement (see Section 7.3.3). Because privileges are required to create certain objects, revoking a privilege can remove the authority that allowed the object to be created (such an object is said to be **abandoned**). The REVOKE statement fails if it results in an abandoned object, such as a view, unless the CASCADE keyword has been specified. If CASCADE is specified, an appropriate DROP statement is issued for any abandoned views, domains, constraints, or assertions.

The privileges that were granted to this user by other users are not affected by this REVOKE statement. Therefore, if another user has granted the user the privilege being revoked, the other user's grant still allows the user to access the table. For example, in Figure 7.1 User A grants User B INSERT privilege on the Staff table WITH GRANT OPTION (step 1). User B passes this privilege on to User C (step 2). Subsequently, User C gets the same privilege from User E (step 3). User C then passes the privilege on to User D (step 4). When User A revokes the INSERT privilege from User B (step 5), the privilege cannot be revoked from User C, because User C has also received the privilege from User E. If User E had not given User C this privilege, the revoke would have cascaded to User C and User D.

EXAMPLE 7.10 REVOKE specific privileges from PUBLIC

Revoke the privilege SELECT on the Branch *table from all users.*

```
REVOKE SELECT
ON Branch
FROM PUBLIC;
```

EXAMPLE 7.11 REVOKE specific privileges from named user

Revoke all privileges you have given to Director *on the* Staff *table.*

```
REVOKE ALL PRIVILEGES
ON Staff
FROM Director;
```

This is equivalent to REVOKE SELECT . . . , as this was the only privilege that has been given to Director.

Chapter Summary

- The ISO standard provides eight base data types: boolean, character, bit, exact numeric, approximate numeric, datetime, interval, and character/binary large objects.
- The SQL DDL statements allow database objects to be defined. The CREATE and DROP SCHEMA statements allow schemas to be created and destroyed; the CREATE, ALTER, and DROP TABLE statements allow tables to be created, modified, and destroyed; the CREATE and DROP INDEX statements allow indexes to be created and destroyed.
- The ISO SQL standard provides clauses in the **CREATE** and **ALTER TABLE** statements to define **integrity constraints** that handle required data, domain constraints, entity integrity, referential integrity, and general constraints. **Required data** can be specified using NOT NULL. **Domain constraints** can be specified using the CHECK clause or by defining domains using the CREATE DOMAIN statement. **Primary keys** should be defined using the PRIMARY KEY clause and **alternate keys** using the combination of NOT NULL and UNIQUE. **Foreign keys** should be defined using the FOREIGN KEY clause and update and delete rules using the subclauses ON UPDATE and ON DELETE. **General constraints** can be defined using the CHECK and UNIQUE clauses. General constraints can also be created using the CREATE ASSERTION statement.
- A **view** is a virtual table representing a subset of columns and/or rows and/or column expressions from one or more base tables or views. A view is created using the CREATE VIEW statement by specifying a **defining query**. It may not necessarily be a physically stored table, but may be recreated each time it is referenced.
- Views can be used to simplify the structure of the database and make queries easier to write. They can also be used to protect certain columns and/or rows from unauthorized access. Not all views are updatable.
- **View resolution** merges the query on a view with the definition of the view producing a query on the underlying base table(s). This process is performed each time the DBMS has to process a query on a view. An alternative approach, called **view materialization**, stores the view as a temporary table in the database when the view is first queried. Thereafter, queries based on the materialized view can be much faster than recomputing the view each time. One disadvantage with materialized views is maintaining the currency of the temporary table.
- The COMMIT statement signals successful completion of a transaction and all changes to the database are made permanent. The ROLLBACK statement signals that the transaction should be aborted and all changes to the database are undone.
- SQL access control is built around the concepts of authorization identifiers, ownership, and privileges. **Authorization identifiers** are assigned to database users by the DBA and identify a user. Each object

that is created in SQL has an **owner**. The owner can pass **privileges** on to other users using the GRANT statement and can revoke the privileges passed on using the REVOKE statement. The privileges that can be passed on are USAGE, SELECT, DELETE, INSERT, UPDATE, and REFERENCES; INSERT, UPDATE, and REFERENCES can be restricted to specific columns. A user can allow a receiving user to pass privileges on using the WITH GRANT OPTION clause and can revoke this privilege using the GRANT OPTION FOR clause.

Review Questions

7.1 Describe the eight base data types in SQL.

7.2 Discuss the functionality and importance of the Integrity Enhancement Feature (IFF).

7.3 Discuss each of the clauses of the CREATE TABLE statement.

7.4 Discuss the advantages and disadvantages of views.

7.5 Describe how the process of view resolution works.

7.6 What restrictions are necessary to ensure that a view is updatable?

7.7 What is a materialized view and what are the advantages of a maintaining a materialized view rather than using the view resolution process?

7.8 Describe the difference between discretionary and mandatory access control. What type of control mechanism does SQL support?

7.9 Describe how the access control mechanisms of SQL work.

Exercises

Answer the following questions using the relational schema from the Exercises at the end of Chapter 4:

7.10 Create the **Hotel** table using the integrity enhancement features of SQL.

7.11 Now create the **Room**, **Booking**, and **Guest** tables using the integrity enhancement features of SQL with the following constraints:
(a) **type** must be one of Single, Double, or Family.
(b) **price** must be between £10 and £100.
(c) **roomNo** must be between 1 and 100.
(d) **dateFrom** and **dateTo** must be greater than today's date.
(e) The same room cannot be double-booked.
(f) The same guest cannot have overlapping bookings.

7.12 Create a separate table with the same structure as the **Booking** table to hold archive records. Using the INSERT statement, copy the records from the **Booking** table to the archive table relating to bookings before 1 January 2007. Delete all bookings before 1 January 2007 from the **Booking** table.

7.13 Create a view containing the hotel name and the names of the guests staying at the hotel.

7.14 Create a view containing the account for each guest at the Grosvenor Hotel.

7.15 Give the users **Manager** and **Director** full access to these views, with the privilege to pass the access on to other users.

7.16 Give the user **Accounts** SELECT access to these views. Now revoke the access from this user.

© CourseSmart

7.17 Consider the following view defined on the Hotel schema:

```
CREATE VIEW HotelBookingCount (hotelNo, bookingCount)
AS SELECT h.hotelNo, COUNT(*)
    FROM Hotel h, Room r, Booking b
    WHERE h.hotelNo = r.hotelNo AND r.roomNo = b.roomNo
    GROUP BY h.hotelNo;
```

For each of the following queries, state whether the query is valid, and for the valid ones, show how each of the queries would be mapped on to a query on the underlying base tables.

(a)
```
SELECT *
FROM HotelBookingCount;
```
(b)
```
SELECT hotelNo
FROM HotelBookingCount
WHERE hotelNo ='H001';
```
(c)
```
SELECT MIN(bookingCount)
FROM HotelBookingCount;
```
(d)
```
SELECT COUNT(*)
FROM HotelBookingCount;
```
(e)
```
SELECT hotelNo
FROM HotelBookingCount
WHERE bookingCount > 1000;
```
(f)
```
SELECT hotelNo
FROM HotelBookingCount
ORDER BY bookingCount;
```

7.19 Assume that we also have a table for suppliers:

Supplier (supplierNo. partNo, price)

and a view **SupplierParts**, which contains the distinct part numbers that are supplied by at least one supplier:

```
CREATE VIEW SupplierParts (partNo)
AS SELECT DISTINCT partNo
    FROM Supplier s, Part p
    WHERE s.partNo = p.partNo;
```

Discuss how you would maintain this as a materialized view and under what circumstances you would be able to maintain the view without having to access the underlying base tables **Part** and **Supplier**.

7.20 Investigate the SQL dialect on any DBMS that you are currently using. Determine the system's compliance with the DDL statements in the ISO standard. Investigate the functionality of any extensions the DBMS supports. Are there any functions not supported?

7.21 Create the *DreamHome* rental database schema defined in Section 4.2.6 and insert the tuples shown in Figure 4.3.

7.22 Using the schema you have just created, run the SQL queries given in the examples in Chapter 6.

7.23 Create the schema for the Hotel schema given at the start of the exercises for Chapter 4 and insert some sample tuples. Now run the SQL queries that you produced for Exercises 6.7–6.28.

Case Study 2

For Exercises 7.24 to 7.40, use the Projects schema defined in the Exercises at the end of Chapter 5.

7.24 Create the Projects schema using the integrity enhancement features of SQL with the following constraints:

(a) **sex** must be one of the single characters 'M' or 'F'.
(b) **position** must be one of 'Manager', 'Team Leader', 'Analyst', or 'Software Developer'.
(c) **hoursWorked** must be an integer value between 0 and 40.

7.25 Create a view consisting of the **Employee** and **Department** tables without the **address**, **DOB**, and **sex** attributes.

7.26 Create a view consisting of the attributes empNo, fName, lName, projName, and hoursWorked attributes.

7.27 Consider the following view defined on the Projects schema:

```
CREATE VIEW EmpProject(empNo, projNo, totalHours)
AS SELECT w.empNo, w.projNo, SUM(hoursWorked)
FROM Employee e, Project p, WorksOn w
WHERE e.empNo = w.empNo AND p.projNo = w.projNo
GROUP BY w.empNo, w.projNo;
```

(a)
```
SELECT *
FROM EmpProject;
```
(b)
```
SELECT projNo
FROM EmpProject
WHERE projNo = 'SCCS';
```
(c)
```
SELECT COUNT(projNo)
FROM EmpProject
WHERE empNo = 'E1';
```
(d)
```
SELECT empNo, totalHours
FROM EmpProject
GROUP BY empNo;
```

General

7.28 Consider the following table:

Part (<u>partNo</u>, <u>contract</u>, partCost)

which represents the cost negotiated under each contract for a part (a part may have a different price under each contract). Now consider the following view **ExpensiveParts**, which contains the distinct part numbers for parts that cost more than £1000:

```
CREATE VIEW ExpensiveParts (partNo)
AS SELECT DISTINCT partNo
   FROM Part
   WHERE partCost > 1000;
```

Discuss how you would maintain this as a materialized view and under what circumstances you would be able to maintain the view without having to access the underlying base table **Part**.

© CourseSmart