CHAPTER 10

Database Creation

At the implementation stage of a database, the principal tasks include creating and modifying the database tables and other related structures, enforcing integrity constraints, populating the database tables, and specifying security and authorizations for access control and transaction processing control.

In this chapter we use the SQL-92 language standard in the code for the SQL statements. It is not necessary that a commercial DBMS implementation include all the standard SQL constructs or follow the standard language syntax verbatim for all SQL constructs. It is highly likely that commercial DBMS products differ in the implementation of at least some of the SQL syntax. Also, some vendors offer additional non-standard SQL constructs. Therefore, the reader using the SQL scripts[1] presented in this chapter may occasionally need to refer to the SQL reference material of the DBMS platform being used. To the extent that a DBMS product does not conform to a common syntactical standard, portability and migration across product platforms might be difficult.

10.1 Data Definition Using SQL

As noted in the introduction of Bart IV, a relation is called a *Cable* in SQL, and attributes and tuples are termed *columns* and *rows,* respectively. In reality, however, the formal object called a "relation" and the informal object called a "table" have several differences. For instance, a tabic can contain duplicate rows, while a relation is prohibited from having duplicate tuples. SQL tables are allowed to have null values while relations are not. The columns of a table have a left-to-right ordering, but the attributes of a relation in a relational data model are unordered. Columns of a table may end up with duplicate column names (e.g., the result of a join of two tables with the same column name) and sometimes, no column name (a derived column); this, however, is not permissible in a relation. The rows of a table have a top-to-bottom ordering, but the tuples of a relation do not. A table is usually expected to have at least one column, while technically a relation with no attributes is permissible. Notwithstanding many such differences, in the context of a relational data model, a table can represent a concrete picture of an abstract object called a relation—i.e., a table can "represent" a relation, but is not precisely equivalent to a relation. In addition to tables, SQL also supports other structures like *views, materialized views,* and *schemas.*

The data definition sublanguage of SQL, known as SQL/DDL, has three major constructs for creating and modifying databases: GREYYTE, ALTER, and DROP.

[1] A script is a command or series of commands usually stored in a file.

10.1.1 Base Table Specification in SQL/DDL

A relation schema (or alternatively a logical scheme) is specified in SQL/DDL by the CREATE TABLE statement. The result of a CREATE TABLE statement is referred to as a *base table* to indicate that the table is actually created, populated with rows of data, and stored as a physical file by the DBMS. This facilitates distinguishing base tables from other derived (or virtual) tables such as views or joined tables. The CREATE TABLE construct has a rich syntax that can specify domain constraints on columns, an entity integrity constraint (specification of the primary key), uniqueness constraints (specification of alternate keys), foreign key constraints, deletion and update rules, and more, in addition to the basic table name and column names. We present this SQL construct in gradual increments using appropriate examples. Let us begin with a simple scenario of patients in a clinic placing orders for medications. The ER model (the ER diagram plus a list of semantic integrity constraints that cannot be incorporated in the ER diagram), the relational schema, and the corresponding information-preserving logical schema appear in Figures 10.1a through 10. Id.

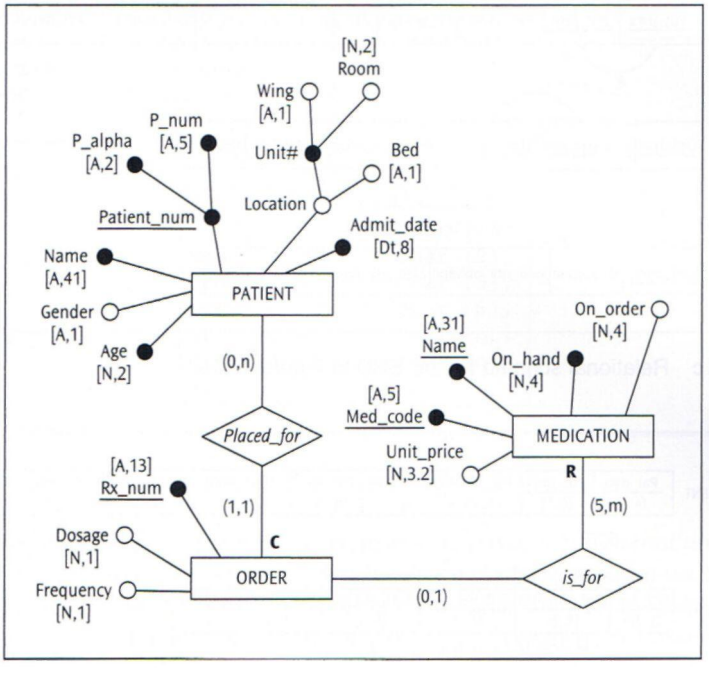


Figure 10.1a ER diagram: An excerpt from a hypothetical medical information system

> Constraint	Gender	IN ('M', 'F')
> Constraint	Age	IN (1 through 90)
> Constraint	Bed	IN ('A', 'B')
> Constraint	Unit_price	< 4.50
> Constraint	(Qty_onhand + Qty_onorder)	IN (1000 through 3000)
> Constraint	Dosage	DEFAULT 2
> Constraint	Dosage	IN (1 through 3)
> Constraint	Frequency	DEFAULT 1
> Constraint	Frequency	IN (1 through 3)

Figure 10.1b Semantic integrity constraints for the Fine-granular Design-Specific ER diagram

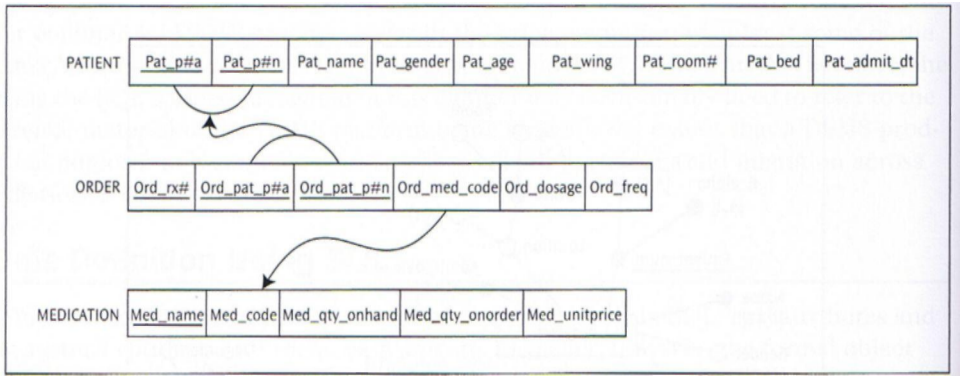


Figure 10.1c Relational schema for the ERD in Figure 10.1a

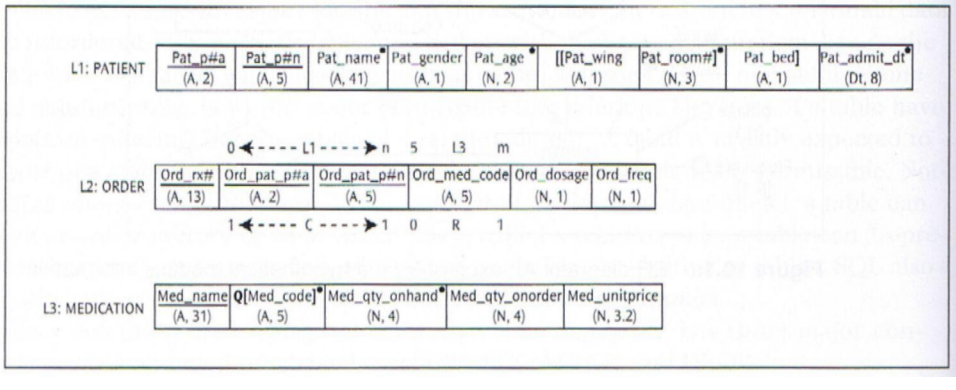
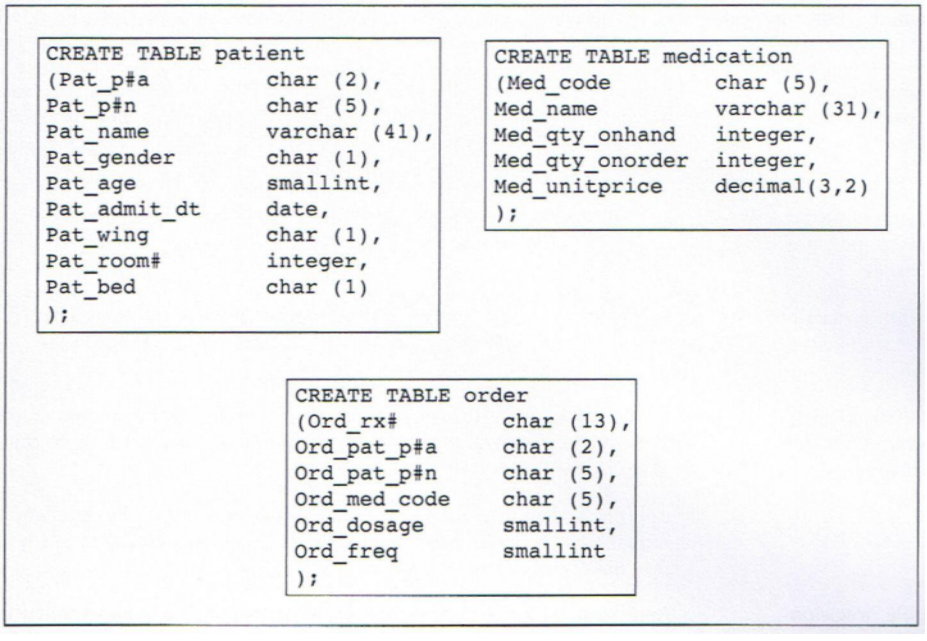


Figure 10.1d An information-preserving logical schema for the ERD in Figure 10.1a

The implementation of the design shown in Figures lO.la-d in a relational database entails creating the three base table structures first. At the minimum, the table specification includes the table name, the names of the columns in the table, and the data type for each column. The data type of a column is a part of the domain specification for the attribute and is sometimes referred to as the *type constraint* on the attribute. In other words, specifying the data type for a column is equivalent to imposing the type constraint on the attribute. The minimal form of a CREATE TABLE statement for the three base tables is shown in Box 1. SQL statements are *case-insensitive* while string values referenced in the SQL statements are *case-sensitive.*



```
CREATE TABLE patient
(Pat_p#a          char (2),
Pat_p#n           char (5),
Pat_name          varchar (41),
Pat_gender        char (1),
Pat_age           smallint,
Pat_admit_dt      date,
Pat_wing          char (1),
Pat_room#         integer,
Pat_bed           char (1)
);
```

```
CREATE TABLE medication
(Med_code         char (5),
Med_name          varchar (31),
Med_qty_onhand    integer,
Med_qty_onorder   integer,
Med_unitprice     decimal(3,2)
);
```

```
CREATE TABLE order
(Ord_rx#          char (13),
Ord_pat_p#a       char (2),
Ord_pat_p#n       char (5),
Ord_med_code      char (5),
Ord_dosage        smallint,
Ord_freq          smallint
);
```

Box 1

Observe that while it is legal to use the same attribute name in different entity types in an ER diagram, relational database theory stipulates that attribute names must be unique over the entire relational schema. Accordingly, the attribute names in the relational/logical schema in our example follow the naming convention proposed in Chapter 6. The column names of

a base table in SQL/DDL are considered to be ordered in the sequence in which they are specified in the CREATE TABLE syntax. However, when populated with data, the rows are not considered to be ordered within a base table. The data types supported by SQL-92 are grouped under *Number, String, Date/time,* and *Interval,* and are listed in Table 10.1. DBMS vendors often build their own data types based on these four categories.

Table 10.1 Data types supported by SQL-92

Number Data Types	
numeric (p, s) where **p** indicates precision and **s** indicates scale	Exact numeric type—literal representation of number values—*decimal portion exactly the size dictated by the scale*—storage length = (precision + 1) when scale is > 0—most DBMSs have an upper limit on **p** (e.g., 28)
decimal (p, s) where **p** indicates precision and **s** indicates scale	Exact numeric type—literal representation of number values—*decimal portion at least the size of the scale; but expandable to limit set by the specific DBMS*—storage length = (precision + 1) when scale is > 0 —most DBMSs have an upper limit on **p** (e.g., 28)
integer or **integer (p)** where **p** indicates precision	Exact numeric type—binary representation of large whole number values—often precision set by the DBMS vendor (e.g., 2 bytes)
smallint or **smallint (p)** where **p** indicates precision	Exact numeric type—binary representation of small whole number values—often precision set by the DBMS vendor (e.g., 1 byte)
float (p) where **p** indicates precision	Approximate numeric type—represents a given value in an exponential format—precision value represents the minimum size used, up to the maximum set by the DBMS
real	Approximate numeric type—represents a given value in an exponential format—has a default precision value set below that set for double precision data type by the DBMS
double precision	Approximate numeric type—represents a given value in an exponential format—has a default precision value set above that set for real data type by the DBMS
String Data Types	
character (ℓ) or **char (ℓ)** where ℓ indicates length	Fixed length character strings including blanks from the defined language set SQL_TEXT within a database—can be compared to other columns of the same type with different lengths or varchar type with different maximum lengths—most DBMS have an upper limit on ℓ (e.g., 255)
character varying (ℓ) or **char (ℓ) varying** or **varchar (ℓ)** where ℓ indicates the maximum length	Variable length character strings except trailing blanks from the defined language set SQL_TEXT within a database—DBMS records actual length of column values—can be compared to other columns of the same type with different maximum lengths or char type with different lengths—most DBMS have an upper limit on ℓ (e.g., 2000)

Table 10.1 Data types supported by SQL-92 (continued)

String Data Types (continued)	
bit (ℓ) where ℓ indicates length	Fixed length binary digits (0,1)—can be compared to other columns of the same type with different lengths or bit varying type with different maximum lengths
bit varying (ℓ) where ℓ indicates maximum length	Variable length binary digits (0,1)—can be compared to other columns of the same type with different maximum lengths or bit type with different lengths
Date/Time & Interval Data Types	
date	10 characters long—format: yyyy-mm-dd—can be compared to only other date type columns—allowable dates conform to the Gregorian calendar
time (p)	Format: hh:mi:ss—sometimes precision (p) specified to indicate fractions of a second—the length of a TIME value is 8 characters, if there are no fractions of a second. Otherwise, the length is 8, plus the precision, plus one for the delimiter: hh:mi:ss.p—if no precision is specified, it is 0 by default—TIME can only be compared to other TIME data type columns
timestamp (p)	Format: yyyy:mm:dd hh:mi:ss.p—a timestamp length is nineteen characters, plus the precision, plus one for the precision delimiter—timestamp can only be compared to other timestamp data type columns
interval (q)	Represents measure of time—there are two types of intervals: year-month (yyyy:mm) which stores the year and month; and day-time (dd hh:mi:ss) which stores the days, hours, minutes, and seconds—the qualifier (q) known in some databases as the interval lead precision, dictates whether the interval is year-month or day-time—implementation of the qualifier value varies

Note that the "CREATE TABLE order" statement in Box 1 will generate an error; the word "order" (or "ORDER," or any case of the word) cannot be used as a user-defined value for a table, column, or any construct in SQL because ORDER itself is an SQL construct and thus a reserved word. A list of SQL reserved words appears in Appendix C.

The CREATE TABLE statement is a single statement starting at "CREATE" and ending with a semicolon (;). The entire statement could be written on a single line, but spans multiple lines to enhance clarity and readability. The general form of the syntax for the CREATE TABLE statement is:

```
CREATE TABLE table_name (comma-delimited list of table-elements);
```

where:

- *table_name* **is a user-supplied name for the base table.**
- **Each** *table-element* **in the list is either a column definition or a constraint definition.**

There must be at least one column definition in order for a base table to exist. The basic svntax for a column definition is of the form:

```
column_name   representation   [default-definition]   [column-
constraint list]
```

where:

- *column_name* **is a user-supplied name for a column.**
- *representation* **specifies the relevant data type, or alternatively, the pre-defined** *domain-name.*

The optional *default-definition* **(optional elements are indicated by square brackets [ ]) specifies a default value for the column, which overrides any default value specified in the domain definition, if applicable. In the absence of an explicit default definition (directly or via the domain definition), the implicit assumption of a NULL value for the default prevails. The** *default-definition* **is of the form:**


```
DEFAULT ( literal | niladic-function[2] | NULL )
```

where:

- **the I implies "or"**
- **The optional column-constraint list specifies constraint-definition for the column.**

The basic syntax for a *constraint-definition* **follows the form:**

```
[ CONSTRAINT constraint_name ] constraint-definition[3]
```

Constraints in a base table arc declarative in nature and are imposed either on a single column in the tabic or on a set of columns in the table. The former is referred to as an *attribute-level* **or** *column-level constraint* **while the latter goes by the name** *tuple-level* **or** *row-level constraint.* **A** *constraint definition* **can be an independent tabic element (i.e., row-level constraint) or, if applicable to a specific column only, can be part of the column definition (i.e., column-level constraint). Constraint definitions include the following:**

- **The primary key definition—i.e., specification of an entity integrity constraint**

  ```
  PRIMARY KEY  (comma-delimited column list)
  ```

  Example:

  ```
  CONSTRAINT pk_pat PRIMARY KEY (Pat_p#a, Pat_p#n)
  ```

- **An alternate key definition—i.e., specification of a uniqueness constraint**

  ```
  UNIQUE  (comma-delimited column list)
  ```

  Example:

  ```
  CONSTRAINT  ung_med  UNIQUE  (Med_COde)
  ```

[2] A niladie-function is a built-in function that takes no arguments (Date and Darwen, 1997, p. 55). The niladic-functions that are allowed here are: USER, CURRENT-USER, SESSION-USER, SYSTEM-USER. CURRENT-DATE, CURRENT-TIME, CURRENT-TIMESTAMR

[3] The CONSTRAINT constraint._name phrase enclosed by [ ] is optional. However, in practice, it is *extremely* useful to use this phrase especially for ease of later references to the constraint by a name known to the creator of the table or individual responsible for altering the tabic. Wc strongly suggest mandatory use of this phrase in constraint specifications.

- **A foreign key constraint—i.e.. specification of a referential integrity constraint**

```
FOREIGN KEY (comma-delimited column list of referencing table)
•REFERENCES table_name
  (comma delimited column list of referenced table)[4]

[ referential triggered action clause ]
```

The *referential triggered action clause* facilitates specification of the action to be taken when the foreign key constraint is violated upon deletion of a referenced tuple or upon modification of the value of the referenced column list (primary key or alternate key as the case may be). The action options are: CASCADE, SET NULL, SET DEFAULT, and RESTRICT. These actions are referred to as reactive constraints, and must be qualified by the referential trigger ON DELETE or ON UPDATE. These action options are described and illustrated with examples in the context of a deletion referential trigger in Section 3.2.3 of Chapter 3 and are equally applicable to an update referential trigger.
Example:


```
CONSTRAINT fk_med FOREIGN KEY (Ord_med_code)
      REFERENCES medication (med_code)
      ON DELETE RESTRICT
      ON UPDATE CASCADE
```

- **A check constraint definition in order to restrict column or domain values[5]**

```
CHECK (conditional expression)
```

The *conditional expression* here can be of arbitrary complexity; it can even refer to other base tables in the database. Violation of a check constraint occurs when an attempt to create or modify a row in a base table causes the conditional expression stated in the constraint to evaluate as 'false'.
Example:

```
CONSTRAINT engender CHECK Gender IN ('M', 'F')
```

To enhance clarity, it is customary to include the column-level constraint definitions as clauses of the respective column definitions to which they belong, and to list only the row-level constraint definitions at the end of all the column definitions. This style of coding is demonstrated in the DDL code for the MEDICATION and ORDERS tables in Box 2. Notice that the name of the base table ORDER (SQL reserved word) has been changed to ORDERS (not a reserved word). An alternative practice is to code all constraint definitions at the end of all the column definitions, as shown in the DDL code for the PATIENT table in Box 2. Haphazard mixing of column definitions and constraint definitions in the DDL code is strictly discouraged in the interest of best programming practice.

[4] If the referenced column list is the primary key of the referenced table, the specification of this column list is optional.

[5] The CHECK clause is also used to specify conditional expressions in ***a general constraint*** called ASSERTION which is addressed in Chapter 12.

At this point, we are ready to add other constraints in the CPvEATE TABLE statement. Let us incorporate the integrity constraints specified in the relational schema (Figure 10.1c) along > with the semantic integrity constraints listed in Figure 10.1b in the SQL/DDL code we already have. The resulting script is shown in Box 2.'


```
CREATE TABLE patient
(Pat_p#a        char (2),
Pat_p#n         char (5),
Pat_name        varchar (41),
Pat_gender      char (1),
Pat_age         smallint,
Pat_admit_dt    date,
Pat_wing        char (1),
Pat_room#       integer,
Pat_bed         char (1),
CONSTRAINT pk_pat PRIMARY KEY (Pat_p#a, Pat_p#n),
CONSTRAINT chk_gender CHECK (Pat_gender IN ('M', 'F')),
CONSTRAINT chk_age CHECK (Pat_age IN (1 through 90)),
CONSTRAINT chk_bed CHECK (Pat_bed IN ('A', 'B'))
);


CREATE TABLE medication
(Med_code         char (5),
Med_name          varchar (31) CONSTRAINT pk_med  PRIMARY KEY,
Med_unitprice     decimal (3,2) CONSTRAINT chk_unitprice CHECK (Med_unitprice < 4.50),
Med_qty_onhand    integer,
Med_qty_onorder   integer,
CONSTRAINT chk_qty CHECK ((Med_qty_onhand + Med_qty_onorder) BETWEEN 1000 AND 3000)
);


CREATE TABLE orders
(Ord_rx#          char (13) CONSTRAINT pk_ord  PRIMARY KEY,
Ord_pat_p#a       char (2),
Ord_pat_p#n       char (5),
Ord_med_code      char (5) CONSTRAINT fk_med FOREIGN KEY REFERENCES medication (med_code),
Ord_dosage        smallint DEFAULT 2 CONSTRAINT chk_dosage CHECK (Ord_dosage BETWEEN 1 AND 3),
Ord_freq          smallint DEFAULT 1 CONSTRAINT chk_freq CHECK (Ord_freq IN (1,2,3)),
CONSTRAINT fk_pat FOREIGN KEY (Ord_pat_p#a, Ord_pat_p#n)
REFERENCES patient (Pat_p#a, Pat_p#n)
);
```

BOX 2

Observe that the DDL script produced on the basis of the relational schema (Figure 10.1c) does not fully capture all the information conveyed in the ERD. For instance, the optional property of some attributes, candidate keys of relation schemas, deletion rules, and participation constraints of relationships have not been mapped from the ERD to the relational schema and hence are not reflected in the DDL. An inspection of the information-preserving logical schema (Figure 10.Id) reveals that:

- Pathname, **Pat_age, Pat_admit_ dt, Med code,** and **Med qty onhand** are mandatory attributes, i.e., cannot have null values in any tuple.
- **Med^code** is the alternate key since **Med_**name has been chosen as the primary key of the MEDICATION table.

" *Note:* In other chapters in this book, tabic names are shown in all capital letters; that convention is preserved in the running text of this chapter ami other chapters of Part IV. However, in this chapter for clarity in distinguishing SQL keywords from table names, table names are sometimes shown in lower ease in SQL code.

- **Participation of ORDER in the *Placed J'or* relationship is total.**
- **Participation of PATIENT in the *Placed J'or* relationship is partial.**
- **Participation of ORDER in the *Is J'or* relationship is partial.**
- **Participation of MEDICATION in the *Is J'or* relationship is total.**
- **The deletion rule for the *Is J'or* relationship is restrict.**
- **The deletion rule for the *Placed J'or* relationship is cascade.**
- **[Pat_wing, Pat_room] is a composite attribute.**
- **[Pat_wing, Pat_room, Pat_bed] is a composite attribute.**

Note: **The cardinality ratio of the form (1, n) in a relationship type is implicitly captured in the DDL specification via the foreign key constraint. Any (1,1) cardinality ratio can be implemented using the UNIQUE constraint definition.**

At this point, let us rewrite the SQL/DDL script a third time so as to capture all these constraint definitions. The revised DDL script appears in Box 3, with the added constraint definitions highlighted.


```
CREATE TABLE patient
(Pat_p#a        char (2),
Pat_p#n         char (5),
Pat_name        varchar (41) constraint nn_Patnm not null,
Pat_gender      char (1),
Pat_age         smallint constraint nn_Patage not null,
Pat_admit_dt    date constraint nn_Patadmdt not null,
Pat_wing        char (1),
Pat_room#       integer,
Pat_bed         char (1),
CONSTRAINT pk_pat PRIMARY KEY (Pat_p#a, Pat_p#n),
CONSTRAINT chk_gender CHECK (Pat_gender IN ('M', 'F')),
CONSTRAINT chk_age CHECK (Pat_age IN (1 through 90)),
CONSTRAINT chk_bed CHECK (Pat_bed IN ('A', 'B'))
);


CREATE TABLE medication
(Med_code       char (5) CONSTRAINT nn_medcd not null CONSTRAINT unq_med UNIQUE,
Med_name        varchar (31) CONSTRAINT pk_med  PRIMARY KEY,
Med_unitprice   decimal (3,2) CONSTRAINT chk_unitprice CHECK (Med_unitprice < 4.50),
Med_qty_onhand  integer CONSTRAINT nn_medqty not null,
Med_qty_onorder integer,
CONSTRAINT chk_qty CHECK ((Med_qty_onhand + Med_qty_onorder) BETWEEN 1000 AND 3000)
);


CREATE TABLE orders
(Ord_rx#        char (13) CONSTRAINT pk_ord  PRIMARY KEY,
Ord_pat_p#a     char (2) CONSTRAINT nn_ord_pat_p#a not null,
Ord_pat_p#n     char (5) CONSTRAINT nn_ord_pat_p#n not null,
Ord_med_code    char (5) CONSTRAINT fk_med REFERENCES medication (Med_code)
ON DELETE RESTRICT ON UPDATE RESTRICT,
Ord_dosage      smallint DEFAULT 2 CONSTRAINT chk_dosage CHECK (Ord_dosage BETWEEN 1 AND 3),
Ord_freq        smallint DEFAULT 1 CONSTRAINT chk_freq CHECK (Ord_freq IN (1, 2, 3)),
CONSTRAINT fk_pat FOREIGN KEY (Ord_pat_p#a, Ord_pat_p#n)
REFERENCES patient (Pat_p#a, Pat_p#n) ON DELETE CASCADE ON UPDATE CASCADE
);
```

Box 3

By default, a column is allowed to have null values in the rows. The "not null" constraint definitions take care of the mandatory attribute value specification in the corresponding columns in the associated base tables. Likewise, unless explicitly prohibited, a column can contain the same value in multiple rows. An alternate key (i.e., a candidate key not chosen as the primary key of a base table) is required to enforce the uniqueness constraint. This is accomplished by the UNIQUE constraint definition for the Med_code column in the MEDICATION table.

Partial participation of a parent as well as a child in a relationship (i.e., min = 0) exists by default. Total participation of a parent in a relationship (i.e., min > 0) cannot be enforced using any of the constraint definitions discussed so far. SQL-92 offers another mechanism called *declarative assertion* to specify broader constraints at the database schema level. This is presented in Chapter 12. Total participation of a child in a relationship (i.e., min = 1) is enforced by specifying a "not null" constraint on the foreign key attribute(s) (e.g., not null constraint definition for (Ord_pat_p#a, Ord_pat_p#n) in the ORDERS table). The deletion rules (referential triggered action clause) are incorporated using the ON DELETE clause of the foreign key constraint definition.[7] Since, by definition, a relation schema has only atomic attributes, SQL/DDL does not provide for the specification of composite columns—all columns in a table are atomic.

10.1.1.2 Syntax of the ALTER TABLE Statement

At this point, we have successfully created the table structures for the logical schema constituting the base tables PATIENT, MEDICATION, and ORDERS. Suppose we want to make some corrections or changes to one or more of these base table structures. The ALTER TABLE statement in SQL/DDL is used to accomplish this. The general form of the syntax for the ALTER TABLE statement is:

```
ALTER TABLE table_name action;
```

where *tablejname* is the name of the base table being altered and *action* is one of the following:

- **Adding a column or altering a column's *default-definition* or removing the existing *default-definition* via the syntax:**

```
ADD [ COLUMN ] column_definition

ALTER [ COLUMN ] column_name
{ SET default-definition | DROP DEFAULT }[8]
```

- **Removing an existing column via the syntax:**

```
DROP [ COLUMN ] column_name { RESTRICT | CASCADE }
```

[7] Similar to the ON DELETE clause, SQL/DDL offers an ON UPDATE clause for referential triggered action which is intended for specifying action to he taken when the referenced attribute(s) value (primary key or alternate key value) in a foreign key constraint is changed. Since in our example this is not specified, we defaulted to an assumption of same as ON DELETE clause specifications.

[8] Braces { } are used to specify that one of the items from the list of items separated by the vertical bar must be chosen.

The REVOKE statement is used to take away all or some of the privileges previously granted to another user or users. The format of the REVOKE statement is:

```
REVOKE [GRANT OPTION FOR] {Privilege-list | ALL PRIVILEGES}
ON Object-name
FROM [{User-list | 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. CASCADE means that if USER_A, whose user name appears in the list *User-list,* has granted those privileges to USER_B, the privileges granted to USER_B are also revoked. If USER_B has granted those privileges to USER_C, those privileges are revoked as well, and so on. The option RESTRICT indicates that if any such dependent privileges exist, the REVOKE statement will fail.

10.3.2 Some Examples of Granting and Revoking Privileges[15]

Suppose that the six users (USER_A, USER_B, USER_C, USER_D, USER_E, and USER_F) exist and that USER_A owns the PATIENT, MEDICATION, and ORDERS tables. Let's experiment by observing what happens as a result of USER_A's granting and subsequently revoking privileges on these three tables.

Example 1. USER_A grants an assortment of privileges on the ORDERS, PATIENT, and MEDICATION tables to USER_B, USER_C, and USER_D. Note the information about these privileges recorded in the System Catalog that contains a record of USER_A's grants.

```
GRANT SELECT, INSERT, DELETE, UPDATE
ON ORDERS
TO USER_B;

Grant succeeded.

GRANT SELECT, INSERT
ON PATIENT
TO USER_C;

Grant succeeded.

GRANT INSERT, DELETE
ON MEDICATION
TO USER_D;

Grant succeeded.
```

USER_A's table privileges granted as recorded in the System Catalog

GRANTEE	OWNER	TABLE_NAME	GRANTOR	PRIVILEGE	GRANTABLE
USER_B	USER_A	ORDERS	USER_A	DELETE	NO
USER_B	USER_A	ORDERS	USER_A	INSERT	NO
USER_B	USER_A	ORDERS	USER_A	SELECT	NO
USER_B	USER_A	ORDERS	USER_A	UPDATE	NO

[15] The examples in this section adhere to the syntax in the SQL-92 standard and may require some slight modifications to run on certain DBMS platforms. In addition, some of the error messages are abridged versions of those that might be generated by a particular DBMS product.

- **Adding to an existing set of constraints via the syntax:**

```
ADD  table_constraint_definition
```

- **'Removing a named constraint via the syntax:**

```
DROP CONSTRAINT constraint_name { RESTRICT | CASCADE }
```

Suppose we want to add a column to the base table PATIENT to store the phone number of every patient. The SQL/DDL code to do this is:

```
ALTER TABLE patient  ADD Pat_phone#      char (10);
```

Now the rows of the base table PATIENT are capable of receiving values for the Pat_phone# column. Since no default has been specified for the new column added to the table, the rows of PATIENT will, by default, have "null" value for the column, Pat_phone#. Clearly, it is not possible to specify a "not null" constraint on the column until either a non-null default value is specified for the column or the column is populated with non-null values in all rows of the base table.

The column can be removed from the base table by either of these two statements:

```
ALTER TABLE patient  DROP Pat_phone# CASCADE;
```

or:

```
ALTER TABLE patient  DROP Pat_phone# RESTRICT;
```

Observe the definition of DROP behavior (i.e., CASCADE or RESTRICT) in the SQE/DDL statement. The SQL-92 standard requires the DROP behavior definition. The CASCADE option implies that all constraints and derived tables that reference the column also be dropped from the database schema. Likewise, the RESTRICT option prevents the deletion of the column should any schema element that references the column exist. Also, SQL-92 provides for the deletion of only one column per ALTER statement.

Suppose we want to specify a default value of `$3.00` for the unit price of all medications. This can be done as follows:

```
ALTER TABLE medication  ALTER Med_unitprice SET DEFAULT 3.00;
```

The default clause can be removed by:

```
ALTER TABLE medication  ALTER Med_unitprice DROP DEFAULT;
```

10.1.1.3 A Best Practice Hint

Let us take a look at two methods of specifying the domain constraints on the column Pat_age in the base table PATIENT. Here is the first method:

```
Pat_age        smallint CONSTRAINT nn_Patage not null,
CONSTRAINT chk_age CHECK  (Pat_age  IN  (1  through  90))
```

Since the naming of the constraint definition is optional, it is possible to write the above DDL using a second method:

```
Pat_age        smallint not null CHECK  (Pat_age  IN  (1  through  90))
```

Clearly, the second method code appears simpler and more concise. However, if we decide to permit null values for Pat_age, in method 2, the entire column definition has to be re-specified. On the other hand, using method 1, we simply drop the "not null" constraint as shown below:

```
ALTER TABLE patient  DROP CONSTRAINT nn_patage CASCADE;
```

or:

```
ALTER TABLE patient  DROP CONSTRAINT nn_patage RESTRICT;
```

This is possible only because we named the constraint. While the DBMS names every constraint when we don't, finding out the constraint name given by the DBMS is an inefficient task; and the constraint name given by the DBMS is not generally a user-friendly name. Thus, the coding technique of method 1 offers greater flexibility and is strongly recommended.

10.1.1.4 Syntax of the DROP TABLE Statement

Just as we can create a base table and make certain changes to it, it is also possible to remove a base table (structure and content) from the database using the DROP TABLE SQL/DDL statement:

```
DROP TABLE table_name drop behavior;
```

where:

- *table_name* **is the name for the base table being deleted.**
- **The** *drop behaviors* **possible are CASCADE and RESTRICT.**

For example,

```
DROP TABLE medication CASCADE;
```

not only deletes the MEDICATION table (all rows and the table definition) from the database, but also removes all schema elements (constraints and derived tables) that reference any column of the MEDICATION table. For instance, the constraint definition, fk_med in the base table ORDERS, defines the foreign key constraint that references a candidate key of the MEDICATION table. The CASCADE option in the DROP TABLE statement automatically drops the constraint fk_med in ORDERS when the MEDICATION table is dropped. The RESTRICT option, on the other hand, disallows the deletion of the MEDICATION table, because the constraint definition, fk_med, exists in the schema.

10.1.1.5 A Comprehensive Example

Figure 10.2 contains a Fine-granular Design-Specific ER diagram for Madeira College which emerged as a result of use of the conceptual modeling process described in Part I of this book. This database is used to keep track of the courses offered by departments, the enrollment of students in courses, and the teaching-related activities of professors.

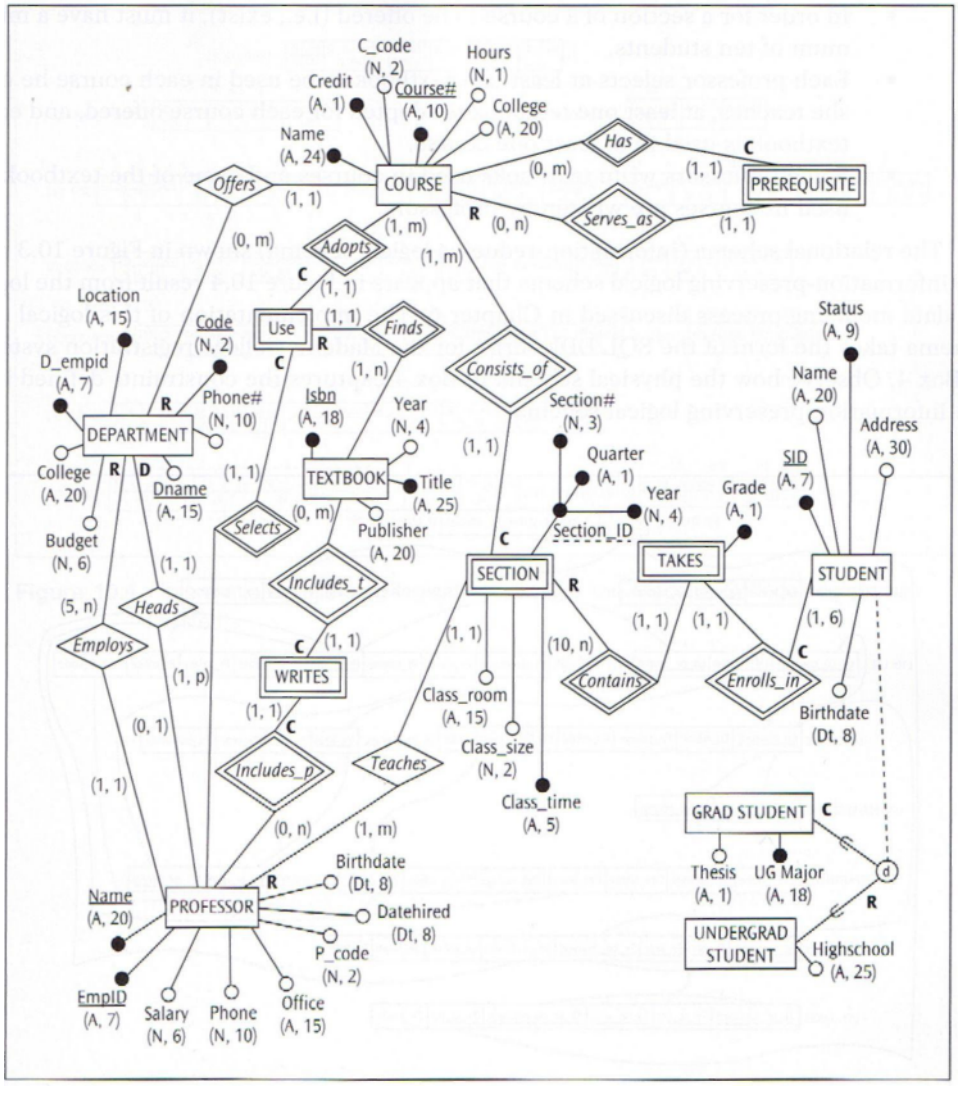


Figure 10.2 A Fine-granular Design-Specific ER diagram for the Madeira College registration system

A brief description of the various relationships follows:

- **Departments can offer a variety of courses with each offered by a single department.**
- **Each department must have at least five professors, one of whom serves as the department head.**
- **Each course consists of at least one section and may serve as a prerequisite of other courses as well as have other courses as its prerequisite.**
- **Madeira College has both graduate and undergraduate students. Each student must enroll in (i.e., take) at least one but no more than six courses.**

- In order for a section of a course to be offered (i.e., exist), it must have a minimum of ten students.
- Each professor selects at least one textbook to be used in each course he or she teaches, at least one textbook is adopted for each course offered, and each textbook is used in at least one course.
- Some professors write textbooks used in courses and some of the textbooks used in courses are written by professors.

The relational schema (information-reducing logical schema) shown in Figure 10.3 and the information-preserving logical schema that appears in Figure 10.4 result from the logical data modeling process discussed in Chapter 6. The implementation of this logical schema takes the form of the SQL/DDL script for the Madeira College registration system in Box 4. Observe how the physical schema in Box 4 captures the constraints defined in the information-preserving logical schema.

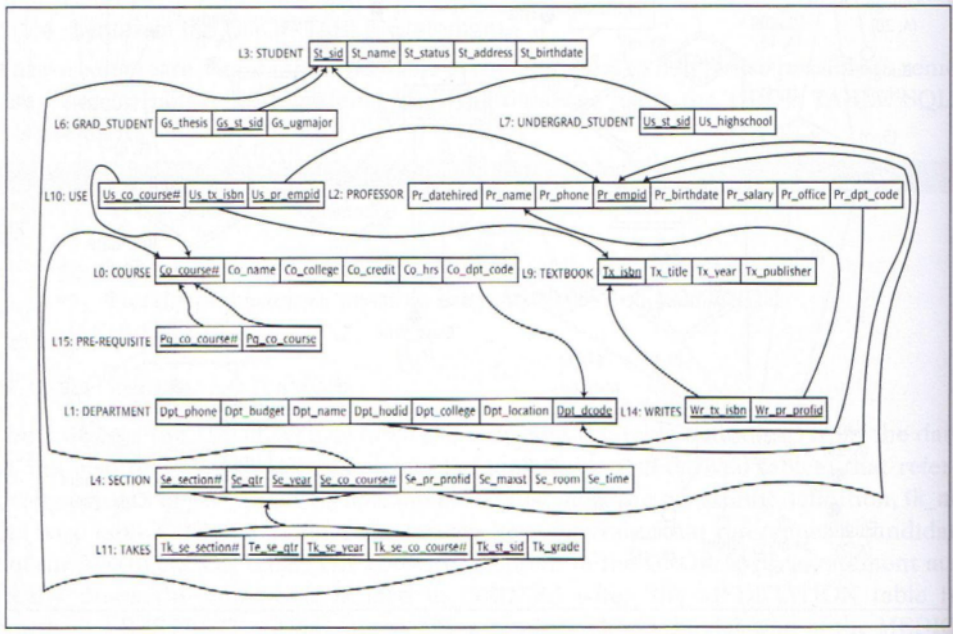


Figure 10.3 Information-reducing logical schema for the Madeira College registration system

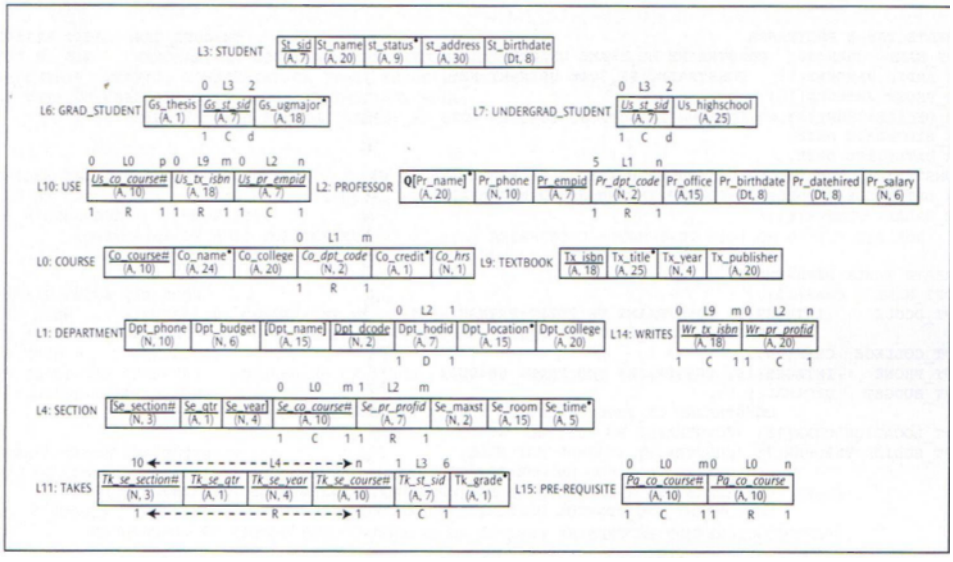


Figure 10.4 Information-preserving logical schema for the Madeira College registration system

```
CREATE TABLE PROFESSOR
(PR_NAME  CHAR(20)  CONSTRAINT NN_3NAME UNIQUE,
PR_EMPID VARCHAR(7)   CONSTRAINT PK_3COU PRIMARY KEY,
PR_PHONE INTEGER(10),
PR_OFFICE CHAR(15),
PR_BIRTHDATE DATE,
PR_DATEHIRED DATE,
CONSTRAINT CK_DATES CHECK (PR_DATEHIRED - PR_BIRTHDATE >= 0),
PR_DPT_DCODE  NUMBER(2) CONSTRAINT NN_3DCOD NOT NULL,
PR_SALARY NUMBER(6));


CREATE TABLE DEPARTMENT
(DPT_NAME   CHAR(15),
DPT_DCODE     INTEGER(2) CONSTRAINT PK_2DEPT PRIMARY KEY,
                  CONSTRAINT CK_2COD CHECK(DPT_DCODE BETWEEN 1 AND 10),
DPT_COLLEGE  CHAR(20),
DPT_PHONE    INTEGER(10) CONSTRAINT UNQ_PHONE UNIQUE,
DPT_BUDGET   DECIMAL(6,0),
                  CONSTRAINT CK_2BUD CHECK(DPT_BUDGET BETWEEN 100000 AND 500000),
DPT_LOCATION CHAR(15)  CONSTRAINT NN_2LOC NOT NULL,
DPT_HODID  VARCHAR(7)  CONSTRAINT NN_2HOD NOT NULL,
                  CONSTRAINT UNQ_2HOD UNIQUE(DPT_HODID),
                  CONSTRAINT FK_2DEPT FOREIGN KEY(DPT_HODID) REFERENCES PROFESSOR(PR_EMPID),
                  CONSTRAINT UNQ_2DNMCOL UNIQUE(DPT_NAME,DPT_COLLEGE));


ALTER TABLE PROFESSOR ADD
CONSTRAINT FK_3DEPT FOREIGN KEY(PR_DPT_DCODE) REFERENCES DEPARTMENT(DPT_DCODE);


CREATE TABLE COURSE
(CO_NAME   CHAR(24)  CONSTRAINT NN_1NAME NOT NULL,
CO_COURSE# CHAR(10)  CONSTRAINT PK_1COU PRIMARY KEY,
CO_CREDIT  CHAR(1)   CONSTRAINT NN_1CRED NOT NULL,
                  CONSTRAINT CK_1CRED CHECK(UPPER(CO_CREDIT) IN ('U','G')),
CO_COLLEGE CHAR(20)  CONSTRAINT CK_1COL CHECK(CO_COLLEGE IN ('ARTS AND SCIENCES','EDUCATION',
'ENGINEERING','BUSINESS')),
CO_HRS   SMALLINT,
CO_DPT_DCODE   INTEGER(2) CONSTRAINT NN_1DCOD NOT NULL,
CONSTRAINT UNQ_1COLNAM UNIQUE(CO_NAME,CO_COLLEGE),
CONSTRAINT FK_1DEP FOREIGN KEY(CO_DPT_DCODE) REFERENCES DEPARTMENT(DPT_DCODE),
CONSTRAINT CK_1HRS CHECK(((UPPER(CO_CREDIT) IN ('U')) AND (CO_HRS<=4)) OR
                        ((UPPER(CO_CREDIT) IN ('G')) AND (CO_HRS=3))));


CREATE TABLE STUDENT
(ST_SID             VARCHAR(7)   CONSTRAINT PK_4STD PRIMARY KEY,
ST_NAME             CHAR(20),
ST_ADDRESS          CHAR(30),
ST_STATUS           CHAR(9),
ST_BIRTHDATE DATE,
                CONSTRAINT CK_4STAT CHECK(ST_STATUS IN ('FULL TIME','PART TIME')));


CREATE TABLE SECTION
(SE_SECTION#        INTEGER(3),
SE_QTR          CHAR(1)   CONSTRAINT CK_5QTR CHECK(SE_QTR IN ('A','W','S','U')),
SE_YEAR             INTEGER(4),
SE_TIME       CHAR(5)   CONSTRAINT NN_5TIM NOT NULL,
SE_MAXST       INTEGER(2) CONSTRAINT CK_5SIZ CHECK(SE_MAXST <= 35),
SE_ROOM       VARCHAR(15),
SE_CO_COURSE#          CHAR(10),
         CONSTRAINT FK_5COU FOREIGN KEY(SE_CO_COURSE#) REFERENCES COURSE(CO_COURSE#) ON DELETE
CASCADE,
SE_PR_PROFID           VARCHAR(7) CONSTRAINT NN_5PROF NOT NULL,
                    CONSTRAINT PK_5SEC PRIMARY KEY(SE_SECTION#,SE_QTR,SE_YEAR,SE_CO_COURSE#),
                    CONSTRAINT FK_5DEPT FOREIGN KEY(SE_PR_PROFID) REFERENCES PROFESSOR(PR_EMPID),
                    CONSTRAINT UNQ_5SECT UNIQUE(SE_QTR,SE_YEAR,SE_TIME,SE_ROOM));
```

Box 4

```
CREATE TABLE GRAD_STUDENT
(GS_ST_SID    VARCHAR(7)  CONSTRAINT PK_7SID PRIMARY KEY,
GS_THESIS  CHAR(1) CONSTRAINT CK_7THES CHECK(GS_THESIS IN ('Y','N')),
GS_UGMAJOR CHAR(18) CONSTRAINT NN_7MAJ NOT NULL,
       CONSTRAINT FK_7STD FOREIGN KEY(GS_ST_SID) REFERENCES STUDENT(ST_SID) ON DELETE CASCADE);

CREATE TABLE UNDERGRAD_STUDENT
(US_ST_SID            VARCHAR(7) CONSTRAINT PK_8SID PRIMARY KEY,
US_HIGHSCHOOL      CHAR(25),
       CONSTRAINT FK_8STD FOREIGN KEY(US_ST_SID) REFERENCES STUDENT(ST_SID) ON DELETE CASCADE);


CREATE TABLE TEXTBOOK
(TX_ISBN     VARCHAR(18) CONSTRAINT PK_10TB PRIMARY KEY,
TX_TITLE     VARCHAR(25)   CONSTRAINT NN_10TIT NOT NULL,
TX_YEAR      INTEGER(4),
TX_PUBLISHER CHAR(20)   CONSTRAINT CK_10PUB CHECK(TX_PUBLISHER IN ('Thomson','Springer',
'Prentice-Hall')));


CREATE TABLE PREREQUISITE
(PQ_CO_COURSE#  CHAR(10),
         CONSTRAINT FK_11APRE FOREIGN KEY(PQ_CO_COURSE#) REFERENCES COURSE(CO_COURSE#),
PQ_CO_COURSE    CHAR(10),
         CONSTRAINT FK_11BPRE FOREIGN KEY(PQ_CO_COURSE) REFERENCES COURSE(CO_COURSE#),
         CONSTRAINT PK_11PRE PRIMARY KEY(PQ_CO_COURSE#,PQ_CO_COURSE),
         CONSTRAINT CK_11PRE CHECK(PQ_CO_COURSE# <> PQ_CO_COURSE));


CREATE TABLE USE
(US_CO_COURSE#        CHAR(10),
US_TX_ISBN            VARCHAR(18),
US_PR_EMPID           VARCHAR(7),
CONSTRAINT PK_12USE PRIMARY KEY(US_CO_COURSE#,US_TX_ISBN,US_PR_EMPID),
CONSTRAINT FK_12AUSE FOREIGN KEY(US_CO_COURSE#) REFERENCES COURSE(CO_COURSE#),
CONSTRAINT FK_12BUSE FOREIGN KEY(US_TX_ISBN) REFERENCES TEXTBOOK(TX_ISBN),
CONSTRAINT FK_12CUSE FOREIGN KEY(US_PR_EMPID) REFERENCES PROFESSOR(PR_EMPID));


CREATE TABLE WRITES
(WR_TX_ISBN          VARCHAR(18),
WR_PR_PROFID          VARCHAR(7),
CONSTRAINT PK_13WRT PRIMARY KEY(WR_TX_ISBN,WR_PR_PROFID),
CONSTRAINT FK_13AWRT FOREIGN KEY(WR_TX_ISBN) REFERENCES TEXTBOOK(TX_ISBN) ON DELETE CASCADE,
CONSTRAINT FK_13BWRT FOREIGN KEY(WR_PR_NAME) REFERENCES PROFESSOR(PR_NAME));


CREATE TABLE TAKES
(TK_SE_SECTION# INTEGER(3),
TK_SE_QTR   CHAR(1),
TK_SE_YEAR      INTEGER(4),
TK_SE_CO_COURSE#   CHAR(10),
TK_GRADE     CHAR(1)   CONSTRAINT NN_14GRADE NOT NULL,
                  CONSTRAINT CK_14GRADE CHECK(UPPER(TK_GRADE) IN ('A','B','C')),
TK_ST_SID       VARCHAR(7),
                  CONSTRAINT FK_14SID FOREIGN KEY(TK_ST_SID) REFERENCES STUDENT(ST_SID) ON
DELETE            CASCADE,
                  CONSTRAINT FK_14TAK FOREIGN
KEY(TK_SE_SECTION#,TK_SE_QTR,TK_SE_YEAR,TK_SE_CO_COURSE#)  REFERENCES
SECTION(SE_SECTION#,SE_QTR,SE_YEAR,SE_CO_COURSE#),
                  CONSTRAINT PK_14TAK PRIMARY
KEY(TK_SE_SECTION#,TK_SE_QTR,TK_SE_YEAR,TK_SE_CO_COURSE#,TK_ST_SID));
```

Box 4 (continued)

10.1.2 Specification of User-Defined Domains

The SQL-92 standard provides for the formal specification of a *domain.* A domain specification can be used to define a constraint over one or more columns of a table with a formal name so that the domain name can be used wherever that constraint is applicable. In other words, a domain may be regarded as a named collection of data values that can be treated as a user-defined data type in a column definition. This approach lends the ability for the design to be modular. The CREATE DOMAIN statement is of the form:

```
CREATE DOMAIN domain_name [ AS ] data type [ default-definition ]
     [ domain-constraint-definition list ];
```

where:

- *domainjname* **is a user-supplied name for the domain.**
- **The optional** *default-definition* **specifies a default value for the domain. In the absence of an explicit default definition, the domain has no default value.**
- **The optional** *domain-constraint-definition* **is of the form:**


```
[ CONSTRAINT constraint_name ] CHECK (VALUE (conditional-expression))
```

Note: **The keyword VALUE is required here; it is not used in any other SQL/DDL statement.**

Several examples of creating a domain follow.

Example 1. Specify a domain to capture integer values 1, 2, and 3, with a default value of 2.

```
CREATE DOMAIN measure smallint DEFAULT 2 CONSTRAINT chk_measure
CHECK (VALUE IN (1,2,3));
```

Since [CONSTRAINT *constraint jname* **] is optional, the above statement can also be stated as:**

```
CREATE DOMAIN measure smallint DEFAULT 2 CHECK (VALUE IN (1,2,3));
```

Although the latter definition appears short and crisp, the former expression is better practice since it provides a direct referencing mechanism to the constraint definition within the domain definition; for instance, it gives the ability to delete the CHECK constraint while retaining the rest of the domain definition. Now, any column definition that is a number in the range of 1—3 and can accept a default value of 2 can use this domain name as the *representation* **for the column instead of the** *data type* **and a** *constraint on the data type* **for the range of values. For instance, the column definition in the ORDERS table:**

```
Ord_dosage smallint DEFAULT 2 CONSTRAINT chk_dosage
CHECK (Ord_dosage BETWEEN 1 AND 3)
```

can be coded as:

```
Ord_dosage  measure
```

Likewise.

```
Ord_freq      smallint DEFAULT 1 CONSTRAINT chk_freq
CHECK (Ord_freq IN (1, 2, 3))
```

can also be coded as:

```
Ord_freq      measure DEFAULT 1
```

Accordingly, Ord_dosage has a default value of 2 propagated from the domain measure and honors the acceptable range of values, 1 through 3. Ord_freq, on the other hand, honors the acceptable range of values, 1 through 3 based on the domain measure it is referencing; but the default value of 2 propagated from measure is overridden by the local default value of 1 specified as part of its column definition. The revised SQL/DDL is highlighted in Box 5.


```
CREATE TABLE orders
 (Ord_rx#          char (13) CONSTRAINT pk_ord  PRIMARY KEY,
Ord_pat_p#a        char (2), CONSTRAINT nn_ord_pat_p#a not null,
Ord_pat_p#n        char (5), CONSTRAINT nn_ord_pat_p#n not null,
Ord_med_code       char (5) CONSTRAINT fk_med REFERENCES medication (Med_code)
ON DELETE RESTRICT ON UPDATE RESTRICT,
Ord_dosage         measure,
Ord_freq           measure DEFAULT 1,
CONSTRAINT fk_pat FOREIGN KEY (Ord_pat_p#a, Ord_pat_p#n)
REFERENCES patient (Pat_p#a, Pat_p#n) ON DELETE CASCADE ON UPDATE CASCADE
);
```

Box 5

Example 2. Specify a domain with mandatory values for the U.S. Postal Service abbreviation for the list of states OH, PA, IL, IN, KY, WV, and MI. Also, designate OH as the default state.

```
CREATE DOMAIN valid_States CHAR (2) DEFAULT 'OH'
    CONSTRAINT nn_states CHECK (VALUE IS NOT NULL)
    CONSTRAINT chk_States CHECK (VALUE IN ('OH', 'PA', 'IL', 'IN', 'KY',
    'WV', 'MI'));
```

This domain name can now be used as the *representation* for any column name in the database that represents a 2-character attribute—hopefully the State attributes in different base tables. Obviously, the default designation cannot contradict the constraint definition on the DOMAIN—i.e., the CREATE DOMAIN operation will fail if the specified DEFAULT value is TX since TX is not present in the list of valid states shown in the constraint definition. Note that:

```
CREATE DOMAIN valid_Slat.es CHAR (2) NO: NUM.;
```

is incorrect syntax, while:

```
CREATE DOMAIN valid_States CHAR (2) CHECK (VALUE IS NOT NULL);
```

is correct syntax.

A predefined *domain name* comes in handy when it can be used in several tables without having to repeat the complete data type and column constraint specification. A domain is a schema element (object) of a database schema like a base table. Thus, changes to the domain definition automatically propagate to all tables in which the domain name is referenced. To demonstrate this, let us follow up on Example 2. While most columns in a database that represent the attribute **State** cannot have a missing value for **State,** suppose a few are allowed to have null values. One way to handle this situation is to let the DOMAIN **valid_states** include a null value and let the individual column definitions specify "not null" as needed. This way, any constraint specification in a column definition overrides the corresponding constraint in the domain definition referenced by the column for that particular column; the rest of the domain definition continues to apply for the referencing column. Therefore, now we need to remove the "not null" constraint from the DOMAIN **valid_states.**

Example 3. Remove the "not null" constraint from the domain specification for **valid_states.** The SQL/DDL statement that accomplishes this task is ALTER **DOMAIN:**

```
ALTER DOMAIN valid_states DROP CONSTRAINT nn_states;
```

Note: Had we not named the constraint, the only recourse we would have is to drop the complete domain definition and create it over again.

The effect of the above ALTER DOMAIN statement is that all columns in the database schema which reference the domain **valid_states** as their column representation will now accept a missing value except, of course, the ones that enforce a local column constraint explicitly prohibiting a missing value.

The general form of the syntax for the ALTER DOMAIN statement is:

```
ALTER DOMAIN domain_name action;
```

where *domain_name* is the name of the DOMAIN being altered and the *actions* possible are:

- Adding *default-definition* or replacing an existing *default-definition via* the syntax:

  ```
  SET default-definition
  ```

- Copying *default-definition* to the columns defined on the domain which do not have explicitly specified default values of their own, then removing *default-definition* from the domain definition via the syntax:

  ```
  DROP DEFAULT
  ```

- Adding to the existing set of constraints, if any, via the syntax:

  ```
  ADD domain_constraint_definition
  ```

- Removing the named constraint via the syntax:

  ```
  DROP CONSTRAINT constraint_name
  ```

Note: **Only one action per ALTER DOMAIN statement is allowed.**

Example 4. **Add Man-land (MD) and Virginia (VA) to the domain valid_states. This is done by adding a new constraint to the domain valid_states as shown here:**

```
ALTER DOMAIN valid_states ADD CONSTRAINT chk_2more
CHECK (VALUE IN ('MD', 'VA'));
```

Note that this constraint labeled chk_2more does not replace the existing chk_states constraint in the domain valid_states. Instead, it is included as an additional constraint of the domain to allow 'MD' and 'VA as valid states of the domain. Having altered the domain valicLstates thus, what if we decide that 'VA is not a valid state, but 'MD' is? In this case, the only recourse is to remove the constraint chk_2more and add a new constraint checking only for 'MD' in the valid_states domain.

Example 5. **Assuming that the scripts in Example 1 and Box 5 have been executed, remove the default value for the DOMAIN measure. As of now, the default for measure is 2, the default for Ord_dosage is 2, and the default for Ord_freq is 1. The SQL/DDL syntax follows:**


```
ALTER DOMAIN measure DROP DEFAULT;
```

The execution of the above code yields the following result. The DOMAIN measure no longer has a default value—*not even nidi,* **Ord_dosage continues to have a default value of 2; and Ord_freq continues to have a default value of 1. The verification of this result is left as an exercise for the reader.**

Example 6. **Change the default value of State in the valid_states domain to PA.**

```
ALTER DOMAIN valid_states SET DEFAULT 'PA';
```

A domain definition can be eliminated using the DROP DOMAIN SQL/DDL syntax:

```
DROP DOMAIN domain_name CASCADE | RESTRICT
```

The rules for the DROP behavior in this case are somewhat different and should be noted carefully. With the RESTRICT option any attempt to drop a domain definition fails if any column in the database tables, views, and/or integrity constraints references the domain name. With the CASCADE option, however, dropping a domain entails dropping of any referencing views and integrity constraints only. The columns referencing the domain are *not* **dropped. Instead the domain constraints are effectively converted into base table constraints and are attached to every base table that has a column(s) defined on the domain.**

Example 7. Suppose the domain named measure is no longer needed and the following DROP DOMAIN statement is used:

```
DROP DOMAIN measure RESTRICT;
```

In this case, since the columns Ord_dosage and Ord_freq are defined on the DOMAIN measure, the DROP DOMAIN operation will fail with the DROP behavior specification of RESTRICT.

On the other hand, suppose the following DROP DOMAIN statement is used:

```
DROP DOMAIN measure CASCADE;
```

In this case, since the columns **Ord_dosage** and **Ord_freq** are defined on the DOMAIN **measure,** the DROP DOMAIN operation first creates row-level constraints in the base table ORDERS equivalent to the domain constraint **chk_measure.** The default definition of the DOMAIN **measure** is mapped to the column **Ord_dosage** since it does not have a column-level default definition. The column-level default definition of **Ord_freq** stays intact.

10.1.3 Schema and Catalog Concepts in SQL/DDL

An SQL-schema represents a named collection of schema elements (objects) such as base table definitions, domain definitions, view definitions, constraint definitions, and so on under the control of a single user (or application) who alone is authorized to create and access objects within it. This concept is introduced in the SQL-92 standard. Before SQL-92, the database objects of all users belonged to the same schema/database. A named collection of SQL-schema in an SQL environment is referred to as a catalog in SQL-92—i.e., the DBMS partitions the catalog into SQL-schemas. The purpose of a catalog is to better organize database elements into groups by application/user. In general, only a few users, such as a database administrator (DBA), are authorized to create an SQL-schema. The privilege to create an SQL-schema and other schema elements must be explicitly granted to other users by the DBA.

10.1.3.1 CREATE SCHEMA

SQL-schemas are created by the `CREATE SCHEMA` statement using the following syntax:

```
CREATE SCHEMA { [ schema_name ] [AUTHORIZATION user_name ] }
       [ schema element list ];
```

where:

- ***schema jname*** is a user supplied name for the SQL-schema.
- *user_name* is intended to identify the owner of the SQL-schema.
- *schema element list* includes objects such as base table definitions, domain definitions, view definitions, constraint definitions, etc.

Note: At least one of *schemajname* and *userjiame* must be present in the definition: however, it is a good practice to include both.

Example:

```
CREATE SCHEMA clinic AUTHORIZATION Debakey;
```

Observe that the inclusion of *schema element list* in the CREATE SCHEMA script is optional. In other words, an SQL-schema can be created without any schema elements in which case the schema elements will be added as and when required because the creation of schema elements as independent operations is valid in SQL. A sample script for the creation of an SQL-schema containing an assortment of schema elements appears in Box 6.

```
CREATE SCHEMA clinic AUTHORIZATION Debakey
CREATE TABLE patient
(Pat_p#a        char (2),
Pat_p#n         char (5),
Pat_name        varchar (41) constraint nn_Patnm not null,
Pat_gender      char (1),
Pat_age         smallint constraint nn_Patage not null,
Pat_admit_dt    date constraint nn_Patadmdt not null,
Pat_wing        char (1),
Pat_room#       integer,
Pat_bed         char (1),
CONSTRAINT pk_pat PRIMARY KEY (Pat_p#a, Pat_p#n),
CONSTRAINT chk_gender CHECK (Pat_gender IN ('M', 'F')),
CONSTRAINT chk_age CHECK (Pat_age IN (1 through 90)),
CONSTRAINT chk_bed CHECK (Pat_bed IN ('A', 'B'))
)

CREATE VIEW senior_citizen AS
  SELECT patient.Pat_name, patient.Pat_age, patient.Pat_gender
    FROM patient
  WHERE patient.Pat_age > 64

CREATE VIEW senior_stat (V_gender, V_#ofpats) AS
  SELECT patient.Pat_gender, count (*)
    FROM patient
  WHERE patient.Pat_age > 64
  GROUP BY patient.Pat_gender

CREATE TABLE medication
(Med_code         char (5) CONSTRAINT nn_medcd not null CONSTRAINT unq_med UNIQUE,
Med_name          varchar (31) CONSTRAINT pk_med  PRIMARY KEY,
Med_unitprice     decimal (3,2) CONSTRAINT chk_unitprice CHECK (Med_unitprice < 4.50),
Med_qty_onhand    integer CONSTRAINT nn_medqty not null,
Med_qty_onorder   integer,
CONSTRAINT chk_qty CHECK ((Med_qty_onhand + Med_qty_onorder) BETWEEN 1000 AND 3000)
)

CREATE VIEW unused_med AS
  SELECT medication.Med_name, medication.Med_code, medication.Med_qty_onhand
    FROM medication
  WHERE medication.Med_code NOT IN
    (SELECT orders.Ord_med_code FROM orders)
WITH CHECK OPTION

CREATE TABLE orders
 (Ord_rx#      char (13) CONSTRAINT pk_ord  PRIMARY KEY,
Ord_pat_p#a    char (2),
Ord_pat_p#n    char (5),
Ord_med_code   char (5) CONSTRAINT fk_med REFERENCES medication (Med_code)
ON DELETE RESTRICT ON UPDATE RESTRICT,
Ord_dosage        measure,
Ord_freq          measure,
CONSTRAINT fk_pat FOREIGN KEY (Ord_pat_p#a, Ord_pat_p#n)
REFERENCES patient (Pat_p#a, Pat_p#n) ON DELETE CASCADE ON UPDATE CASCADE
)
```

Box 6

```
CREATE DOMAIN measure AS smallint CHECK (measure > 0 and measure < 4)

CREATE VIEW used_med AS
  SELECT orders.Ord_pat_p#a, orders.Ord_pat_p#n, medication.Med_name,
orders.Ord_dosage,
              orders.Ord_frequency
    FROM medication, orders
  WHERE medication.Med_code = orders.Ord_med_code

CREATE ASSERTION Chk_orders
            CHECK (SELECT COUNT (*) FROM orders >= 100)

CREATE ASSERTION Chk_ordr_per_med
CHECK (NOT EXISTS
(SELECT * FROM medication
 WHERE medication.Med_code NOT IN
     (SELECT medx.Med_code  FROM medication medx
         WHERE medx.Med_code IN
        (SELECT Ord_med_code FROM orders
           GROUP BY Ord_med_code
                                    HAVING COUNT (*) >= 5)))
                        )

CREATE ASSERTION Chk_unit#
CHECK (NOT EXISTS
(SELECT * FROM patient
 WHERE Pat_wing IS NULL AND Pat_room# IS NULL
                          )

CREATE ASSERTION Chk_no_ordr_pats
CHECK (NOT EXISTS
(SELECT * FROM patient
 WHERE (Pat_p#a, Pat_p#n) NOT IN
     (SELECT Ord_pat_p#a, Ord_pat_p#n   FROM orders))
                         )

CREATE TRIGGER Pat_discharge_dt
AFTER DELETE ON patient
FOR EACH ROW
INSERT INTO PATIENT_AUDIT VALUES (Pat_p#a, Pat_p#a, SYSDATE);

);
```

Box 6 (continued)

A'ote: Assertions, triggers, and views are covered in Chapter 12 because they require some knowledge of the SQL querying language.

10.1.3.2 DROP SCHEMA

The SQL-92 DROP SCHE.NL\ syntax for deleting an SQL-schema is:

```
DROP SCHEMA schema_name { CASCADE | RESTRICT };
```

Example:

```
DROP SCHEMA clinic RESTRICT;
```

If RESTRICT is specified as the drop behavior, the DROP operation will fail unless the SQL-schema is empty. If CASCADE is specified as the drop behavior, the SQL-schema and all the objects contained in it will be eradicated from the database.

10.1.3.3 The INFORMATION_SCHEMA

The SQL-92 standard does not stipulate the structure of a catalog or the format of the SQL-schemas in a catalog. However, an idealized catalog structure for a DBMS product to emulate has been proposed. Known as the definition schema in the SQL-92 standard, this idealized catalog structure defines a set of system-level tables (see Groff and Weinberg, 2002). While not requiring a DBMS product to support the idealized catalog, SQL-92 does require that every system catalog contain one particular schema named 1NF0RMATI0N_SCHEMA in order to claim adherence to intermediate or full SQL-92 compliance level. The INFORMATION_SCHEMA essentially defines a series of views on the idealized catalog tables that identify database objects accessible to current users in the catalog. A list of these catalog views in the INFORMATION_SCHEMA is also available in Groff and Weinberg (2002). All object definitions from all the SQL-schemas in a catalog are also captured in the INFORMATION_SCHEMA via the defined catalog views. Only authorized users may access the INFORMATION_SGHEMA of a catalog. Also, SQL-schemas within the same catalog are allowed to share many of the schema elements. Finally, SQL-92 also presents the concept of a cluster of catalogs in the SQL-92 environment. The CLUSTER identifies the set of databases a single SQL program can access.

10.2 Data Population Using SQL

Every database is subject to continual change. Three SQL statements (INSERT, DELETE, and UPDATE) are used for data insertion, deletion, and modification. This section discusses these three statements in the context of the PATIENT, MEDICATION, and ORDERS tables introduced Section 10.1.1. For convenience of the reader, the SQL/DDL script from Box 3 in Section 10.1.1.1 is reproduced as Box 7.

```
CREATE TABLE patient
(Pat_p#a        char (2),
Pat_p#n         char (5),
Pat_name        varchar (41) constraint nn_Patnm not null,
Pat_gender      char (1),
Pat_age         smallint constraint nn_Patage not null,
Pat_admit_dt    date constraint nn_Patadmdt not null,
Pat_wing        char (1),
Pat_room#       integer,
Pat_bed         char (1),
CONSTRAINT pk_pat PRIMARY KEY (Pat_p#a, Pat_p#n),
CONSTRAINT chk_gender CHECK (Pat_gender IN ('M', 'F')),
CONSTRAINT chk_age CHECK (Pat_age IN (1 through 90)),
CONSTRAINT chk_bed CHECK (Pat_bed IN ('A', 'B'))
);


CREATE TABLE medication
(Med_code        char (5) CONSTRAINT nn_medcd not null CONSTRAINT unq_med UNIQUE,
Med_name         varchar (31) CONSTRAINT pk_med  PRIMARY KEY,
Med_unitprice    decimal (3,2) CONSTRAINT chk_unitprice CHECK (Med_unitprice < 4.50),
Med_qty_onhand   integer CONSTRAINT nn_medqty not null,
Med_qty_onorder integer,
CONSTRAINT chk_qty CHECK ((Med_qty_onhand + Med_qty_onorder) BETWEEN 1000 AND 3000)
);


CREATE TABLE orders
(Ord_rx#         char (13) CONSTRAINT pk_ord  PRIMARY KEY,
Ord_pat_p#a      char (2) CONSTRAINT nn_ord_pat_p#a not null,
Ord_pat_p#n      char (5) CONSTRAINT nn_ord_pat_p#n not null,
Ord_med_code     char (5) CONSTRAINT fk_med REFERENCES medication (Med_code)
ON DELETE RESTRICT ON UPDATE RESTRICT,
Ord_dosage       smallint DEFAULT 2 CONSTRAINT chk_dosage CHECK (Ord_dosage BETWEEN 1 AND 3),
Ord_freq         smallint DEFAULT 1 CONSTRAINT chk_freq CHECK (Ord_freq IN (1, 2, 3)),
CONSTRAINT fk_pat FOREIGN KEY (Ord_pat_p#a, Ord_pat_p#n)
REFERENCES patient (Pat_p#a, Pat_p#n) ON DELETE CASCADE ON UPDATE CASCADE
);
```

Box 7

10.2.1 The INSERT Statement

The SQL-92 standard provides two types of INSERT statements to add new rows of data to a database:

- **Single-row INSERT—adds a single row of data to a table.**
- **Multi-row INSERT—extracts rows of data from another part of the database and adds them to a table.**

These statements take the following forms:

```
INSERT INTO <table-name> [(column-name {, column-name})]
VALUES (expression {, expression})
```

```
INSERT INTO <table-name> [(column-name {, column-name})]
<select-statement>[9]
```

[9] The SQL SELECT statement is used to retrieve (i.e., **query)** data from tables. In its simplest form, SELECT • FROM *table_name,* all columns from the table_name listed are retrieved. The remainder of this chapter contains several examples that make use of this form of the SQL SELECT statement. Chapters 11 and 12 contain an extensive discussion of the SQL SELECT statement.

The values should be listed in the same order in which they are specified in the CREATE TABLE statement. The following three INSERT statements add one row to the PATIENT, MEDICATION, and ORDERS tables. Note that SQL allows us to omit the column names from the INSERT statement when assigning a value to each column in the table.

```
INSERT INTO PATIENT VALUES ('DB','77642','Davis, Bill', 'M', 27, '2007-07-
07', 'B', 108, 'B');

1 row created.

INSERT INTO MEDICATION VALUES ('TAG', 'Tagament', 3.00, 3000, 0);

1 row created.

INSERT INTO ORDERS VALUES ('104', 'DB', '77642', 'TAG', 3, 1);

1 row created.
```


Each of these INSERT statements is successful only because each honors the declarative constraints established in the respective CREATE TABLE statements.

It is also permissible for an INSERT statement to specify explicit column names that correspond to the values provided in the INSERT statement. This is useful if a table has a number of columns but only a few columns are assigned values in a particular new row. Example 1 inserts a row into the PATIENT table that contains only the patient number, patient name, age, and date of admission. In this case, specification of the column names is required.

Example 1.

```
INSERT INTO PATIENT (PATIENT.PAT_P#A, PATIENT.PAT_P#N, PATIENT.PAT_
NAME, PATIENT.PAT_AGE, PATIENT.PAT_ADMIT_DT) VALUES ('GD','72222','Grimes,
David', 44, '2007-07-12');

1 row created.
```

This INSERT statement was successful because each of the columns not listed in the INSERT statement permits null values.[11] For example, had an attempt been made to insert a patient without specifying a date of admission, the INSERT statement would have failed. Thus every row in the PATIENT table must contain a patient name, age, and date of admission. In addition, since the patient number consisting of the combination of PATIENT.Pat_p#a and PATIENT.Pat_p#n constitutes the primary key, these two columns must be defined as well.

The '2007-07-07' character string represents the date of admission of the patient. For a date data type, SQL-92 uses a default date format where the first four digits represent the year component, the next two digits (1-12) represent the month component, and final two digits (as constrained by the rules of the Gregorian calendar) represent the day of month (see Table 10.1). Other formats for representing dates are covered in Chapter 12.

Although not illustrated here, it is also permissible to omit columns with a DEFAULT value. If a DEFAULT exists for *a* column not explicitly listed in the INSERT statement, the default value will also be included for this column when the row is inserted.

Each order must involve both an existing patient and existing medication. Observe what happens in Example 2 when an attempt is made to insert an order for an existing patient (David Grimes) but nonexistent medication (KEF).

Example 2.

```
INSERT INTO ORDERS VALUES ('109', 'GD', '72222', 'KEF', 1, 1);
 integrity constraint FK_MED violated—parent key not found
```

Note that FK.MED (see Box 7) is the name of the referential integrity constraint requiring each medication code in the ORDERS table to exist in the MEDICATION table.

The multi-row INSERT statement adds multiple rows of data to a table via the execution of a query. In this form of the INSERT statement, the data values for the new rows appear in a SELECT statement specified as part of the INSERT statement. Suppose, for example, separate patient tables exist for different hospitals within the same hospital system. The INSERT statement in Example 3 inserts all rows in the PATIENT_SUGARLAND table into the PATIENT table. Since the PATIENTJSUGARLAND table has only six columns while the PATIENT table has nine columns, the column names are specified in the INSERT statement.

Example 3.

```
INSERT INTO PATIENT
(PATIENT.PAT_P#A,PATIENT.PAT_P#N,PATIENT.PAT_NAME,PATIENT.PAT_GENDER,PATIENT.PAT_AGE,
PATIENT. PAT_ADMIT_DT)
       SELECT * FROM PATIENT_SUGARLAND;

3 rows created.

SELECT * FROM PATIENT;
```

Pat_p#a	Pat_p#n	Pat_nome	Pat_gender	Pat_age	Pat_admit_dt	Pat_wing	Pat_room#	Pat_bed
DB	77642	Davis, Bill	M	27	2007-07-07	B	108	B
GD	72222	Grimes. David		44	2007-07-12			
LH	97384	Lisauckis. Hal	M	69	2008-06-06			
HJ	99182	Hargrove, Jan	F	21	2008-05-25			
RN	31678	Robins, Nancy	F	57	2008-06-01			

Had there been an interest in inserting only those rows in the PATIENT_SUGARLAND table with a date of admission after June 1, 2008, a WHERE clause referencing the appropriate column name in the PATIENT_SUGARLAND table could have been added to the SELECT statement in the INSERT statement given above.

10.2.2 The DELETE Statement

The DELETE statement removes selected rows of data from a single table and takes the following form:

```
DELETE FROM <table-name> [WHERE <search-condition>]
```

Since the WHERE clause in a DELETE statement is optional, a DELETE statement of the form DELETE FROM <*tahle-name*> can be used to delete all rows in a table. When used in this manner, while the target table has no rows after execution of the deletion, the table still exists and new rows can still be inserted into the table with the INSERT statement. To erase the table definition from the database, the DROP TABLE statement (described in Section 10.1.1.3) must be used.

Care must be exercised when using the DELETE statement. While rows are deleted from only one table at a time, the deletion may propagate to rows in other tables if actions are specified in the referential integrity constraints. For example, the constraint in the ORDERS table:

```
constraint fk_pat foreign key (Ord_pat_p#a, Ord_pat_p#n)
references patient (Pat_p#a, Pat_p#n) ON DELETE CASCADE ON UPDATE CASCADE
```

results in the deletion of all orders for a particular patient when that patient is deleted from the PATIENT table. The DELETE statement in Example 1 illustrates the propagation of a deletion to another table by deleting the first patient inserted into the PATIENT table, Bill Davis. Observe the content of the PATIENT and ORDERS tables before and after the deletion.

Content of Tables Prior to Deletion

```
SELECT * FROM PATIENT;
Pat_p#a  Pat_p#n  Pat_name         Pat_gender    Pat_age Pat_admit_dt Pat_wing  Pat_room# Pat_bed
-------- -------- ------------------- ---------- ---------- ------------ -------- ---------- ---
DB       77642    Davis, Bill      M                  27 2007-07-07   B               108 B
GD       72222    Grimes, David                       44 2007-07-12

SELECT * FROM MEDICATION;
Med_code Med_name                        Med_unitprice Med_qty_onhand Med_qty_onorder
 -------- ------------------------------ ------------- -------------- ---------------
TAG      Tagament                                    3           3000               0

SELECT * FROM ORDERS;
Ord_rx        Ord_pat_p#a  Ord_pat_p#n  Ord_med_code Ord_dosage   Ord_freq
 ------------ ------------ ------------ ------------ ---------- ----------
 104          DB           77642        TAG                   3          1
```

Example 1.

```
DELETE FROM PATIENT WHERE PATIENT.PAT_NAME LIKE '%Davis, Bill%';[12]

1 row deleted.
```

Content of Tables After Deletion

```
SELECT * FROM PATIENT;
Pat_p#a  Pat_p#n Pat_name         Pat_gender    Pat_age Pat_admit_dt Pat_wing  Pat_room# Pat_bed
------- ------- ------------------ ---------- ---------- ------------ -------- ---------- ------
GD       72222   Grimes, David                       44 2007-07-12

 SELECT * FROM MEDICATION;
 Med_code Med_name                        Med_unitprice Med_qty_onhand Med_qty_onorder
 -------- ------------------------------ ------------- -------------- ---------------
 TAG      Tagament                                    3           3000               0

 SELECT * FROM ORDERS;
 no rows selected
```

[12] The LIKE operator and the percent character (%) are used for pattern matching. Pattern matching in SQL is discussed in Chapter 11.

On the other hand, observe the effect of the constraint in the ORDERS table:

```
Ord_med_code char(5) constraint fk_med references medication (Med_code)
,ON DELETE RESTRICT ON UPDATE RESTRICT
```

when the attempt is made in Example 2 below to delete a medication for which one or more orders exists. Assume that the rows previously deleted from the PATIENT and ORDERS tables have been reinserted prior to the execution of the DELETE statement which attempts to delete the Tagament medication from the MEDICATION table.

Example 2.

```
DELETE FROM MEDICATION WHERE MEDICATION.MED_CODE = 'TAG';
integrity constraint (FK_MED) violated - child record found
```

Note that FK_MED is the name of the referential integrity constraint requiring each medication code in the ORDERS table to exist in the MEDICATION table and restricting the deletion of a medication with one or more orders.

10.2.3 The UPDATE Statement

The UPDATE statement modifies the values of one or more columns in selected rows of a single table. The UPDATE statement takes the following form:

```
UPDATE <table-name>
SET column-name = expression
        {, column-name = expression}
[WHERE <search-condition>]
```

The SET clause specifics which columns are to be updated and calculates the new values for the columns.

It is important that an UPDATE statement not violate any existing constraints. See Example 1.

Example 1.

```
UPDATE MEDICATION SET MEDICATION.MED_UNITPRICE = 5.00
WHERE MEDICATION.MED_CODE = 'TAG';
```

The UPDATE statement in this example violates the check constraint GHK_UNITPRTCE and thus generates the following message:

```
check constraint (CHK_UNITPRICE) violated
```

Several rows can be modified by a single UPDATE statement. As an example, suppose the MEDICATION table now contains the following six rows:

```
SELECT * FROM MEDICATION;
```

Med_code	Med_name	Med_unitprice	Med_qty_onhand	Med_qty_onorder
TAG	Tagament	3	3000	0
VIB	Vibramycin	1.5	1700	300
KEF	Keflin	2.5	900	410
ASP	Aspirin	.02	3000	0
PCN	Penicillin	.4	2700	0
VAL	Valium	.75	2100	0

Observe the effect of the UPDATE statement in Example 2 designed to add 500 to the quantity on hand for each medication with a unit price greater than 0.50.

Example 2.

```
UPDATE MEDICATION
SET MEDICATION.MED_QTY_ONHAND = MEDICATION.MED_QTY_ONHAND + 500
WHERE MEDICATION.MED_UNITPRICE > 0.50;

check constraint (CHK_QTY) violated

SELECT * FROM MEDICATION;
```

Med_code	Med_name	Med_unitprice	Med_qty_onhand	Med_qty_onorder
TAG	Tagament	3	3000	0
VIB	Vibramycin	1.5	1700	300
KEF	Keflin	2.5	900	410
ASP	Aspirin	.02	3000	0
PCN	Penicillin	.4	2700	0
VAL	Valium	.75	2100	0

While the CHK_QTY constraint requiring the quantity on hand plus the quantity on order is violated for only one of the four otherwise qualifying rows, none of the four rows is updated. When the WHERE clause excludes Tagament, as shown in Example 3, the UPDATE is successful.

Example 3.

```
UPDATE MEDICATION
SET MEDICATION.MED_QTY_ONHAND = MEDICATION.MED_QTY_ONHAND + 500
WHERE MEDICATION.MED_UNITPRICE > 0.50 AND MEDICATION.MED_CODE <> 'TAG';

3 rows updated.

SELECT * FROM MEDICATION;
```

Med_code	Med_name	Med_unitprice	Med_qty_onhand	Med_qty_onorder
TAG	Tagament	3	3000	0
VIB	Vibramycin	1.5	2200	300
KEF	Keflin	2.5	1400	410
ASP	Aspirin	.02	3000	0
PCN	Penicillin	.4	2700	0
VAL	Valium	.75	2600	0

10.3 Access Control in the SQL-92 Standard

In a database environment, the DBMS must ensure that only authenticated users[13] are authorized to access the database and that they are allowed to access only the information that has been specifically made available to them. Privileges constitute the set of actions a user is permitted to carry out on a table or a view. The privileges defined by the SQL-92 standard arc:

- **SELECT—Permission to retrieve data from a table or view**
- **INSERT—Permission to insert rows into a table or view**

[13] An authenticated user is one who has demonstrated, by providing a password and perhaps meeting a series of other requirements, the right to access information in a database.

- UPDATE—Permission to modify column values of a row in a table or view
- DELETE—Permission to delete rows of data in a table or view
- REFERENCES—Permission to reference columns of a table named in integrity constraints
- USAGE—Permission to use domains, collation sequences, character sets, and translations[14]

The UPDATE privilege can be restricted to specific columns of a table, making it possible to change these columns but disallowing changes to any other columns. A similar column list associated with the INSERT privilege restricts the grantee to supply values only for the listed columns while inserting a row. Likewise, the REFERENCES privilege can be restricted to specific columns of a table, allowing these columns to be referenced in foreign key constraints and check constraints but disallowing other columns from being referenced.

Many DBMS products offer additional privileges beyond those specified in the SQL-92 standard. For example, Oracle and DB2 support ALTER and INDEX privileges for tables. With the ALTER privilege on a given table, a user can use the ALTER TABLE statement to modify the definition (i.e., structure) of a table. A user with the INDEX privilege on a given table can create an index for that table with the CREATE INDEX statement. DBMS products without the ALTER and INDEX privileges only allow the user who created the table to use the ALTER TABLE and CREATE INDEX statements.

10.3.1 The GRANT and REVOKE Statements

The GRANT statement is used to grant privileges on database objects to specific users. The format of the GRANT statement is:

```
GRANT {Privilege-list |ALL PRIVILEGES}
ON Object-name
TO  {User-list | PUBLIC}
[WITH GRANT OPTION]
```

Privilege-list consists of one or more of the following privileges, separated by commas:

```
SELECT
DELETE
INSERT [(Column-name [, ... ])]
UPDATE [(Column-name [, ... ])]
REFERENCES [(Column-name [, ... ])]
USAGE
```

Object-name can be the name of a base table, view, domain, character set, collation sequence, or translation.

The keywords ALL PRIVILEGES and PUBLIC are shortcuts that can be used when j granting all privileges to all users. Note that when privileges are given to the PUBLIC they are granted to all present and future authorized users, not just to the users currently known to the DBMS. The WITH GRANT OPTION clause allows for the horizontal propagation of privileges by permitting the user(s) in the designated *User-list* to pass on the privileges they have been given for the named object to other users.

[14] Collation sequences, character sets, and translations are not covered in this book and thus the USAGE privilege is not included in the remainder of this discussion. The interested reader may wish to reference Cannan and Otten (1993) for a discussion of these database structures.

```
USER_C    USER_A    PATIENT     USER_A    INSERT    NO
USER_C    USER_A    PATIENT     USER_A    SELECT    NO
USER_D    USER_A    MEDICATION  USER_A    DELETE    NO
USER_D    USER_A    MEDICATION  USER_A    INSERT    NO
```

Example 2. USER_A grants all privileges on the MEDICATION table to the PUBLIC. This means that USER_D has received the DELETE and INSERT privileges on the MEDICATION table twice (once explicitly and once implicitly as a member of the PUBLIC). Thus should USER_A choose to revoke all privileges on the MEDICATION table from the PUBLIC, USER_D would still maintain the DELETE and INSERT privileges.

```
GRANT ALL PRIVILEGES
ON MEDICATION
TO PUBLIC;

Grant succeeded.
```


USER_A's table privileges granted as recorded in the System Catalog (PUBLIC'S privileges on the MEDICATION table have been added)

GRANTEE	OWNER	TABLE_NAME	GRANTOR	PRIVILEGE	GRANTABLE
PUBLIC	USER_A	MEDICATION	USER_A	DELETE	NO
PUBLIC	USER_A	MEDICATION	USER_A	INSERT	NO
PUBLIC	USER_A	MEDICATION	USER_A	SELECT	NO
PUBLIC	USER_A	MEDICATION	USER_A	UPDATE	NO
PUBLIC	USER_A	MEDICATION	USER_A	REFERENCES	NO
USER_B	USER_A	ORDERS	USER_A	DELETE	NO
USER_B	USER_A	ORDERS	USER_A	INSERT	NO
USER_B	USER_A	ORDERS	USER_A	SELECT	NO
USER_B	USER_A	ORDERS	USER_A	UPDATE	NO
USER_C	USER_A	PATIENT	USER_A	INSERT	NO
USER_C	USER_A	PATIENT	USER_A	SELECT	NO
USER_D	USER_A	MEDICATION	USER_A	DELETE	NO
USER_D	USER_A	MEDICATION	USER_A	INSERT	NO

Example 3. USER_A has not granted USER_B any privileges on the PATIENT table. The following GRANT statement allows USER_B to retrieve (i.e., select) rows from USER_A's PATIENT table and also grant the SELECT privilege to other users.

```
GRANT SELECT
ON PATIENT
TO USER_B
WITH GRANT OPTION;

Grant succeeded.
```

Example 4. At this point assume USER_B is connected to the database and attempts to grant the SELECT privilege received from USER_A to USER_D. Note how the first attempt fails because USER_B failed to qualify the table name (PATIENT) with the name of its owner (USER_A). After the successful grant, the system catalog now records the fact that USERJB has granted the SELECT privilege to USER_D.

```
GRANT SELECT
ON PATIENT
TO USER_D;

ERROR at line 2:  table PATIENT does not exist
```

```
GRANT SELECT
ON USER_A.PATIENT
TO USER_D;'

Grant succeeded.
```

USER_B's table privileges granted as recorded in the System Catalog

GRANTEE	OWNER	TABLE_NAME	GRANTOR	PRIVILEGE	GRANTABLE
USER_D	USER_A	PATIENT	USER_B	SELECT	NO

As a result of the GRANT statement in Example 3, the System Catalog now records USER_A's granting of the SELECT privilege to USER_B on the PATIENT table as grantable. In addition, it also records USER_B's granting of the SELECT privilege on the PATIENT table to USER_D.

USER_A's table privileges granted as recorded in the System Catalog

GRANTEE	OWNER	TABLE_NAME	GRANTOR	PRIVILEGE	GRANTABLE
PUBLIC	USER_A	MEDICATION	USER_A	DELETE	NO
PUBLIC	USER_A	MEDICATION	USER_A	INSERT	NO
PUBLIC	USER_A	MEDICATION	USER_A	SELECT	NO
PUBLIC	USER_A	MEDICATION	USER_A	UPDATE	NO
PUBLIC	USER_A	MEDICATION	USER_A	REFERENCES	NO
USER_B	USER_A	PATIENT	USER_A	SELECT	YES
USER_B	USER_A	ORDERS	USER_A	DELETE	NO
USER_B	USER_A	ORDERS	USER_A	INSERT	NO
USER_B	USER_A	ORDERS	USER_A	SELECT	NO
USER_B	USER_A	ORDERS	USER_A	UPDATE	NO
USER_C	USER_A	PATIENT	USER_A	INSERT	NO
USER_C	USER_A	PATIENT	USER_A	SELECT	NO
USER_D	USER_A	PATIENT	USER_B	SELECT	NO
USER_D	USER_A	MEDICATION	USER_A	DELETE	NO
USER_D	USER_A	MEDICATION	USER_A	INSERT	NO

Another table in the System Catalog records the five table privileges received by USER B from USER A.

USER_B's table privileges received as recorded in the System Catalog

OWNER	TABLE_NAME	GRANTOR	PRIVILEGE	GRANTABLE
USER_A	PATIENT	USER_A	SELECT	YES
USER_A	ORDERS	USER_A	DELETE	NO
USER_A	ORDERS	USER_A	INSERT	NO
USER_A	ORDERS	USER_A	SELECT	NO
USER_A	ORDERS	USER_A	UPDATE	NO

Example 5. Privileges can be granted on specific columns of a table as well as on all columns. Here USER_A grants USER_E the UPDATE privilege on three columns of the PATIENT table. Notice that, as shown below, a record of column privileges granted by USER_A is recorded in a different table in the System Catalog.

```
GRANT UPDATE (PAT_WING, PAT_ROOM#, PAT_BED)
ON PATIENT
'TO USER_E;

Grant succeeded.
```

USER_A's table privileges granted as recorded in the System Catalog

GRANTEE	OWNER	TABLE_NAME	GRANTOR	PRIVILEGE	GRANTABLE
PUBLIC	USER_A	MEDICATION	USER_A	DELETE	NO
PUBLIC	USER_A	MEDICATION	USER_A	INSERT	NO
PUBLIC	USER_A	MEDICATION	USER_A	SELECT	NO
PUBLIC	USER_A	MEDICATION	USER_A	UPDATE	NO
PUBLIC	USER_A	MEDICATION	USER_A	REFERENCES	NO
USER_B	USER_A	PATIENT	USER_A	SELECT	YES
USER_B	USER_A	ORDERS	USER_A	DELETE	NO
USER_B	USER_A	ORDERS	USER_A	INSERT	NO
USER_B	USER_A	ORDERS	USER_A	SELECT	NO
USER_B	USER_A	ORDERS	USER_A	UPDATE	NO
USER_C	USER_A	PATIENT	USER_A	INSERT	NO
USER_C	USER_A	PATIENT	USER_A	SELECT	NO
USER_D	USER_A	PATIENT	USER_B	SELECT	NO
USER_D	USER_A	MEDICATION	USER_A	DELETE	NO
USER_D	USER_A	MEDICATION	USER_A	INSERT	NO

Observe how the record of the column privileges granted by USER_A to USER_E is recorded in a separate part of the System Catalog.

USER_A's column privileges granted as recorded in the System Catalog

GRANTEE	OWNER	TABLE_NAME	COLUMN_NAME	GRANTOR	PRIVILEGE	GRANTABLE
USER_E	USER_A	PATIENT	PAT_WING	USER_A	UPDATE	NO
USER_E	USER_A	PATIENT	PAT_ROOM#	USER_A	UPDATE	NO
USER_E	USER_A	PATIENT	PAT_BED	USER_A	UPDATE	NO

When connected as USER_E, we can observe how the System Catalog also records the column privileges on the PATIENT table received from (i.e., granted by) USER_A. Note that at this point USER_E has not received any explicit table privileges but only the three column privileges on the PATIENT table. Of course, USER_E does have all privileges on the MEDICATION table received by virtue of being a member of the PUBLIC.

USER_E's column privileges received as recorded in the System Catalog

OWNER	TABLE_NAME	COLUMN_NAME	GRANTOR	PRIVILEGE	GRANTABLE
USER_A	PATIENT	PAT_WING	USER_A	UPDATE	NO
USER_A	PATIENT	PAT_ROOM#	USER_A	UPDATE	NO
USER_A	PATIENT	PAT_BED	USER_A	UPDATE	NO

Example 6. Here USER_A is granting the UPDATE privilege on all columns of the PATIENT table to USER_F. Observe how information about this grant is recorded in the System Catalog table that contains a record of table privileges granted (i.e., made). Likewise, information about this grant is recorded when USER_F looks at the System Catalog table to view the table privileges received.

```
GRANT UPDATE
ON PATIENT
TO USER_F,;

Grant succeeded.
```

USER_A's table privileges granted as recorded in the System Catalog (note addition of USER_F's UPDATE privilege)

GRANTEE	OWNER	TABLE_NAME	GRANTOR	PRIVILEGE	GRANTABLE
PUBLIC	USER_A	MEDICATION	USER_A	DELETE	NO
PUBLIC	USER_A	MEDICATION	USER_A	INSERT	NO
PUBLIC	USER_A	MEDICATION	USER_A	SELECT	NO
PUBLIC	USER_A	MEDICATION	USER_A	UPDATE	NO
PUBLIC	USER_A	MEDICATION	USER_A	REFERENCES	NO
USER_B	USER_A	PATIENT	USER_A	SELECT	YES
USER_B	USER_A	ORDERS	USER_A	DELETE	NO
USER_B	USER_A	ORDERS	USER_A	INSERT	NO
USER_B	USER_A	ORDERS	USER_A	SELECT	NO
USER_B	USER_A	ORDERS	USER_A	UPDATE	NO
USER_C	USER_A	PATIENT	USER_A	INSERT	NO
USER_C	USER_A	PATIENT	USER_A	SELECT	NO
USER_D	USER_A	PATIENT	USER_B	SELECT	NO
USER_D	USER_A	MEDICATION	USER_A	DELETE	NO
USER_D	USER_A	MEDICATION	USER_A	INSERT	NO
USER_F	**USER_A**	**PATIENT**	**USER_A**	**UPDATE**	**NO**

USER_F's table privileges received as recorded in the System Catalog

OWNER	TABLE_NAME	GRANTOR	PRIVILEGE	GRANTABLE
USER_A	PATIENT	USER_A	UPDATE	NO

Figure 10.5 .summarizes all privileges granted at this point by USER_A and USER_B to other users as well as to the PUBLIC.

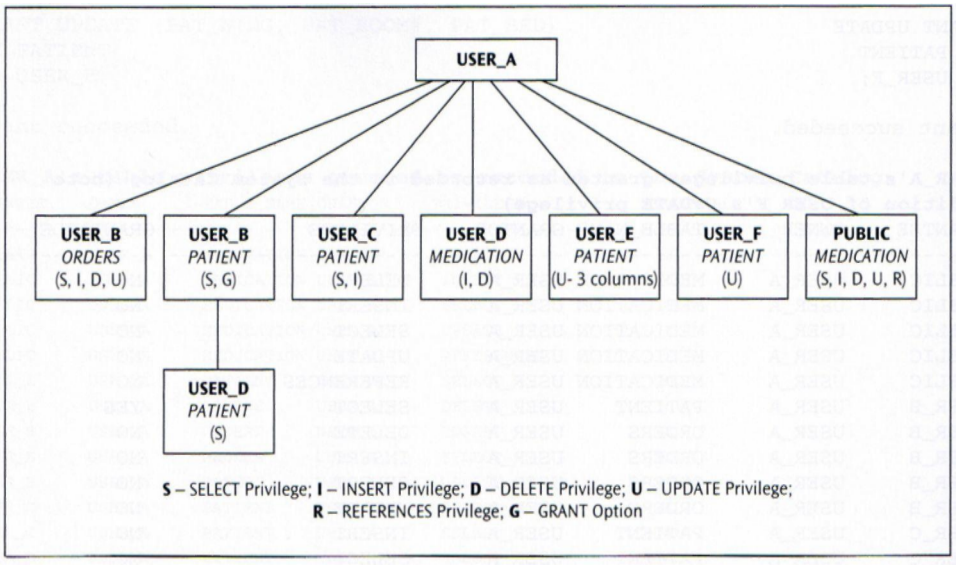


Figure 10.5 Privileges granted by User_A and User_B

Example 7. As shown below, the System Catalog records the fact that USER_B along with the entire public has been granted all privileges by USER_A on the MEDICATION table. Here USER_B is creating the table MY_MEDICATION which references the MED_CODE column in USER_A's MEDICATION table in a referential integrity constraint. Note that it is important that the references constraint in the MY_MEDICATION table specify that the MEDICATION table is owned by USER_A.

```
USER_B's table privileges received as a result of being a member of the PUBLIC
as recorded in the System Catalog
GRANTEE    OWNER      TABLE_NAME GRANTOR    PRIVILEGE              GRANTABLE
-------------------------------------------------------------------------------
PUBLIC     USER_A     MEDICATION USER_A     DELETE                 NO
PUBLIC     USER_A     MEDICATION USER_A     INSERT                 NO
PUBLIC     USER_A     MEDICATION USER_A     SELECT                 NO
PUBLIC     USER_A     MEDICATION USER_A     UPDATE                 NO
PUBLIC     USER_A     MEDICATION USER_A     REFERENCES             NO
```

```
CREATE TABLE MY_MEDICATION
 (My_name char(20),

My_med_C0de CHAR(5)  REFERENCES  USER_A.MEDICATION(MED_CODE));

Table created.
```

Observe what happens when USER_B attempts to insert a row into the MY_MEDICATION table that honors and fails to honor the referential integrity constraint.

```
INSERT INTO MY_MEDICATION VALUES ('Thomas Jones', 'TAG'

1 row created.

INSERT INTO MY_MEDICATION VALUES ('Sally Fields', 'XXX');
integrity constraint (USER_B.SYS_C004527) violated - parent key not found
```

Example 8. Observe that the REVOKE statement issued here by USER_A revoking all privileges on the PATIENT table from USER_C is successful because no dependent privileges have been based on this grant.

```
REVOKE ALL PRIVILEGES
ON PATIENT
FROM USER_C;

Revoke succeeded.
```


USER_A's table privileges granted as recorded in the System Catalog (note the removal of USER_C's two privileges)

GRANTEE	TABLE_NAME	GRANTOR	PRIVILEGE	GRANTABLE
PUBLIC	MEDICATION	USER_A	DELETE	NO
PUBLIC	MEDICATION	USER_A	INSERT	NO
PUBLIC	MEDICATION	USER_A	SELECT	NO
PUBLIC	MEDICATION	USER_A	UPDATE	NO
PUBLIC	MEDICATION	USER_A	REFERENCES	NO
USER_D	MEDICATION	USER_A	DELETE	NO
USER_D	MEDICATION	USER_A	INSERT	NO
USER_B	ORDERS	USER_A	DELETE	NO
USER_B	ORDERS	USER_A	INSERT	NO
USER_B	ORDERS	USER_A	SELECT	NO
USER_B	ORDERS	USER_A	UPDATE	NO
USER_B	PATIENT	USER_A	SELECT	YES
USER_F	PATIENT	USER_A	UPDATE	NO
USER_D	PATIENT	USER_B	SELECT	NO

Example 9. USER_A now wishes to revoke all privileges on the MEDICATION table previously given to the PUBLIC. Since USER_B has used the REFERENCES privilege received as a member of the PUBLIC in the creation of the MY_MEDICATION table, the default RESTRICT option causes the REVOKE statement issued by USER_A to fail.

```
REVOKE ALL PRIVILEGES
ON MEDICATION
FROM PUBLIC-

ERROR: CASCADE option must be specified to perform this revoke

REVOKE ALL PRIVILEGES
ON MEDICATION
FROM PUBLIC
CASCADE;

Revoke succeeded.
```

USER A's table privileges granted as recorded in the System Catalog

GRANTEE	TABLE_NAME	GRANTOR	PRIVILEGE	GRANTABLE
USER_D	MEDICATION	USER_A	DELETE	NO
USER_D	MEDICATION	USER_A	INSERT	NO
USER_B	ORDERS	USER_A	DELETE	NO
USER_B	ORDERS	USER_A	INSERT	NO
USER_B	ORDERS	USER_A	SELECT	NO
USER_B	ORDERS	USER_A	UPDATE	NO
USER_B	PATIENT	USER_A	SELECT	YES
USER_F	PATIENT	USER_A	UPDATE	NO
USER_D	PATIENT	USER_B	SELECT	NO

Observe that none of the PUBLIC'S privileges on the MEDICATION table exists, including the REFERENCES constraint. Thus, as shown below, it is possible for USER_B to insert a row into the MYJVIEDICATION table that refers to a non-existent My_med_code (i.e., XXX).


```
SELECT * FROM MY_MEDICATION;

My_name              My_med_code

Thomas Jones         TAG

INSERT INTO MY_MEDICATION VALUES ('Chet Gladchuk', 'TAG');

1 row created.

INSERT INTO MY_MEDICATION VALUES ('Sally Fields', 'XXX');

1 row created.
```

Example 10. At this point USER_L) is still able to make use of its SELECT privilege to query the PATIENT table owned by USER_A (see the query given immediately below). Observe what happens when USER_A revokes the SELECT privilege on the PATIENT table from USER_B who had subsequently granted this privilege to USER_D. USER_D is now unable to query the PATIENT table owned by USER_A.

```
SELECT * FROM USER_A.PATIENT;
```

Pat_p#a	Pat_p#n	Pat_name	Pat_gender	Pat_age	Pat_admit_dt	Pat_wing	Pat_room#	Pat_bed
DB	77642	Davis, Bill	M	27	2007-07-07	B	108	B

USER_D's table privileges received as recorded in the System Catalog

OWNER	TABLE_NAME	GRANTOR	PRIVILEGE	GRANTABLE
USER_A	PATIENT	USER_B	SELECT	NO
USER_A	MEDICATION	USER_A	DELETE	NO
USER_A	MEDICATION	USER_A	INSERT	NO

USER_A revokes all privileges on the PATIENT table from USER_B.

```
REVOKE ALL PRIVILEGES
ON PATIENT
FROM USERJB;

Revoke succeeded.
```

USER_A's table privileges granted as recorded in the System Catalog (note USER_B's privileges on the PATIENT table no longer exist; although not recorded in this portion of the System Catalog, the SELECT privilege on the PATIENT table granted by USER_B to USER_D no longer exists as well).

GRANTEE	TABLE_NAME	GRANTOR	PRIVILEGE	GRANTABLE
USER_D	MEDICATION	USER_A	DELETE	NO
USER_D	MEDICATION	USER_A	INSERT	NO
USER_B	ORDERS	USER_A	DELETE	NO
USER_B	ORDERS	USER_A	INSERT	NO
USER_B	ORDERS	USER_A	SELECT	NO
USER_B	ORDERS	USER_A	UPDATE	NO
USER_F	PATIENT	USER_A	UPDATE	NO

USER_B's table privileges received as recorded in the System Catalog (compare with comparable System Catalog information in Example 4)

OWNER	TABLE_NAME	GRANTOR	PRIVILEGE	GRANTABLE
USER_A	ORDERS	USER_A	DELETE	NO
USER_A	ORDERS	USER_A	INSERT	NO
USER_A	ORDERS	USER_A	SELECT	NO
USER_A	ORDERS	USER_A	UPDATE	NO

USER_D's table privileges received as recorded in the System Catalog (compare with comparable System Catalog immediately prior to REVOKE statement immediately above)

OWNER	TABLE_NAME	GRANTOR	PRIVILEGE	GRANTABLE
USER_A	MEDICATION	USER_A	DELETE	NO
USER_A	MEDICATION	USER_A	INSERT	NO

USER_D is no longer able to query the PATIENT table owned by USER__A.

```
SELECT * FROM USER_A.PATIENT;

ERROR: table or view does not exist
```

CHAPTER 11

Data Manipulation: Relational Algebra and SQL

From the SQL/DDL for database creation in the previous chapter, we now move on to learn about data retrieval. Relational algebra, a mathematical expression of data retrieval methods prescribed by E. F. Codd, is introduced first as a means to specify the logic for data retrieval from a relational database. A query expressed in relational algebra involves a sequence of operations which, when executed in the order specified, produces the desired results. SQL is the most common way that relational algebra is implemented for data retrieval operations in a relational database.

Chapter 11 is divided into two major sections. Section 11.1 discusses relational algebra with examples. Section 11.2 presents the syntax for SQL and enumerates various ways in which SQL can be used for data manipulation, with copious examples. Included as part of the discussion are illustrations of how SQL treats missing data (or what are called null values), and a discussion of different types of subqueries.

11.1 Relational Algebra[1]

The relational data model includes a group of basic data manipulation operations. As a result of its theoretical foundation in set theory, the relational data model's operations include Union, Intersection, and Difference. Five other relational operators also exist: Select, Project, Cartesian Product, Join, and Divide. Collectively these eight operators comprise relational algebra. This section discusses each of these eight operators and gives examples of their use in the formulation of queries. The examples are based on the Madeira College registration system introduced in Chapter 10. The Fine-granular Design-Specific ER diagram for Madeira College appears in Figure 10.2, and its information-reducing and information-preserving logical schema are shown in Figures 10.3 and 10.4, respectively. Figure 11.1 contains representative data for the DEPARTMENT, PROFESSOR, COURSE, and SECTION relations used in the relational algebra examples in Section 11.1 along with representative data for other Madeira College relations used in conjunction with the SQL examples that begin in Section 11.2.

The discussion of these relational algebra operators begins with the two that operate on a single relation (unary operators) followed by those that operate on two relations (binary operators). Figure 11.2 shows the fundamental relational algebra operators, along with their symbolic representation. In Figure 11.2, the fundamental operators are indicated by a double asterisk (**). The rest can be defined from the fundamental operators. Nonetheless, these operators are usually included in relational algebra as a matter of convenience.

[1] The discussion of and the notation used for various relational algebra operations in this section is based on R. Elmasri and S.B. Navathe (2003), *Fundamental of Database Systems,* Addison-Wesley. The reader is encouraged to refer to Elmasri and Navathe (2003) and G. J. Date (1995), *An Introduction to Database Systems,* Addison-Wesley for a more in-depth discussion of relational algebra.

DEPARTMENT Relation

Dpt_name	Dpt_dcode	Dpt_college	Dpt_phone	Dpt_budget	Dpt_location	Dpt_hodid
Economics	1	Arts and Sciences	5235567654	433545	123 McMicken	FM49276
QA/QM	3	Business	5235566656	134556	333 Lindner	BA54325
Economics	4	Education	5235569978	400000	336 Dyer	CM65436
Mathematics	6	Engineering	5235564379	433567	728 Old Chem	RR79345
IS	7	Business	5235567489	400000	333 Lindner	CC49234
Philosophy	9	Arts and Sciences	5235565546	333333	272 McMicken	CM87659

COURSE Relation

Co_name	Co_course#	Co_credit	Co_college	Co_hrs	Co_dpt_dcode
Intro to Economics	15ECON112	U	Arts and Sciences	3	1
Operations Research	22QA375	U	Business	2	3
Intro to Economics	18ECON123	U	Education	4	4
Supply Chain Analysis	22QA411	U	Business	3	3
Principles of IS	22IS270	G	Business	3	7
Programming in C++	20ECES212	G	Engineering	3	6
Optimization	22QA888	G	Business	3	3
Financial Accounting	18ACCT801	G	Education	3	4
Database Concepts	22IS330	U	Business	4	7
Database Principles	22IS832	G	Business	3	7
Systems Analysis	22IS430	G	Business	3	7

STUDENT Relation

St_sid	St_name	St_address	St_status	St_birthdate
BE76598	Elijah Baley	2920 Scioto Street	Part time	
OD76578	Daniel Olive	338 Bishop Street	Full time	1982-05-12
SW56547	Wanda Seldon	3138 Probasco	Full time	1970-03-03
BG66765	Gladis Bale	356 Vine Street	Full time	1977-10-23
GS76775	Shweta Gupta	356 Probasco	Full time	1979-05-21
HT67657	Troy Hudson		Part time	
FR45545	Rick Fox	314 Clifton	Full time	1983-10-09
FV67733	Vanessa Fox	314 Clifton	Full time	1983-10-20
HJ45633	Jenna Hopp	2930 Scioto Street	Full time	1970-03-03
SD23556	David Sane	245 University Avenue	Part time	1984-07-14
DT87656	Tim Duncan		Part time	1975-05-21
KJ56656	Joumana Kidd	2920 Scioto Street	Part time	
AJ76998	Jenny Aniston	88 MLK	Full time	
KP78924	Poppy Kramer	437 Love Lane	Full time	1980-11-11
KS39874	Sweety Kramer	748 Hope Avenue	Full time	1980-11-11
JD35477	Diana Jackson	2920 Scioto Street	Part time	1976-02-20

GRAD_STUDENT Relation

Gs_st_sid	Gs_thesis	Gs_ugmajor
BE76598	Y	Marketing
SW56547	Y	Finance
BG66765	N	Archeology
GS76775	N	Archeology
HJ45633	Y	History
DT87656	N	Physics
KJ56656	Y	History
AJ76998	Y	Child Care
JD35477	N	Mathematics

SECTION Relation

Se_section#	Se_qtr	Se_year	Se_time	Se_maxst	Se_room	Se_co_course#	Se_pr_profid
101	A	2007	T1015	25		22QA375	HT54347
901	A	2006	W1800	35	Rhodes 611	22IS270	SK85977
902	A	2006	H1700	25	Lindner 108	22IS270	SK85977
101	S	2006	T1045	29	Lindner 110	22IS330	SK85977
102	S	2006	H1045	29	Lindner 110	22IS330	CC49234
701	W	2007	M1000	33	Braunstien 211	22IS832	CC49234
101	A	2007	W1800		Baldwin 437	20ECES212	RR79345
101	U	2007	T1015	33		22QA375	HT54347
101	A	2007	H1700	29	Lindner 108	22IS330	SK85977
101	S	2007	T1015	30		22QA375	HT54347
101	W	2007	T1015	20		22QA375	HT54347

Figure 111 Madeira College relations

TAKES Relation

Tk_se_section#	Tk_se_qtr	Tk_se_year	Tk_se_co_course#	Tk_grade	Tk_st_sid
101	A	2007	22QA375	A	KP78924
101	A	2007	22QA375	A	KS39874
101	A	2007	22QA375	B	BG66765
101	S	2006	22IS330	C	BE76598
101	A	2007	22IS330	B	KJ56656
101	A	2007	22IS330	A	KP78924
101	A	2007	22IS330	A	KS39874
701	W	2007	22IS832	A	KS39874
101	A	2007	22IS330	A	BE76598
701	W	2007	22IS832	B	BG66765
101	A	2007	22IS330	C	GS76775

PROFESSOR Relation

Pr_name	Pr_empid	Pr_phone	Pr_office	Pr_birthdate	Pr_datehired	Pr_dpt_dcode	Pr_salary
John Smith	SJ89324	5235567645	223 McMicken	1966-10-12	2001-06-23	1	45000
Mike Faraday	FM49276	5235568492	249 McMicken	1960-08-26	1996-05-01	1	92000
Kobe Bryant	BK68765	5235568522	322 McMicken	1968-03-02	1998-05-01	1	66000
Ram Raj	RR79345	5235567244	822 Old Chem	1970-02-06	2001-06-23	6	44000
John B Smith	SJ65436	5235567556	838 Old Chem			6	
Prester John	JP77869	5235567244	822 Old Chem	1955-08-25	1995-08-25	6	44000
Chelsea Bush	BC65437	5235567777	227 Lindner	1946-09-03	1993-05-01	3	77000
Tony Hopkins	HT54347	5235569977	324 Lindner	1949-11-24	1997-01-20	3	77000
Alan Brodie	BA54325	5235569876	238 Lindner	1944-01-14	2000-05-16	3	76000
Jessica Simpson	SJ67543	5235565567	324 Lindner	1955-08-25	1995-08-25	3	67000
Laura Jackson	JL65436	5235565436	336 Lindner	1973-10-16	2000-09-23	3	43000
Marie Curie	CM65436	5235569899	331 Dyer	1972-02-29	1999-10-22	4	99000
Jack Nicklaus	NJ33533	5235566767		1976-01-01	1999-12-31	4	67000
John Nicholson	NJ43728	5235569999	324 Dyer	1966-05-01	2003-06-22	4	99000
Sunil Shetty	SS43278	5235566764	526 Lindner		1993-06-28	7	64000
Katie Shef	SK85977	5235568765	572 Lindner	1948-08-08	1997-06-06	7	65000
Cathy Cobal	CC49234	5235565345	544 Lindner	1968-02-28	2001-01-23	7	45000
Jeanine Troy	TJ76546	5235565545	423 McMicken	1968-01-16		9	45000
Tiger Woods	WT65487	5235565563		1975-11-14	2003-11-14	9	
Mike Crick	CM87659	5235565569	444 McMicken	1970-05-31	2002-05-30	9	69000

TEXTBOOK Relation

Tx_isbn	Tx_title	Tx_year	Tx_publisher
000-66574998	Database Management	1999	Thomson
003-6679233	Linear Programming	1997	Prentice-Hall
001-55-435	Simulation Modeling	2001	Springer
118-99898-67	Systems Analysis	2000	Thomson
77898-8769	Principles of IS	2002	Prentice-Hall
0296748-99	Economics For Managers	2001	
0296437-1118	Programming in C++	2002	Thomson
012-54765-32	Fundamentals of SQL	2004	
111-11111111	Data Modeling	2006	

USES Relation

Us_co_course#	Us_tx_isbn	Us_pr_empid
22IS832	000-66574998	SS43278
22IS270	000-66574998	SS43278
22IS270	77898-8769	SS43278
22IS270	77898-8769	SK85977
22IS270	77898-8769	CC49234
20ECES212	0296437-1118	CC49234
22QA375	0296437-1118	SJ65436
22IS330	003-6679233	BC65437
18ECON123	0296748-99	CM65436
22IS330	118-99898-67	SS43278
22IS832	118-99898-67	SK85977
22QA888	001-55-435	HT54347

Figure 111 Madeira College relations (continued)

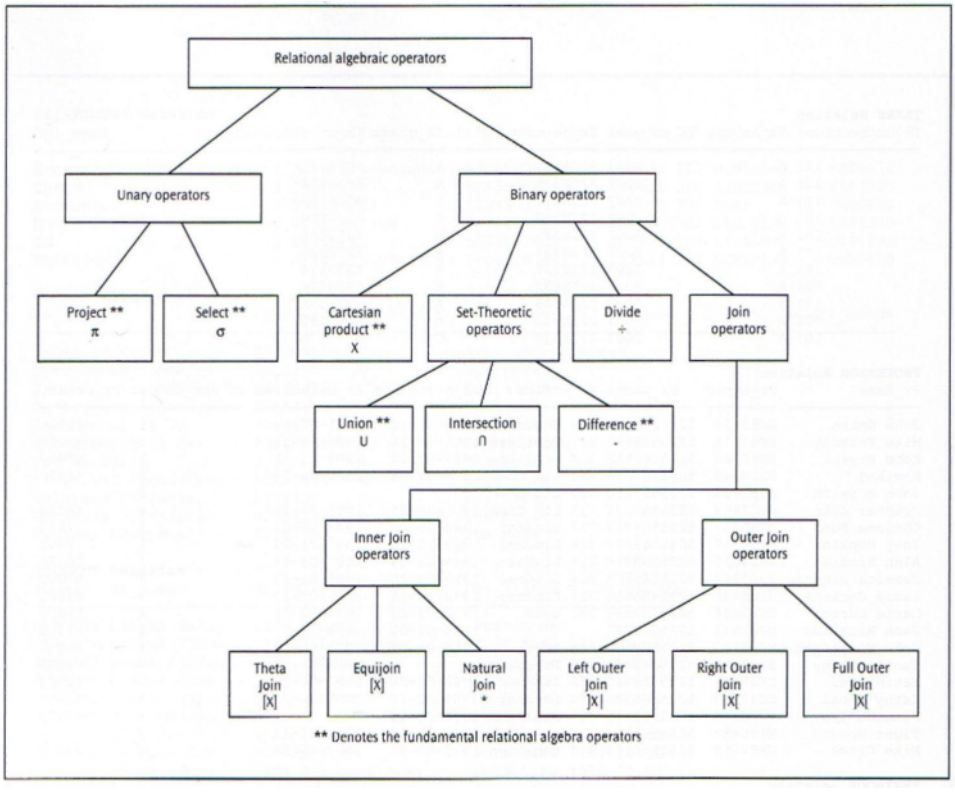


Figure 11.2 Classification of relational algebra operators

11.11 Unary Operators

Unary operators "operate" on a single relation. There are two relational algebra unary operators: the Select operator and the Project operator.

11.11.1 The Select Operator

The Select operator is used to select a horizontal subset of the tuples that satisfy a selection condition from a relation. The general form of the Selection operation is:

$$\sigma_{\text{<selection condition>}}(R)$$

where:

- **the symbol σ (sigma) designates the Select operator, and**
- **the selection condition is a Boolean expression specified on the attributes of relation schema R.**

R is generally a *relational algebra expression* whose result is a relation; the simplest expression is the name of a single relation. The relation resulting from the Selection

operation has the same attributes as R. The Boolean expression specified as <selection condition> is composed of a number of clauses of the form:

<attribute' namexcomparison operatorxconstant value>

or:

<attribute namexcomparison operatorxattribute name>

where:

- <attribute name> is the name of an attribute of R,
- <comparison operator> is normally one of the operators {=, *jt* , <, < , >, >}, and
- <constant value> is a constant value from the domain of the attribute.

Following are three examples of Selection operations performed on the COURSE relation shown in Figure 11.1.

Selection Example 1. Which courses are three-hour courses?

Relational Algebra Syntax:

°<Co_hrs = 3) (COURSE)

Observe that the result is an unnamed relation that contains a subset of the tuples in the COURSE relation.

Result:

Co_name	Co_course#	Co_credit	Co_college	Co_hrs	Co_dpt_dcode
Supply Chain Analysis	22QA411	U	Business	3	3
Principles of IS	22IS270	G	Business	3	7
Programming in C++	20ECES212	G	Engineering	3	6
Optimization	22QA888	G	Business	3	3
Financial Accounting	18ACCT801	G	Education	3	4
Database Principles	22IS832	G	Business	3	7
Systems Analysis	22IS430	G	Business	3	7

The Boolean operators AND, OR, and NOT can be used to form a general selection condition.

Selection Example 2. Which courses offered by department 7 are three-hour courses?

Relational Algebra Syntax:

$\sigma_{(Co\_dpt\_dcode = 7 \text{ and } Co\_hrs = 3)}(COURSE)$

Use of the logical operator AND requires that *both* conditions (**Co_dpt_dcode** = 7 and **Co_hrs** = 3) be satisfied.

Result:

Co_name	Co_course#	Co_credit	Co_college	Co_hrs	Co_dpt_dcode
Principles of IS	22IS270	G	Business	3	7
Database Principles	22IS832	G	Business	3	7
Systems Analysis	22IS430	G	Business	3	7

Selection Example 3. Which courses are offered in either the College of Arts and Sciences or the College of Education?

Relational Algebra Syntax:

$$\sigma_{(Co\_college = \text{'Arts and Sciences'} \text{ or } Co\_college = \text{'Education'})}(COURSE)$$

Use of the logical operator OR allows either condition (Co_college = 'Arts and Sciences' or Co_college = 'Education') to be satisfied.

Result:

Co_name	Co_course#	Co_credit	Co_college	Co_hrs	Co_dpt_dcode
Intro to Economics	15ECON112	U	Arts and Sciences	3	1
Intro to Economics	18ECON123	U	Education	4	4
Financial Accounting	18ACCT801	G	Education	3	4

11.1.1.2 The Project Operator

While the Select operator selects some of the *tuples* from the relation while eliminating unwanted tuples, the Project operator selects certain *attributes* from the relation and eliminates unwanted attributes. In other words, a Selection operation forms a new relation by taking a *horizontal* subset of an existing relation, whereas a Projection operation forms a new relation by taking a *vertical* subset of an existing relation. The difference between Selection and Projection is shown pictorially in Figure 11.3.

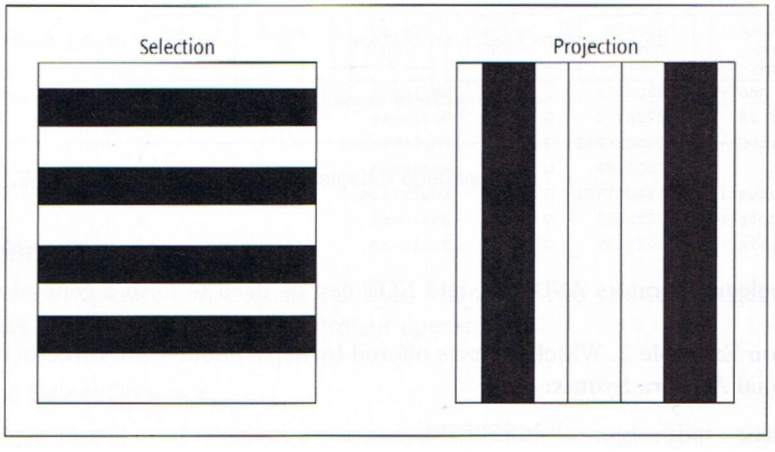


Figure 11.3 Selection compared to projection

The general form of the Projection operation is:

$$\pi_{\text{<attribute list>}}(R)$$

where:

- **the symbol π (pi) is used to represent the Project operator, and**
- **<attribute list> is a subset of the attributes of the relation schema R.**

As was the case in the Selection operation, R, in general, is a *relational algebra expression* whose result is a relation, which in the simplest case is just the name of a single relation. The result of the Projection operation contains only the attributes specified in the <attribute list> in the same order as they appear in the list.

In cases where the attribute list produced as a result of the Projection operation is not a superkey of R, duplicate tuples are likely to occur in the result. Since relations are sets and do not allow duplicate tuples, only one copy of each group of identical tuples is included in the result of a Projection. Following are a couple of examples of Projection operations based on the COLLEGE and DEPARTMENT relations in Figure 11.1.

Projection Example 1. Which colleges offer courses?

Relational .\lgebra Syntax:

$$\pi_{(Co\_college)}\ (COURSE)$$

Since **Co_college** is not a superkey of COURSE, the number of tuples in the result is less than the number of tuples in COURSE.

Result:

```
Co_college
--------------------
Arts and Sciences
Business
Education
Engineering
```


Projection Example 2. What is the name and college of each department?

Relational Algebra Syntax:

$$\pi_{(Dpt\_name,\ Dpt\_college)}\ (DEPARTMENT)$$

Since **[Dpt_name, Dpt_college]** is a superkey of DEPARTMENT, the number of tuples resulting from a Projection operation on DEPARTMENT is equal to the number of tuples in DEPARTMENT.

Result:

```
Dpt_name        Dpt_college
--------------- ----------------
Economics       Arts and Sciences
QA/QM           Business
Economics       Education
Mathematics     Engineering
IS              Business
Philosophy      Arts and Sciences
```

11.1.2 Binary Operators

The relational algebra binary operators "operate" on two relations. There are four binary operators:

- The Cartesian Product operator
- Set theoretic operators
- Join operators
- The Divide operator

11.1.2.1 The Cartesian Product Operator

The Cartesian Product operator (often referred to as the Product or Cross-Product operation), denoted by X, is used to combine tuples from any two relations in a combinatorial fashion. The Cartesian Product of relations R and S is created by (a) concatenating the

attributes of R and S together, and (b) attaching to each tuple in R each of the tuples in S. Thus if R has n_R tuples and S has n_S tuples, then the Cartesian Product Q will have n_R times n_S tuples and take the form shown in Figure 11.4.

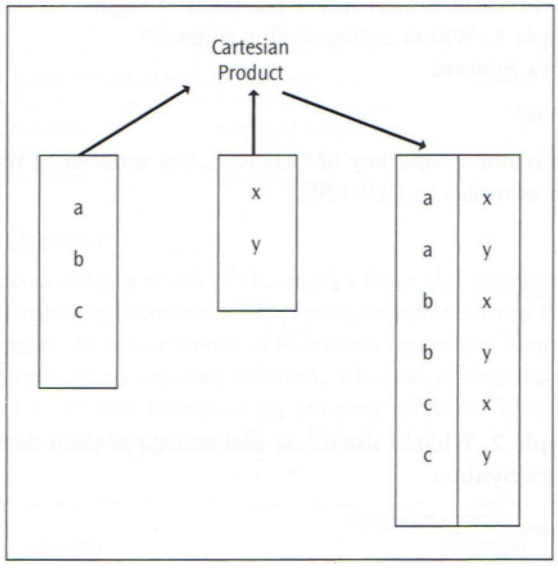


Figure 11.4 Cartesian Product operation

Cartesian Product Example 1. What is the product of the DEPARTMENT and COURSE relations?

Relational Algebra Syntax:

COURSE X DEPARTMENT

Since there are six tuples in DEPARTMENT and eleven tuples in COURSE, a total of 66 tuples that contain 13 attributes per tuple is produced when the Cartesian Product of DEPARTMENT and COURSE is formed.

In order to illustrate the result obtained by a Cartesian Product operation, consider two relations—D and C derived from the DEPARTMENT and COURSE relations. Relation D contains six tuples with the attributes D_name, D_dcode, and D^college; relation C contains eleven tuples with the attributes C_name, C_course#, C_credit. and C_d_dcode. The content of relations C and D is shown in Figure 11.5.

C Relation

C_name	C_course#	C_credit	C_d_dcode
Intro to Economics	15ECON112	U	1
Operations Research	22QA375	U	3
Intro to Economics	18ECON123	U	4
Supply Chain Analysis	22QA411	U	3
Principles of IS	22IS270	G	7
Programming in C++	20ECES212	G	6
Optimization	22QA888	G	3
Financial Accounting	18ACCT801	G	4
Database Concepts	22IS330	U	7
Database Principles	22IS832	G	7
Systems Analysis	22IS430	G	7

D Relation

D_name	D_dcode	D_college
Economics	1	Arts and Sciences
QA/QM	3	Business
Economics	4	Education
Mathematics	6	Engineering
IS	7	Business
Philosophy	9	Arts and Sciences

Figure 11.5 The C and D relations

Relational Algebra Syntax:

C **X** D

The first 18 of the 66 tuples produced by the Cartesian Product of relations C and D follow.

Result (First 18 tuples of the Cartesian Product):

D_name	D_dcode	D_college	C_name	C_course#	C_credit	C_d_dcode
Economics	1	Arts and Sciences	Intro to Economics	15ECON112	U	1
QA/QM	3	Business	Intro to Economics	15ECON112	U	1
Economics	4	Education	Intro to Economics	15ECON112	U	1
Mathematics	6	Engineering	Intro to Economics	15ECON112	U	1
IS	7	Business	Intro to Economics	15ECON112	U	1
Philosophy	9	Arts and Sciences	Intro to Economics	15ECON112	U	1
Economics	1	Arts and Sciences	Operations Research	22QA375	U	3
QA/QM	3	Business	Operations Research	22QA375	U	3
Economics	4	Education	Operations Research	22QA375	U	3
Mathematics	6	Engineering	Operations Research	22QA375	U	3
IS	7	Business	Operations Research	22QA375	U	3
Philosophy	9	Arts and Sciences	Operations Research	22QA375	U	3
Economics	1	Arts and Sciences	Intro to Economics	18ECON123	U	4
QA/QM	3	Business	Intro to Economics	18ECON123	U	4
Economics	4	Education	Intro to Economics	18ECON123	U	4
Mathematics	6	Engineering	Intro to Economics	18ECON123	U	4
IS	7	Business	Intro to Economics	18ECON123	U	4
Philosophy	9	Arts and Sciences	Intro to Economics	18ECON123	U	4

Note that the result shown above is the concatenation of the first three tuples of relation C (see columns 4-7) with all six tuples of relation D (see columns 1-3, rows 1-6, 7-12, and 13-18).

The Cartesian Product operation by itself is generally of little value. It is useful when followed first by a Selection operation that matches values of attributes coming from the component relations (technically, a Cartesian Product operation followed by a Selection operation is equivalent to a Join operation) and sometimes by a Projection operation that selects certain columns from the selected set of tuples.

Cartesian Product Example 2. What are the names of the departments and associated colleges that offer a four-hour course?

Relational Algebra Syntax:

$$\pi_{(\text{Dpt\_name, Dpt\_college})}(\sigma_{(\text{Co\_hrs = 4 and Co\_dpt\_dcode = Dpt\_dcode})}(\text{COURSE} \times \text{DEPARTMENT}))$$

As we will see in Section 11.1.2.3, $\sigma_{(\text{Co\_hrs = 4 and Co\_dpt\_dcode = Dpt\_dcode})}$ (COURSE X DEPARTMENT) represents a JOIN operation with COURSE X DEPARTMENT serving as its fundamental building block.

Result:

Dpt_name	Dpt_college
Economics	Education
IS	Business

11.1.2.2 Set Theoretic Operators

Three set theoretic operators—Union, Intersection, and Difference—are used to combine the tuples from two relations. These are binary operations as each is applied to two sets. When adapted to relational databases, the two relations on which any of the above three operations are applied must be union compatible. Two relations $R(A_1, A_2, A_n)$ and $S(B_1, B_2, B_n)$ are said to be union compatible if (a) they have the same degree (i.e., have the same number of attributes), and (b) each pair of corresponding attributes in R and S share the same domain. Venn diagrams illustrating Union, Intersection, and Difference are shown in Figure 11.6.

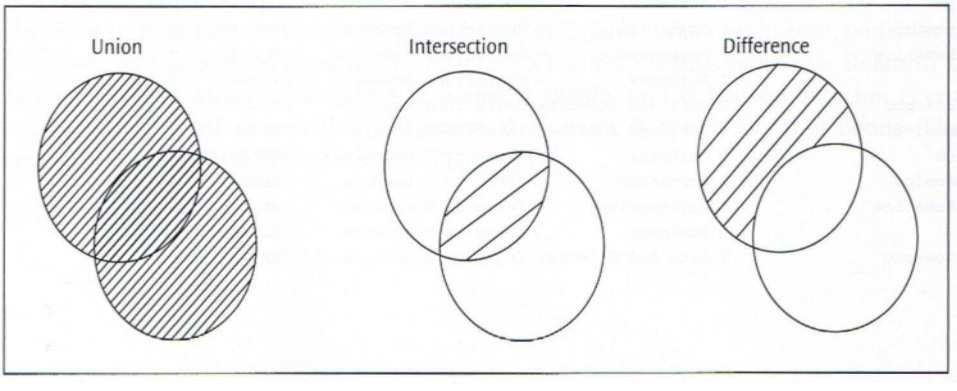


Figure 11.6 The Union, Intersection, and Difference operations

The remainder of this section consists of the definition and examples of the three set theoretic operators.

Union: The result of this operation, denoted by R u S, is a relation that includes all tuples that belong to either R or S or to both R and S. Duplicate tuples are eliminated.

Union Example. Let relations R and S be derived from the SECTION relation. R contains tuples indicating Fall quarter sections (Se_qtr = 'A') and S contains tuples listing sections offered in a Lindner classroom (Se_room = 'Lindner').

RELATION R

Se_section#	Se_qtr	Se_year	Se_time	Se_maxst	Se_room	Se_co_course#	Se_pr_profid
902	A	2006	H1700	25	Lindner 108	22IS270	SK85977
901	A	2006	W1800	35	Rhodes 611	22IS270	SK85977
101	A	2007	H1700	29	Lindner 108	22IS330	SK85977
101	A	2007	T1015	25		22QA375	HT54347
101	A	2007	W1800		Baldwin 437	20ECES212	RR79345

RELATION S

Se_section#	Se_qtr	Se_year	Se_time	Se_maxst	Se_room	Se_co_course#	Se_pr_profid
902	A	2006	H1700	25	Lindner 108	22IS270	SK85977
101	S	2006	T1045	29	Lindner 110	22IS330	SK85977
102	S	2006	H1045	29	Lindner 110	22IS330	CC49234
101	A	2007	H1700	29	Lindner 108	22IS330	SK85977

Relational Algebra Syntax and Result: R∪S

Se_section#	Se_qtr	Se_year	Se_time	Se_maxst	Se_room	Se_co_course#	Se_pr_profid
101	A	2007	H1700	29	Lindner 108	22IS330	SK85977
101	A	2007	T1015	25		22QA375	HT54347
101	A	2007	W1800		Baldwin 437	20ECES212	RR79345
101	S	2006	T1045	29	Lindner 110	22IS330	SK85977
102	S	2006	H1045	29	Lindner 110	22IS330	CC49234
901	A	2006	W1800	35	Rhodes 611	22IS270	SK85977
902	A	2006	H1700	25	Lindner 108	22IS270	SK85977

The Union R u S contains the sections that are offered either exclusively in a fall quarter or exclusively in a Lindner classroom, or offered in a Lindner classroom during a fall quarter (see the first and seventh tuples).

Intersection: The result of this operation, denoted by R n S, is a relation that includes all tuples that are in both R and S.

Intersection Example. Using the data in the relations R and S given above, form the intersection of R and S.

Relational Algebra Syntax and Result: R n S

Se_section#	Se_qtr	Se_year	Se_time	Se_maxst	Se_room	Se_co_course#	Se_pr_profid
101	A	2007	H1700	29	Lindner 108	22IS330	SK85977
902	A	2006	H1700	25	Lindner 108	22IS270	SK85977

Observe that the sections shown above are the only sections offered in a Lindner classroom during a fall quarter.

Difference: The result of this operation, denoted by **R - S**, is a relation that includes all tuples that are in **R** but not in **S**.

Difference Example 1. Using the data in the relations **R** and **S** given above, form the difference **R** minus **S**.

Relational Algebra Syntax and Result: R - S

Se_section#	Se_qtr	Se_year	Se_time	Se_maxst	Se_room	Se_co_course#	Se_pr_profid
101	A	2007	T1015	25		22QA375	HT54347
101	A	2007	W1800		Baldwin 437	20ECES212	RR79345
901	A	2006	W1800	35	Rhodes 611	22IS270	SK85977

Note that the result obtained by subtracting **S** from **R** is equal to all sections offered during a fall quarter in a classroom other than Lindner. The classroom associated with the section in the first tuple is not available (i.e., is a null value) and satisfies the condition that the section be offered in a classroom other than Lindner.

Difference Example 2. Using the data in the relations **R** and **S** given above, form the difference **S** minus **R**.

Relational Algebra Syntax and Result: S - R

Se_section#	Se_qtr	Se_year	Se_time	Se_maxst	Se_room	Se_co_course#	Se_pr_profid
101	S	2006	T1045	29	Lindner 110	22IS330	SK85977
102	S	2006	H1045	29	Lindner 110	22IS330	CC49234

Observe that the result obtained by subtracting **R** from **S** is equal to all sections offered in a classroom located in Lindner during a quarter other than the fall quarter.

11.1.2.3 Join Operators

Joins come in several varieties. Basically in each, the Join operation, denoted by [X][2], is used to combine related tuples from two relations into single tuples. The example given previously in Cartesian Product Example 2 where the names of the departments and their associated colleges that offer a four-hour course is requested can be specified using the Join operation by replacing:

$$\pi_{(Dpt\_name,\ Dpt\_college)}(\sigma_{(Co\_hrs\ =\ 4\ and\ Co\_dpt\_dcode\ =\ Dpt\_dcode)}\ (COURSE\ \mathbf{x}\ DEPARTMENT))$$

with a Join operation followed by a Projection operation as follows:

$$\pi_{(Dpt\_name,\ Dpt\_college)}\ (COURSE\ [\mathbf{X}]_{Co\_hrs\ =\ 4\ and\ Co\_dpt\_dcode\ =\ Dpt\_dcode}\ DEPARTMENT)$$

The general form of a Join operation on two relations $R(A_1, A_2, ..., A_n)$ and $S(B_1, B_2, ..., B_m)$ is:

$$R\ [\mathbf{X}]_{<join\ condition>}\ S$$

The result of the Join operation is a relation Q with n + m attributes $Q(A_1, A_2, \ldots A_n, B_1, B_2, \ldots B_m)$, in that order. Q has one tuple for each combination of tuples—one from R and one from S—whenever the combination satisfies the join condition. This is the main difference between Cartesian Product and Join; in Join only combinations of tuples satisfying the join

[2] In this book, [X] is used to represent the Equijoin and Theta Join operations as well as all Outer Join operations while * is used to represent the Natural Join operation.

condition appear in the result, while in the Cartesian Product all combinations of tuples are included in the result. The join condition is specified on attributes from the two relations R and S and is evaluated for each combination of tuples. Each tuple combination for which the join condition evaluates to true is included in the resulting relation Q as a single combined tuple. In order to join the two relations R and S, they must be join compatible, that is, the join condition must involve attributes from R and S which share the same domain.

A general join condition is of the form:

```
<condition> AND <condition> AND ... AND <condition>
```

where each condition is of the form Aj 0 Bj, Aj is an attribute of R, Bj is an attribute of S, Aj and Bj have the same domain, and 0 (theta) is one of the comparison operators {=, <, , >,, # }• A Join operation with such a general join condition is called a Theta join. Tuples whose join attributes are null do not appear in the result.

The Equijoin Operator The most common join involves join conditions where the comparison operator is "=". This type of join is called the Equijoin. The result of an Equijoin includes all attributes from both relations participating in the Join operation. This implies duplication of the joining attributes in the result.

Equijoin Example. Join the C and D relations (derived from the COURSE and DEPARTMENT relations) over their common attribute department code (**D_dcode** in relation D and **C_d_dcode** in relation C).

Relational Algebra Syntax:

```
C  [X]  c_a_dcc.de  =  D_dcode  ^
```

Result:

C_name	C_course#	C_credit	C_d_dcode	D_name	D_dcode	D_college
Intro to Economics	15ECON112	U	1	Economics	1	Arts and Sciences
Operations Research	22QA375	U	3	QA/QM	3	Business
Optimization	22QA888	G	3	QA/QM	3	Business
Supply Chain Analysis	22QA411	U	3	QA/QM	3	Business
Intro to Economics	18ECON123	U	4	Economics	4	Education
Financial Accounting	18ACCT801	G	4	Economics	4	Education
Programming in C++	20ECES212	G	6	Mathematics	6	Engineering
Principles of IS	22IS270	G	7	IS	7	Business
Database Concepts	22IS330	U	7	IS	7	Business
Database Principles	22IS832	G	7	IS	7	Business
Systems Analysis	22IS430	G	7	IS	7	Business

Observe that the name (**D_name**) and college (**D_college**) of the department associated with each course appears in the result shown above.

Had the Equijoin involved the complete COURSE and DEPARTMENT relations joined on the **Co_dpt_dcode** and **Dpt_dcode** attributes instead of just relations C and D, the result would have also included 11 tuples with each tuple containing 13 attributes.

While it is possible to join COURSE and DEPARTMENT over the **Co_college** attribute from COURSE and the **Dpt_college** attribute from DEPARTMENT, the result of such a join would be meaningless—each tuple in COURSE would be concatenated not only with the tuple from DEPARTMENT that offers the course, but also with the tuples from DEPARTMENT with the same **Dpt college** but associated with a different department code than that associated with the course. Using the data in the COURSE and DEPARTMENT relations in

Figure 11.1, it is left as an exercise for the reader to demonstrate that an Equijoin of COURSE and DEPARTMENT on the attributes **Co_college** and **Dpt_college** yields a result that contains 19 tuples.

The Natural Join Operator **Because the result of an Equijoin results in pairs of attributes with identical values in all the tuples (see C_d_dcode and D_dcode in the result shown above), a new relational algebra operation called a Natural Join, denoted by *, was created to omit the second (and unnecessary) attribute in an Equijoin condition. The Natural Join of two relations with the common attribute b is illustrated in Figure 11.7.**

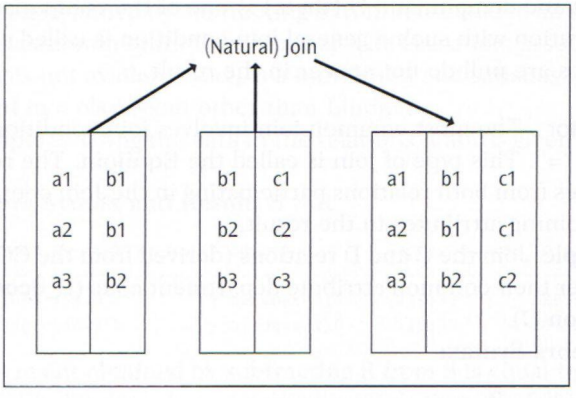


Figure 11.7 The Natural Join operation

Natural Join Example. Join the G and D relations over their common attribute department code (D_dcode in relation D and C_d_dcode in relation C).

Relational Algebra Syntax:[3]

$$C * \underset{C\_d\_dcode = D\_dcode}{} n$$

Result:

C_name	C_course#	C_credit	D_name	D_dcode	D_college
Intro to Economics	15ECON112	U	Economics	1	Arts and Sciences
Operations Research	22QA375	U	QA/QM	3	Business
Optimization	22QA888	G	QA/QM	3	Business
Supply Chain Analysis	22QA411	U	QA/QM	3	Business
Intro to Economics	18ECON123	U	Economics	4	Education
Financial Accounting	18ACCT801	G	Economics	4	Education
Programming in C++	20ECES212	G	Mathematics	6	Engineering
Principles of IS	22IS270	G	IS	7	Business
Database Concepts	22IS330	U	IS	7	Business
Database Principles	22IS832	G	IS	7	Business
Systems Analysis	22IS430	G	IS	7	Business

Contrary to the requirement that attributes have unique names over the entire relational schema, the standard definition of Natural Join requires that the two join attributes (or each pair of join attributes) have the same name. If this is not the case, a renaming operation must be applied first. See Elmasri and Navathe (2004) for a discussion of the use of the renaming operation in representing a Natural Join.

Observe that only one (D_dcode) of the two join attributes common to both relations (D_dcode and C_d_dcode) appears in the result shown above.

The Join operation is used to combine data from multiple relations so that related information can be presented in a single relation. As illustrated in Cartesian Product Example 2, Join operations are typically followed by a Projection operation.

The Theta Join Operator While occurring infrequently in practical applications, Theta Joins that do not involve equality conditions are possible as long as the join condition involves attributes that share the same domain.

Theta Join Example. Instead of doing an Equijoin of C and D when an equality condition involving their two common attributes (C_d_dcode and D_dcode) exists, do a join of C and D when an inequality condition exists for these common attributes.

Relational Algebra Syntax:

```
C [X] C_d_dcode <> D_dcode D
```

In such a Theta Join, a tuple from D is concatenated with a tuple from C only when an inequality exists for each of the join conditions. Thus the first tuple in D is not concatenated with the first tuple in C because the join condition is not satisfied. Observe, however, that the join condition is satisfied when the first tuple of D is evaluated against all other tuples in C, thus resulting in the first ten tuples of the 55 tuples in the result. Using the data for relations C and D shown in Figure 11.5, the reader is encouraged to verify why this Theta Join produces a total of 55 tuples.

Outer Join Operators The Join operations discussed to this point are Inner Join operations, meaning that for the relations R and S, only tuples from R that have matching tuples in S (and vice versa) appear in the result. In other words, tuples without a matching (or related) tuple are eliminated from the join result. Tuples with null values in the join attributes are also eliminated. A set of operations called Outer Joins, can be used when we want to keep all the tuples in R, or those in S, or those in both relations in the result of the join, whether or not they have matching tuples in the other relation.

The Left Outer .loin operation, denoted by]X|, keeps every tuple in *the first or left* relation R in R]XI S. If no matching tuple is found in S, then the attributes of S in the join result are filled or "padded" with null values. Thus, in effect a tuple of null values is added to relation S, and any tuple in relation R without a matching tuple in relation S is concatenated with the tuple of null values in relation S. On the other hand, a Right Outer Join, denoted by IX[, keeps *every* tuple in the *second or right* relation S in the result R IX[S. If for a particular tuple in relation S there is no matching tuple in relation R, then for that particular tuple of relation S attributes from relation R are "padded" with null values. A third operation, Full Outer Join, denoted by]X[, keeps all tuples in both the left and right relations when no matching tuples are found, padding them with null values as needed.

Left Outer Join Example. Do a Left Outer Join of relations D and C over their common attributes (D_dcode in relation D and C_d_dcode in relation C).

Relational ^Vlgebra Syntax:

```
D ]X| D_dcode = C_d_dcode C
```

Result:

D_name	D_dcode	D_college	C_name	C_course#	C_credit	C_d_dcode
Economics	1	Arts and Sciences	Intro to Economics	15ECON112	U	1
QA/QM	3	Business	Operations Research	22QA375	U	3
Economics	4	Education	Intro to Economics	18ECON123	U	4
QA/QM	3	Business	Supply Chain Analysis	22QA411	U	3
IS	7	Business	Principles of IS	22IS270	G	7
Mathematics	6	Engineering	Programming in C++	20ECES212	G	6
QA/QM	3	Business	Optimization	22QA888	G	3
Economics	4	Education	Financial Accounting	18ACCT801	G	4
IS	7	Business	Database Concepts	22IS330	U	7
IS	7	Business	Database Principles	22IS832	G	7
IS	7	Business	Systems Analysis	22IS430	G	7
Philosophy	9	Arts and Sciences				

Observe how the sixth tuple of D is concatenated with the tuple of null values from the C relation to produce the twelfth tuple in the result, revealing that the Philosophy Department has yet to offer a course.

Right Outer Join Example. In order to conveniently illustrate the result of a Right Outer Join operation, the relation SS has been created from the SECTION relation. Like SECTION, SS contains 11 tuples, but only four of the eight attributes found in SECTION. These four attributes are: Ss_section#, Ss qtr, Ss year, and Ss_c_course#. The content of relation SS is shown in Figure 11.8.

Ss_section#	Ss_qtr	Ss_year	Ss_c_course#
101	A	2007	22QA375
901	A	2006	22IS270
902	A	2006	22IS270
101	S	2006	22IS330
102	S	2006	22IS330
701	W	2007	22IS832
101	A	2007	20ECES212
101	U	2007	22QA375
101	A	2007	22IS330
101	S	2007	22QA375
101	W	2007	22QA375

Figure 11.8 Relation **SS**

The following shows a Right Outer Join of relations SS and C over their common attributes (Ss_c_course# in relation SS and C_course# in relation C).

Relational Algebra Syntax:

```
SS |X[ Ss_c_course# = C_course# C
```

Result:

Ss_section#	Ss_qtr	Ss_year	Ss_c_course#	C_name	C_course#	C_credit	C_d_dcode
101	A	2007	22QA375	Operations Research	22QA375	U	3
901	A	2006	22IS270	Principles of IS	22IS270	G	7
902	A	2006	22IS270	Principles of IS	22IS270	G	7
101	S	2006	22IS330	Database Concepts	22IS330	U	7
102	S	2006	22IS330	Database Concepts	22IS330	U	7
701	W	2007	22IS832	Database Principles	22IS832	G	7
101	A	2007	20ECES212	Programming in C++	20ECES212	G	6
101	U	2007	22QA375	Operations Research	22QA375	U	3
101	A	2007	22IS330	Database Concepts	22IS330	U	7
101	S	2007	22QA375	Operations Research	22QA375	U	3
101	W	2007	22QA375	Operations Research	22QA375	U	3
				Financial Accounting	18ACCT801	G	4
				Careers Colloquium	06US100	U	
				Systems Analysis	22IS430	G	7
				Intro to Economics	18ECON123	U	4
				Supply Chain Analysis	22QA411	U	3
				Intro to Economics	15ECON112	U	1
				Optimization	22QA888	G	3

Observe how each tuple of C (including those for which a section has not been offered) appears in the result shown above.

Full Outer Join Example. Join the D and C relations, making sure that each tuple from each relation appears in the result.

In order to illustrate a Full Outer Join, assume that it is possible for a course to exist without being affiliated with a department or college. The revised C relation shown below reflects the addition of a one-hour Careers Colloquium course. *For purposes of this example, assume that the business rule requiring each course to be affiliated with a department has been temporarily disabled.*

Revised C Relation

C_name	C_course#	C_credit	C_d_dcode
Intro to Economics	15ECON112	U	1
Operations Research	22QA375	U	3
Intro to Economics	18ECON123	U	4
Supply Chain Analysis	22QA411	U	3
Principles of IS	22IS270	G	7
Programming in C++	20ECES212	G	6
Optimization	22QA888	G	3
Financial Accounting	18ACCT801	G	4
Database Concepts	22IS330	U	7
Database Principles	22IS832	G	7
Systems Analysis	22IS430	G	7
Careers Colloquium	06US100	U	

This example uses the D relation and the revised C relation to illustrate the distinction between a Left, Right, and Full Outer Join. The result of a Left Outer Join of D and G,

- expressed as:

```
D ]X| D_dcode = C_d_dcode C
```

adds a blank tuple to the revised G relation to ensure that each tuple in relation D is reflected in the result shown below.

D_name	D_dcode	D_college	C_name	C_course#	C_credit	C_d_dcode
Economics	1	Arts and Sciences	Intro to Economics	15ECON112	U	1
QA/QM	3	Business	Operations Research	22QA375	U	3
Economics	4	Education	Intro to Economics	18ECON123	U	4
QA/QM	3	Business	Supply Chain Analysis	22QA411	U	3
IS	7	Business	Principles of IS	22IS270	G	7
Mathematics	6	Engineering	Programming in C++	20ECES212	G	6
QA/QM	3	Business	Optimization	22QA888	G	3
Economics	4	Education	Financial Accounting	18ACCT801	G	4
IS	7	Business	Database Concepts	22IS330	U	7
IS	7	Business	Database Principles	22IS832	G	7
IS	7	Business	Systems Analysis	22IS430	G	7
Philosophy	9	Arts and Sciences				

A Right Outer Join of the relation D and the revised C relation, expressed as:

```
D |X[ D_dcode = C_d_dcode C
```

adds a blank tuple to relation D to ensure that each tuple in the revised C relation is reflected in the result shown below.

D_name	D_dcode	D_college	C_name	C_course#	C_credit	C_d_dcode
Economics	1	Arts and Sciences	Intro to Economics	15ECON112	U	1
QA/QM	3	Business	Optimization	22QA888	G	3
QA/QM	3	Business	Supply Chain Analysis	22QA411	U	3
QA/QM	3	Business	Operations Research	22QA375	U	3
Economics	4	Education	Financial Accounting	18ACCT801	G	4
Economics	4	Education	Intro to Economics	18ECON123	U	4
Mathematics	6	Engineering	Programming in C++	20ECES212	G	6
IS	7	Business	Systems Analysis	22IS430	G	7
IS	7	Business	Database Principles	22IS832	G	7
IS	7	Business	Database Concepts	22IS330	U	7
IS	7	Business	Principles of IS	22IS270	G	7
			Careers Colloquium	06US100	U	

A Full Outer Join of D and the revised C relation, expressed as:

```
D ]X[ D_dcode = C_d_dcode C
```

adds a blank tuple to both relation D and the revised G relation to insure that each tuple in each relation is reflected in the result. Observe how each tuple in each relation appears in the result shown below.

D_name	D_dcode	D_college	C_name	C_course#	C_credit	C_d_dcode
Economics	1	Arts and Sciences	Intro to Economics	15ECON112	U	1
QA/QM	3	Business	Operations Research	22QA375	U	3
Economics	4	Education	Intro to Economics	18ECON123	U	4

QA/QM	3	Business	Supply Chain Analysis	22QA411	U	3
IS	7	Business	Principles of IS	22IS270	G	7
Mathematics	6	Engineering	Programming in C++	20ECES212	G	6
QA/QM	3	Business	Optimization	22QA888	G	3
Economics	4	Education	Financial Accounting	18ACCT801	G	4
IS	7	Business	Database Concepts	22IS330	U	7
IS	7	Business	Database Principles	22IS832	G	7
IS	7	Business	Systems Analysis	22IS430	G	7
Philosophy	9	Arts and Sciences				
			Careers Colloquium	06US100	U	

It is left as an exercise for the reader to verify that the union of the results of a Left Outer Join and a Right Outer Join is equal to the result of a Full Outer Join.

11.1.2.4 The Divide Operator

The Divide operator is useful when there is a need to identify tuples in one relation that match *all* tuples in another relation. For example, let R be a relation schema with attributes Aj, A_2, A_n, B_1, B_2, B_m and S be a relation schema with attributes Bj, B_2, B_m In other words, the set of attributes of S *(the divisor)* is a subset of the attributes of R *(the dividend).* The division of R by S can be expressed as R/S. Figure 11.9 contains a schematic view of the Division operation. Observe how the two tuples in relation T represent a subset of the tuples in relation R that match all three tuples in relation S. In order to divide R by S, relations R and S must be division compatible in that the set of attributes of S must be a subset of the attributes of R. In other words, if relation S in Figure 11.9 were also to contain the attribute p, then division compatibility would not exist.

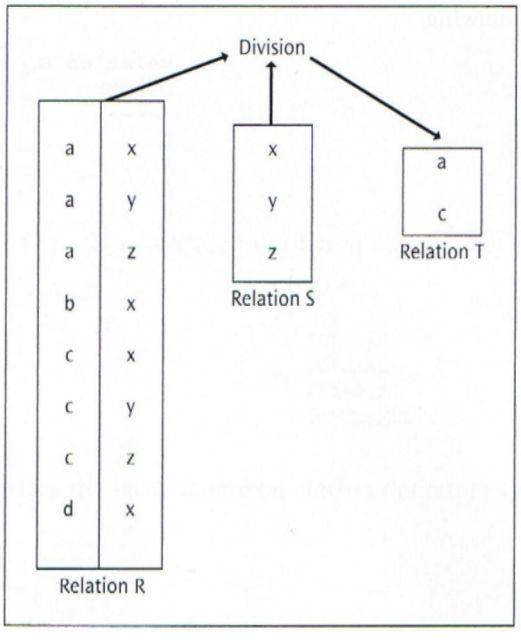


Figure 11.9 The Division operation

For ease of understanding, Kifer, Bernstein, and Lewis (2005) illustrate how the Division operation can be decomposed into a sequence of Projection, Cartesian Product, and • Difference operations as shown here:

$T_1 = \pi_A (R) \times S$	All possible associations between the A values in R and B values in S.
$T_2 = \pi_A(T_1 - R)$	All those A values in R that are not associated in R with every B value in S. These are those A values that should not be in the answer.
$T_3 = \pi_A (R) - T_2$	The quotient: all those A values in R that are associated in R with all B values in S.

Divide Example. List the course numbers of courses that are offered in all quarters during which course sections are offered.

Relational Algebra Syntax:

$$R = \pi_{(Se\_co\_course\#, Se\_qtr)} (SECTION)$$
$$S = \pi_{(Se\_qtr)} (SECTION)$$
$$R \div S$$

Divide Example. List the course numbers of courses that are offered in all quarters during which course sections are offered.

Relational Algebra Syntax:

$$R = \langle Se\_co\_course\#, Se\_qtr \rangle (SECTION)$$
$$S = JT_{(Se\_qtr)} (SECTION)$$
$$R \div S$$

Result:

Relation R

Se_co_course#	Se_qtr
22QA375	A
22QA375	S
22IS270	A
22IS330	S
22IS832	W
20ECES212	A
22QA375	U
22IS330	A
22QA375	S
22QA375	W

Relation S

Se_qtr
A
S
W
U

Expressing the result as a sequence of projection, Cartesian Product, and Difference operations yields the following:

Ti = 71 (Se_co_course#,Se_qtr) (R) X S — All possible associations between courses and quarters during which sections of courses are offered:

π Se_co_course#(R)	S
22QA375	A
22IS270	S
20IS330	W
22IS832	U
20ECES212	

T_1	
22QA375	A
22QA375	S
22QA375	W
22QA375	U
22IS270	A
22IS270	S
22IS270	W
22IS270	U
20IS330	A
20IS330	S
20IS330	W
20IS330	U
22IS832	A
22IS832	S
22IS832	W
22IS832	U
20ECES212	A
20ECES212	S
20ECES212	W
20ECES212	U

T2 = pi() Se_co_course# (Ti - R) — All courses that are not offered during each quarter:

$(T_1 - R)$	
22IS270	S
22IS270	W
22IS270	U
22IS832	A
22IS832	S
22IS832	U
20ECES21	S
20ECES212	W
20ECES212	U

T_2
22IS270
20IS330
20IS832
20ECES212

$T_3 = \pi_{Se\_co\_course\#}(R) - T_2$ — All courses offered during each quarter:

π Se_co_course# (R)	T_2	T_3
22QA375	20IS270	
22IS270	20IS330	22QA375
20IS330	22IS832	
22IS832	20ECES212	
20ECES212		

Tabic 11.1 summarizes the basic relational algebra operators and notation.

Table 11.1 Basic relation algebra operators[1]

Relational Algebra Operator	Purpose	Notation
Select	Selects all tuples that satisfy the selection condition from a relation R.	$\sigma_{\text{<selection condition>}}(R)$
Project	Produces a new relation with only some of the attributes of R, and removes duplicate tuples.	$\pi_{\text{<attribute list>}}(R)$
Cartesian Product	Produces a relation that has the attributes of R_1 and R_2 and includes as tuples all possible combinations of tuples from R_1 and R_2.	$R_1 \times R_2$
Union	Produces a relation that includes all the tuples in R_1 or R_2 or both R_1 and R_2; R_1 and R_2 must be union compatible.	$R_1 \cup R_2$
Intersection	Produces a relation that includes all the tuples in both R_1 and R_2; R_1 and R_2 must be union compatible.	$R_1 \cap R_2$
Difference	Produces a relation that includes all the tuples in R_1 that are not in R_2; R_1 and R_2 must be union compatible.	$R_1 - R_2$
Equijoin	Produces all the combinations of tuples from R_1 and R_2 that satisfy a join condition with only equality comparisons. R_1 and R_2 must be join compatible.	$R_1 [X]_{\text{<join condition>}} R_2$
Natural Join	Produces all the combinations of tuples from R_1 and R_2 that satisfy a join condition with only equality comparisons, except that one of the join attributes is not included in the resulting relation. R_1 and R_2 must be join compatible.	$R_1 *_{\text{<join condition>}} R_2$
Theta Join	Produces all the combinations of tuples from R_1 and R_2 that satisfy a join condition which does not have to involve equality comparisons. R_1 and R_2 must be join compatible.	$R_1 [X]_{\text{<join condition>}} R_2$
Divide	Returns every tuple from R_1 that match all tuples in R_2; R_1 and R_2 must be division compatible.	$R_1 \div R_2$

11.1.2.5 Additional Relational Operators

Some common database requests cannot be performed with the basic relational algebra operations described previously. This section defines three additional operations to express these requests.

The Semi-Join Operation **The Semi-Join operation defines a relation that contains the tuples of R that participate in the join of R with S. In other words, a Semi-Join is defined to**

[1] Source: R. Elmasri and S.B. Navathe (2000). *Fundamentals of Database Systems,* Addison-Wesley.

be equal to the join of R and S. projected back on the attributes of R. The Semi-Join operation can be expressed using the Projection and Join operations as follows:

$$\Pi_R(R \mid X_{A=B} S)$$

where A and B are domain-compatible attributes of R and S, respectively.

Semi-Join Example. List the complete details of all courses for which sections are being offered in the fall (Se_qtr = 'A') 2007 quarter.

Relational Algebra Syntax:

$$\Pi \text{ COURSE}(\text{COURSE} \mid X_{(\text{Co\_course\#=Se\_co\_course\# and Se\_qtr = 'A' and Se\_year = 2007})} \text{ SECTION})$$

Result:

Co_name	Co_course#	Co_credit	Co_college	Co_hrs	Co_dpt_code
Database Concepts	22IS330	U	Business	4	7
Operations Research	22QA375	U	Business	2	3
Programming in C++	20ECES212	G	Engineering	3	6

The Semi-Minus Operation The semi-difference between Rand S (in this order) is defined to be equivalent to:

$$R - \Pi_R(R \mid X_{A=B} S)$$

The result of the Semi-Minus operation is thus the tuples of R that have no counterpart in S.

Semi-Minus Example. List complete details of all courses for which sections are not offered in the fall 2007 quarter.

Relational Algebra Syntax:

$$\text{COURSE} - \Pi_{\text{COURSE}}(\text{COURSE} \mid X_{(\text{Co\_course\#=Se\_co\_course\# and Se\_qtr = 'A' and Se\_year = 2007})} \text{ SECTION})$$

Result:

Co_name	Co_course#	Co_credit	Co_college	Co_hrs	Co_dpt_dcode
Database Principles	22IS832	G	Business	3	7
Financial Accounting	18ACCT801	G	Education	3	4
Intro to Economics	15ECON112	U	Arts and Sciences	3	1
Intro to Economics	18ECON123	U	Education	4	4
Optimization	22QA888	G	Business	3	3
Principles of IS	22IS270	G	Business	3	7
Supply Chain Analysis	22QA411	U	Business	3	3
Systems Analysis	22IS430	G	Business	3	7

Aggregate Functions and Grouping One type of request that cannot be expressed in the basic relational algebra involves mathematical aggregate functions on collections of values from the database. Examples of such functions include retrieving the sum, average, maximum, and minimum of a scries of numeric values. Another type of request involves the grouping of tuples in a relation by the value of some of their attributes and then applying an aggregate function independently to each group. An example would be to group the courses taken by course number and then count the number of students taking each course during the year 2007.

The Aggregate Function operation can be defined using the symbol J to specify these types of requests as follows:

$$\langle \text{grouping attributes} \rangle \ \mathcal{F} \ \langle \text{function list} \rangle \ (R)$$

where <grouping attributes> is a list of attributes of the relation specified in R, and <function list> is a list of (<function> <attribute>) pairs where:

- <function> is one of the allowed functions, such as SUM, AVERAGE, MAXIMUM, MINIMUM, COUNT, and
- <attribute> is an attribute of the relation specified by R.

The resulting relation has the grouping attributes plus one attribute for each element in the function list.

Aggregate Functions and Grouping Example. Compute the average number of hours for the courses offered by each college.

Relational Algebra Syntax:

$$Co\_college\ \mathcal{F}\ AVERAGE\ Co\_hrs\ (COURSE)$$

Result:

Co_college	Average
Arts and Sciences	3
Business	3
Education	3.5
Engineering	3

11.2 Structured Query Language (SQL)

SQL is the standard language for manipulating relational databases. The language was created to facilitate implementation of relational algebra in a database. Like relational algebra, SQL uses one or more relations as input and produces a single relation as output.[5] This section and Chapter 12 contain an informal overview of the use of the SQL SELECT statement for information retrieval. Through the use of numerous example queries, Section 11.2 introduces the SELECT statement's features gradually beginning with queries based on a single table followed by queries that retrieve data from several tables. As was the case in Chapter 10, except where indicated, the examples in this chapter as well as in Chapter 12 are based on the syntax associated with the SQL-92 standard.

Before proceeding further, it is important to note that space docs not permit a thorough and complete discussion of all features of the SQL SELECT statement as well as other features associated with SQL's data definition language,[6] data manipulation language, and data control language. Hundreds of books and reference manuals are available on SQL, and the interested reader is encouraged to consider such sources for more comprehensive syntax than that shown in this book. In addition, many Web sites have information about standard SQL and its implementation under various database platforms. For example, considerable online documentation is available from the Microsoft developers site *(http://msdn.microsoft.com)* and the Oracle TechNet site *(http://technet.oracle.com)*. The O'Reilly site *(http://www.ora.com)* also publishes information online about SQL.

[5] SQL uses the terms *table, row,* and *column* for relation, tuple, and attribute, respectively. Thus, when the discussion focuses on SQL the terms table, row, and column will be used. When the discussion focuses on relational algebra operations, the terms relation, tuple, and attribute will be used.

[6] Three more advanced features of SQL-92 are assertions, triggers, and views. Assertions and triggers are SQL/DDL facilities for capturing constraints while views are "virtual tables" created using SQL/DDL. These features are discussed in Chapter 12.

As mentioned previously, the SQL SELECT statement is used to retrieve (i.e., query) data from tables (i.e., relations) and is used in conjunction with all relational algebra operations. For example, using a SELECT statement, it is possible to view all the columns and rows within a table (i.e., to execute a relational algebra Selection operation) or specify that only certain columns and rows be viewed (execute a Selection operation followed by a Projection operation).

The syntax for an SQL statement gives the basic structure, or rules, required to execute the statement. The basic form of the SQL SELECT statement is called a select-from-where block and contains the three clauses SELECT, FROM, and WHERE in the following form:

```
SELECT <column list>
FROM <table list>
WHERE <condition>
```

where:

- <column list> is a list of column names whose values are to be retrieved by the query,
- <table list> is a list of the table names required to process the query,[7] and
- <condition> is a conditional (Boolean) expression that identifies the rows to be retrieved by the query.

Kifer, Bernstein, and Lewis (2005) provide a useful description of the basic algorithm for evaluating an SQL query.

1. The FROM clause is evaluated. It produces a table that is the Cartesian Product of the tables listed as arguments. If a table occurs more than once in the FROM clause, then this table occurs as many times in the Cartesian Product.
2. The WHERE clause is evaluated by taking the table produced in Step 1 and processing each row individually. Values from the row are substituted for the column names in the condition and the condition is evaluated. The table produced by the WHERE clause contains exactly those rows for which the condition evaluates to true.
3. The SELECT clause is evaluated. It takes the table produced in Step 2 and retains only those columns that are listed as arguments. The resulting table is output by the SELECT statement.

Additional features of the language result in the addition of steps to this algorithm. Also, certain steps need not be present in a particular evaluation. For example, Step 2 above is not evaluated if there is no WHERE clause.[8] Likewise, if the FROM clause refers to only a single table, the Cartesian Product is not formed.

Three other clauses can also be used in an SQL SELECT statement:

```
GROUP BY group_by_expression
HAVING group_condition
ORDER BY column name(s)
```

[7] As we will see, the table list may include both database views as well as inline views.

[8] While the \MIERE clause is optional, the **SELECT** and FROM clauses are not.

where:

- *group_by_expression* forms groups of rows with the same value,
- *group_condition* filters the groups subject to some condition, and
- *column name(s)* specifies the order of the output.

In this book, a semicolon (;) follows the last clause in an SQL SELECT statement.[9] In addition, while the keywords and clauses (e.g., SELECT, FROM, WHERE, GROUP BY, etc.) in an SQL SELECT statement appear capitalized here, only conditions in the WHERE clause that involve non-numeric (i.e., character or string) literals are actually case sensitive.

Many of the illustrations in this chapter make use of the DEPARTMENT, COURSE, STUDENT, GRAD_STUDENT, SECTION, TAKES, PROFESSOR, TEXTBOOK, and USES tables from the Madeira College registration system described in Chapter 10. The initial form of these nine tables is shown in Figure 11.1 at the beginning of this chapter. Each illustration has been tested using Oracle. Thus please note that not all syntax shown will work on other platforms such as IBM DB2, Microsoft SQL Server, and MySQL. For more comprehensive syntax than that shown in conjunction with the examples in this book, refer to the SQL reference material of the respective database platform.

11.2.1 SQL Queries Based on a Single Table

This section considers several SQL features in the context of queries that involve a single table. Included in the discussion are illustrations of (a) the Select and Project operators, the two unary relational algebra operators based on a single relation, (b) the hierarchy of operations, (c) the handling of null values, and (d) pattern matching.

The examples in the remainder of this chapter are labeled in accordance with the following scheme.

- The first two elements of the section number are ignored (e.g., in Section 11.2.1.1, the 11.2 portion is ignored)
- The examples within the section are numbered from 1 to n beginning with the third element of the section number.

Thus you will find that the three examples in Section 11.2.1.1 are labeled 1.1.1 through 1.1.3; the ten examples in Section 11.2.1.6 are labeled 1.6.1 through 1.6.10; and the three examples in Section 11.2.3.1 are labeled 3.1.1.1 through 3.1.1.3.

11.2.1.1 Examples of the Selection Operation

Three examples of the use of the relational algebra Select operator in a Selection operation appear in Section 11.1.1.1. SQL SELECT statements that correspond to each of these examples follow.

Example 1.1.1 (Corresponds to Selection Example 1 in Section 11.1.1.1). Which courses are three-hour courses?

SQL SELECT Statements:

```
SELECT COURSE.CO_NAME, COURSE.CO_COURSE#, COURSE.CO_CREDIT, COURSE.CO_
COLLEGE, COURSE.CO_HRS, COURSE.CO_DPT_DCODE
FROM COURSE
WHERE COURSE.CO_HRS = 3;
```

[9] The SQL-92 standard actually prescribes the semicolon as a statement terminator only in the case of embedded SQL.

or:

```
SELECT *
FROM COURSE
WHERE COURSE.CO_HRS = 3;
```

Result:

Co_name	Co_course#	Co_credit	Co_college	Co_hrs	Co_dpt_dcode
Intro to Economics	15ECON112	U	Arts and Sciences	3	1
Supply Chain Analysis	22QA411	U	Business	3	3
Principles of IS	22IS270	G	Business	3	7
Programming in C++	20ECES212	G	Engineering	3	6
Optimization	22QA888	G	Business	3	3
Financial Accounting	18ACCT801	G	Education	3	4
Database Principles	22IS832	G	Business	3	7
Systems Analysis	22IS430	G	Business	3	7

The asterisk (*) means that all columns from the table are to be selected. The WHERE clause tells SQL to search the rows in the COURSE table and to return (i.e., display) only those rows where the value of COURSE.Co_hrs is equal to exactly 3. In addition, while not required, in an SQL SELECT statement it is a good idea to prefix each column name with its table name in order to minimize ambiguity and confusion.[10]

Since SQL SELECT statements are not case sensitive, the two SQL statements shown above could also have been written as follows:

```
select co_name, co_course#, co_credit, co_college, co_hrs, co_dpt_dcode
from course
where co_hrs = 3;

select *
from course
where co_hrs = 3;
```

While each of these statements is semantically correct, once again we recommend that each column name referenced be preceded (i.e., prefixed) by the table name COURSE in either lower or upper case.

The ORDER BY clause is used to specify how the results of a query are to be sorted. The default sort is an ascending sort. The keywords, ASCENDING and DESCENDING (abbreviated ASC and DESC), are used to control the sorting of each column. Thus the query:

```
SELECT *
FROM COURSE
WHERE COURSE.CO_HRS = 3
ORDER BY COURSE.CO_DPT_DCODE DESC;
```

[10] The convention of prefixing each column name with its table name is used throughout this chapter. SQL however, permits duplicate column names as long as they appear in different tables. When two tables involved in a SELECT statement share a common column name or names, the qualification of the column name(s) with the appropriate table name is required if the column name appears in the <attribute lists or in the WHERE condition. Otherwise ambiguity will exist and an error message of the form "column ambiguously defined" will appear. Since the column names used in the tables for Madeira College adhere to the convention for developing attribute names described in Chapter 6 (see Section 6.2), ambiguous column names will not occur in the examples in this book.

lists the three-hour courses in descending order by the department number in which the course is offered. The query:

```
SELECT *
FROM COURSE
WHERE COURSE.CO_HRS = 3
ORDER BY COURSE.CO_DPT_DCODE DESC, COURSE.CO_NAME;
```

lists the three-hour courses in descending order by the department number in which the course is offered and in ascending order by course name within each department.

Example 1.1.2 (Corresponds to Selection Example 2 in Section 11.1.1.1). Which courses offered by department 7 are three-hour courses?

SQL SELECT Statement:

```
SELECT *
FROM COURSE
WHERE COURSE.CO_DPT_DCODE = 7 AND COURSE.C0_HRS = 3;
```

Result:

Co_name	Co_course#	Co_credit	Co_college	Co_hrs	Co_dpt_dcode
Principles of IS	22IS270	G	Business	3	7
Database Principles	22IS832	G	Business	3	7
Systems Analysis	22IS430	G	Business	3	7


```
SELECT *
FROM COURSE
WHERE COURSE.CO_HRS = 3
ORDER BY COURSE.CO_DPT_DCODE DESC, COURSE.CO_NAME;
```

lists the three-hour courses in descending order by the department number in which the course is offered and in ascending order by course name within each department.

Example 1.1.2 (Corresponds to Selection Example 2 in Section 11.1.1.1). Which courses offered by department 7 are three-hour courses?

SQL SELECT Statement:

```
SELECT *
FROM COURSE
WHERE COURSE.CO_DPT_DCODE = 7 AND COURSE.CO_HRS = 3;
```

Result:

Co_name	Co_course#	Co_credit	Co_college	Co_hrs	Co_dpt_dcode
Principles of IS	22IS270	G	Business	3	7
Database Principles	22IS832	G	Business	3	7
Systems Analysis	22IS430	G	Business	3	7

Use of the logical operator AND requires that *both* conditions (i.e., **COURSE.Co_dpt_dcode = 7** and **COURSE.Co_hrs = 3**) be satisfied.

Example 1.1.3. Which sections have a maximum number of students greater than 30 or are offered in Lindner 110?

SQL SELECT Statement:

```
SELECT *
FROM SECTION
WHERE SECTION.SE_MAXST > 30 OR SECTION.SE_ROOM = 'Lindner 110';
```

Result:

Se_section#	Se_qtr	Se_year	Se_time	Se_maxst	Se_room	Se_co_course#	Se_pr_profid
901 A	2006	WI800	35	Rhodes 611	22IS270	SR85977	
101 S	2006	FI045	29	Lindner 110	22IS330	SR85977	
102 S	2006	HI045	29	Lindner 110	22IS330	CC49234	
701 M	2007	MI000	33	Braunstien 211	22IS832	CC49234	

Use of the logical operator OR requires that either one or both conditions (i.e., **SECTION.Se_maxst > 30** OR **SECTION.Se_room = 'Lindner 110'**) be satisfied. Observe that in Example 1.1.3 above, none of the rows in the SECTION tabic satisfy both conditions of the WHERE clause. Also, whenever a character or string literal (e.g., Lindner 110) is used as part of a condition, the value must be enclosed within *single quotation marks.* Character or string literals enclosed in single quotation marks are case sensitive. Thus the WHERE condition will not be true if anything other than Lindner 110 appears inside the single quotation marks.

Combining AND or OR in the same logical expression must be done with care. When AND and OR appear in the same WHERE clause, all the ANDs are performed first; then all the ORs are performed. In this way, AND is said to have a higher precedence than OR.

For example, the WHERE clause:

```
WHERE COURSE.CO_CREDIT = 'U' AND COURSE.CO_COLLEGE = 'Business' OR
COURSE.CO_COLLEGE = 'Engineering'
```

is evaluated in the following manner:

1. **Each of the component expressions is evaluated yielding a value that is either "true" or "false."**
2. **The results of COURSE.Co_credit = 'IT AND COURSE.Co_college = 'Business' are combined, yielding a result that is "true" if both expressions are true, and otherwise is "false."**
3. **The result of COURSE.Co_college = 'Engineering' is evaluated as being either "true" or "false."**
4. **The results of Steps 2 and 3 are combined, yielding a result that is "true" if either result is "true" and "false" if both results are "false."**

Applying this evaluation scheme to the first row of the COURSE table yields the following results:

1. **COURSE.Co_credit = *'IT* — True**
   COURSE.Co_college = 'Business' — False
   COURSE.Co_college = 'Engineering' — False
2. **COURSE.Co_credit = U' AND COURSE.Co_college = 'Business' — False**
3. **COURSE.Co_college = 'Engineering' — False**
4. **Since both steps 2 and 3 are false, the overall result *is false* and the first row fails to satisfy the WHERE clause.**

In SQL, all operators are arranged in a hierarchy that determines their precedence. In any expression, operations are performed in order of their precedence, from highest to lowest. When operators of equal precedence are used next to each other, they are performed from left to right. The precedence of common logical operators in SQL is:

1. **All of the comparison operators have equal precedence**
   `= , <>, <=, <, >=, >`
2. **NOT**
3. **AND**
4. **OR**

When the normal rules of operator precedence do not fit the needs, one can override them by placing part of an expression in parentheses. That part of the expression will be evaluated first; then the rest of the expression will be evaluated.

The following examples illustrate the incorporation of logical operators in a SELECT statement. Note that Example 1.2.1 is based on the WHERE clause discussed above.

Example 1.2,1

```
SELECT *
FROM COURSE
WHERE COURSE.CO_CREDIT = 'U'
```

```
AND COURSE.CO_COLLEGE = 'Business'
OR COURSE.CO_COLLEGE = 'Engineering';
```

Result:

Co_name	Co_course#	Co_credit	Co_college	Co_hrs	Co_dpt_dcode
Operations Research	22QA375	U	Business	2	3
Supply Chain Analysis	22QA411	U	Business	3	3
Programming in C++	20ECES212	G	Engineering	3	6
Database Concepts	22IS330	U	Business	4	7

In Example 1.2.2, the rows with COURSE.Co_college equal to Business or Engineering are selected first as a result of the presence of the parentheses. However, in order to be included in the result, COURSE.Co_credit must be equal to U. This eliminates the graduate Programming in C++ course offered in the College of Engineering.

Example 1.2.2


```
SELECT *
FROM COURSE
WHERE COURSE.CO_CREDIT = 'U'
AND (COURSE.CO_COLLEGE = 'Business'
OR COURSE.CO_COLLEGE = 'Engineering');
```

Result:

Co_name	Co_course#	Co_credit	Co_college	Co_hrs	Co_dpt_dcode
Operations Research	22QA375	U	Business	2	3
Supply Chain Analysis	22QA411	U	Business	3	3
Database Concepts	22IS330	U	Business	4	7

In Example 1.2.3, each of the 11 rows satisfies at least one of the conditions in parentheses (COURSE.Co_college <> 'Business' OR COURSE.Co college <> 'Engineering'). Five of these rows are also associated with a course that is offered for undergraduate credit.

Example 1.2.3

```
SELECT *
FROM COURSE
WHERE (COURSE.CO_COLLEGE <> 'Business'
OR COURSE.CO_COLLEGE <> 'Engineering')
AND CO_CREDIT = 'U';
```

Result:

Co_name	Co_course#	Co_credit	Co_college	Co_hrs	Co_dpt_dcode
Intro to Economics	15ECON112	U	Arts and Sciences	3	1
Operations Research	22QA375	U	Business	2	3
Intro to Economics	18ECON123	U	Education	4	4
Supply Chain Analysis	22QA411	U	Business	3	3
Database Concepts	22IS330	U	Business	4	7

In Example 1.2.4, the condition COURSE.Co college <> 'Engineering' AND COURSE.Co credit = 'U' is evaluated first and is satisfied by the rows for the five undergraduate courses. The condition COURSE.Co_college <> 'Business' is evaluated next and satisfied by the rows for the two Intro to Economics courses, the graduate Programming in C++

course, and the graduate Financial Accounting course. The rows for the two Intro to Economics courses satisfy both conditions, while each of the other five rows satisfies one condition.

Example 1.2.4

```
SELECT *
FROM COURSE
WHERE COURSE.CO_COLLEGE <> 'Business'
OR COURSE.CO_COLLEGE <> 'Engineering'
AND CO_CREDIT = 'U';
```

Result:

Co_name	Co_course#	Co_credit	Co_college	Co_hrs	Co_dpt_dcode
Intro to Economics	15ECON112	U	Arts and Sciences	3	1
Operations Research	22QA375	U	Business	2	3
Intro to Economics	18ECON123	U	Education	4	4
Supply Chain Analysis	22QA411	U	Business	3	3
Programming in C++	20ECES212	G	Engineering	3	6
Financial Accounting	18ACCT801	G	Education	3	4
Database Concepts	22IS330	U	Business	4	7

The keywords IN or NOT IN can also be used as comparison operators. IN is evaluated in the context of being "equal to any member of" a set of values. Thus Example 1.2.5 is equivalent to Example 1.2.2.

Example 1.2.5

```
SELECT *
FROM COURSE
WHERE COURSE.CO_CREDIT = 'U'

AND COURSE.CO_COLLEGE IN ('Business', 'Engineering');
```

Result:

Co_name	Co_course#	Co_credit	Co_college	Co_hrs	Co_dpt_dcode
Operations Research	22QA375	U	Business	2	3
Supply Chain Analysis	22QA411	U	Business	3	3
Database Concepts	22IS330	U	Business	4	7

The logical operator NOT reverses the result of a logical expression. NOT can be used to precede any of the comparison operators =, <>, <=, <, >=, > as well as the word IN. Thus Example 1.2.6 displays all courses offered for undergraduate credit that are not offered at either the College of Business or the College of Engineering.

Example 1.2.6

```
SELECT *
FROM COURSE
WHERE COURSE.CO_CREDIT = 'U'

AND COURSE.CO_COLLEGE NOT IN ('Business', 'Engineering');
```

Result:

Co_name	Co_course#	Co_credit	Co_college	Co_hrs	Co_dpt_dcode
Intro to Economics	15ECON112	U	Arts and Sciences	3	1
Intro to Economics	18ECON123	U	Education	4	4

So far, the conditions in the WHEPvE clause have involved comparisons where the operands were of the form:

```
<column namexcomparison operatorxconstant value>
```

and the constant value was either a numeric constant or a character constant (i.e., character string). However, as we will see, the operands of a comparison operator can be expressions, not just a simple column name or constant. For example, suppose we are interested in identifying all professors with a monthly salary that exceeds S6,000.

Example 1.2.7

```
SELECT *
FROM PROFESSOR
WHERE PROFESSOR. PR_SALARY/12 > 6000;
```

Result:

Pr_name	Pr_empid	Pr_phone	Pr_office	Pr_birthdate	Pr_datehired	Pr_dpt_dcode	Pr_salary
Mike Faraday	FM49276	5235568492	249 McMicken	1960-08-26	1996-05-01	1	92000
Chelsea Bush	BC65437	5235567777	227 Lindner	1946-09-03	1993-05-01	3	77000
Tony Hopkins	HT54347	5235569977	324 Lindner	1949-11-24	1997-01-20	3	77000
Alan Brodie	BA54325	5235569876	238 Lindner	1944-01-14	2000-05-16	3	76000
Marie Curie	CM65436	5235569899	331 Dyer	1972-02-29	1999-10-22	4	99000
John Nicholson	NJ43728	5235569999	324 Dyer	1966-05-01	2003-06-22	4	99000

If we follow this Selection operation with a Projection operation, the query in Example 1.2.7 could be rewritten to display just the name and monthly salary of each qualifying professor.

```
SELECT PROFESSOR.PR_NAME, PROFESSOR.PR^SALARY/12 AS "Monthly Salary"
FROM PROFESSOR
WHERE PROFESSOR. PR_SALARY/12 > 6000;
```

Result:

Pr_name	Monthly Salary
Mike Faraday	76611.66667
Chelsea Bush	64111.66667
Tony Hopkins	64111.66667
Alan Brodie	6333.33333
Marie Curie	8250
John Nicholson	8250

Rather than displaying the heading of the second column as **PROFESSOR.Pr_salary/12**, SQL allows the column name to be changed with the use of the keyword AS[1] and thus provide a more descriptive column heading as a column alias. It should be noted that this column alias cannot be used in other clauses associated with the SELECT statement. The TRUNC function of the form TEUNC **(PROFESSOR.Pr_salary/12,2)**, or the ROUND function of the form ROUND **(PROFESSOR.Pr_salary/12.2)**, could be used to either truncate or round each monthly salary to two places to the right of the decimal point.

[1] The keyword, AS, is optional and is often used in the column list to distinguish between the column name and column alias.

WHERE clauses can also refer to a range of values through use of the comparison operator BETWEEN, which searches for rows in a specific range of values. For example, suppose we are interested in identifying the name and monthly salary of all professors whose monthly salary is between S6,000 and S7,000.

Example 1.2.8

```
SELECT PROFESSOR.PR_NAME, PROFESSOR.PR_SALARY/12 AS "Monthly Salary"
FROM PROFESSOR
WHERE PROFESSOR.PR_SALARY/12 BETWEEN 6000 AND 7000;
```

Result:

```
Pr_name          Monthly Salary
---------------  --------------
Chelsea Bush        64111.66667
Tony Hopkins        64111.66667
Alan Brodie          6333.33333
```

The BETWEEN operator is inclusive (i.e., professors with a monthly salary of exactly $6,000 or 87,000 would also be included in the result) and can be applied to all data types. Further, NOT can also be used with the BETWEEN operator. For example, the query in Example 1.2.9 identifies all professors whose monthly salary is outside the range of from $6,000 to $7,000. A close inspection of the results indicates that the two professors without a salary (John B. Smith and Tiger Woods) do not appear. Afore: Professor John Smith has a $45,000 annual salary.

Example 1.2.9

```
SELECT PROFESSOR.PR_NAME, ROUND(PROFESSOR.PR_SALARY/12,0) AS
"Monthly Salary"
FROM PROFESSOR
WHERE PROFESSOR.PR_SALARY/12 NOT BETWEEN 6000 AND 7000;
```

Result:

```
Pr_name         Monthly Salary
--------------- --------------
John Smith                3750
Mike Faraday              7667
Kobe Bryant               5500
Ram Raj                   3667
Prester John              3667
Jessica Simpson           5583
Laura Jackson             3583
Marie Curie               8250
Jack Nicklaus             5583
John Nicholson            8250
Sunil Shetty              5333
Katie Shef                5417
Cathy Cobal               3750
Jeanine Troy              3750
Mike Crick                5750
```

Section 11.2.1.5, "Handling Null Values," explains why John B. Smith and Tiger Woods are not included in the results for either Example 1.2.8 or 1.2.9. Section 11.2.2 illustrates how the WHERE and ON clauses have comparisons where operands of the form:

<column name><comparison operator><coluron name>

are used to express join conditions.

11.2.1.3 Examples of the Projection Operation

Examples illustrating use of the relational algebra Project operator in a Projection operation appear in Section 11.1.1.2. SQL SELECT statements that correspond to one of these examples follow along with other examples illustrating how a Projection operation typically follows a Selection operation.

Example 1.3.1 (Corresponds to Projection Example 1 in Section 11.1.1.2). Which colleges offer courses?

SQL SELECT Statement:


```
SELECT COURSE.CO_COLLEGE
FROM COURSE;
```

Execution of the SELECT statement given above, since it refers to only one of the six columns in the COURSE table, actually generates duplicate rows and thus the result shown below does not constitute a relation.

Result:

```
Co_college
--------------------
Arts and Sciences
Business
Education
Business
Business
Engineering
Business
Education
Business
Business
Business
```

However, if the qualifier DISTINCT is specified as part of the SELECT statement, duplicate rows are removed. In other words, in SQL, without use of the qualifier DISTINCT, the general form of the SELECT statement is:

```
SELECT ALL <column list>
FROM  <table list>
WHERE <condition>
```

where ALL retains duplicate values in queries (note that ALL is the default as compared to DISTINCT).

Thus the revised SELECT statement with the DISTINCT qualifier produces the following result:

```
SELECT DISTINCT COURSE.CO_COLLEGE
FROM COURSE;
```

Result:

```
Co_college
----------
Arts and Sciences
Business
Education
Engineering
```

Example 1.3.2. What is the name and status of each student?

SQL SELECT Statement:

```
SELECT STUDENT.ST_NAME, STUDENT.ST_STATUS
FROM STUDENT;
```

Result:

```
St_name         St_status
--------------- ---------
Elijah Baley    Part time
Daniel Olive    Full time
Wanda Seldon    Full time
Gladis Bale     Full time
Shweta Gupta    Full time
Troy Hudson     Part time
Rick Fox        Full time
Vanessa Fox     Full time
Jenna Hopp      Full time
David Sane      Part time
Tim Duncan      Part time
Joumana Kidd    Part time
Jenny Aniston   Full time
Poppy Kramer    Full time
Sweety Kramer   Full time
Diana Jackson   Part time
```


In most cases, several relational algebra operations are applied one after the other (e.g., a Selection operation is followed by a Projection operation).

Example 1.3.3. What are the names of the courses offered in the College of Business?

SQL SELECT Statement:

```
SELECT COURSE.CO_NAME
FROM COURSE
WHERE COURSE.CO_COLLEGE = 'Business';
```

Result:

```
Co_name
---------------------
Operations Research
Supply Chain Analysis
Principles of IS
Optimization
Database Concepts
Database Principles
Systems Analysis
```

Example 1.3.4. What are the names and addresses of all part time students?

SQL SELECT Statement

```
SELECT STUDENT.ST_NAME, STUDENT.ST_ADDRESS
FROM STUDENT
WHERE ST_STATUS = 'Part time';
```

Result:

```
St_name         St_address
--------------- --------------------
Elijah Baley    2920 Scioto Street
Troy Hudson
David Sane      245 University Avenue
Tim Duncan
Joumana Kidd    2920 Scioto Street
Diana Jackson   2920 Scioto Street
```


Examples 1.2.7 - 1.2.9 in Section 11.2.1.2 illustrate how an expression can be included in column-list of a query. The following query in Example 1.3.5 illustrates that an expression that involves dates can also be part of a column-list of a query.

Example 1.3.5. What was the age (in years) of each professor in department 3 when hired?

SQL SELECT Statement:

```
SELECT PROFESSOR.PR_NAME,
TRUNC((PROFESSOR.PR_DATEHIRED - PROFESSOR.PR_BIRTHDATE)/365.25,0)
"Age When Hired"
FROM PROFESSOR
WHERE PROFESSOR.PR_DPT_DCODE = 3;
```

Result:

```
Pr_name          Age When Hired
---------------- --------------
Chelsea Bush                 46
Tony Hopkins                 47
Alan Brodie                  56
Jessica Simpson              40
Laura Jackson                26
```

Date arithmetic allowing one date to be subtracted from another date is discussed in Chapter 12.

11.2.1.4 Grouping and Summarizing

SQL allows the grouping of rows into sets; it then can summarize data in such a way that one row is returned for each set. The GROUP BY and HAVING clauses facilitate the grouping, while built-in aggregate functions are used in conjunction with grouping.

Aggregate functions take as input a set of values, one from each row in a group of rows, and return one value as output. Common aggregate functions include:

- **COUNT (x) — counts the number of non-null values in a set of values**
- **SUM (x) — sums all numbers in a set of values**
- **AVG (x) — computes the average of a set of numbers in a set of values**

- **MAX (x) — computes the maximum of a set of numbers in a set of values**
- **MIX (x) — computes the minimum of a set of numbers in a set of values**

Note: x **represents a numeric column name.**

Example 1.4.1. Count the number of salaries in the PROFESSOR table and, at the same time, display the sum of the salaries, the average salary, the maximum salary, and the minimum salary.

SQL SELECT Statement:

```
SELECT COUNT(PR_SALARY), SUM(PR_SALARY),AVG(PR_SALARY),
MAX(PR_SALARY) , MIN(PR_SALARY)
FROM  PROFESSOR;
```

Result:

```
COUNT(Pr_salary)  SUM(Pr_salary)  AVG(Pr_salary)  MAX(Pr_salary)  MIN(Pr_salary)

              18         1184000      65777.7778           99000           43000
```


Note that while there are 20 rows in the PROFESSOR tabic, the salaries of two professors (John B. Smith and Tiger Woods) are unknown and thus a null value is stored in the respective Pr_salary column for these professors. Section 11.2.1.5 discusses the impact of null values on aggregate functions.

Aggregate functions are often used in conjunction with groups of rows rather than with all rows in a table. The column or columns on which the grouping takes place are called the grouping column(s). Doing this requires the application of the GROUP BY clause. The GROUP BY clause divides data into sets (i.e., groups) based on the contents of specified columns. The general form of the GROUP BY clause is:

```
GROUP BY column name,  [,column name,...]
```

The grouping column(s) must also appear in the SELECT clause so that the value from applying each function to a group of rows appears along with the value of the grouping column(s).

Example 1.4.2. Count the number full-time and part-time students.

SQL SELECT Statement:

```
SELECT STUDENT.ST_STATUS,  COUNT(*)
FROM STUDENT
GROUP BY STUDENT.ST_STATUS;
```

Result:

```
St_status    COUNT(*)
-----------  ----------
Full time          10
Part time           6
```

Example 1.4.3. Calculate the average salary of the professors in each department.

SQL SELECT Statement:

```
SELECT PROFESSOR.PR_DPT_DCODE, AVG(PROFESSOR.PR_SALARY)
FROM PROFESSOR
GROUP BY PROFESSOR.PR_DPT_DCODE;
```

Result:

```
Pr_dpt_dcode AVG(PROFESSOR.Pr_salary)
------------ ------------------------
           1              67666.6667
           3                   68000
           4              88333.3333
           6                   44000
           7                   58000
           9                   57000
```


As mentioned above, grouping can be done on a combination of columns. Example 1.4.4 illustrates the use of the GROUP BY clause to count the number of sections offered during each year and quarter combination.

Example 1.4.4. Count the number of sections offered during each quarter of each year.

SQL SELECT Statement:

```
SELECT SECTION.SE_YEAR, SECTION.SE_QTR, COUNT(*)
FROM SECTION
GROUP BY SECTION.SE_YEAR, SECTION.SE_QTR;
```

Result:

```
Se_year Se_qtr   COUNT(*)
------- ------ ----------
   2006 A               2
   2006 S               2
   2007 A               3
   2007 S               1
   2007 U               1
   2007 W               2
```

The HAVING clause is used in conjunction with the GROUP RY clause to place restrictions on the rows returned by the GROUP BY clause in a query. A condition in a HAVING clause must always involve an aggregation. In addition, a HAVING clause cannot be used apart from an associated GROUP BY clause.

Example 1.4.5. Calculate the average salary of the professors in each department for those departments where the minimum salary of a professor is $45,000 or more.

SQL SELECT Statement:

```
SELECT PROFESSOR.PR_DPT_DCODE, AVG(PROFESSOR.PR_SALARY)
FROM PROFESSOR
GROUP BY PROFESSOR.PR_DPT_DCODE
HAVING MIN(PROFESSOR.PR_SALARY) >= 45000;
```

Result:

```
Pr_dpt_dcode AVG(PROFESSOR.Pr_salary)
------------ ------------------------
           1              67666.6667
           4              88333.3333
```

```
7          58000
9          57000
```

Note that departments 3 and 6 do not appear in the results because there is at least one professor in each department with a salary less than S45,000. In addition, observe that the calculation of the average salary of the professors in department 9 ignores Tiger Woods since his salary is a null value (i.e., is not available). The following section goes into more detail on the handling of null values. In addition, Section 11.2.2.3 describes how to join tables in SQL, which would represent one way to revise the query in Example 1.4.5 in order to display department names instead of department numbers.

11.2.1.5 Handling Null Values

It is important to understand how null values are handled in SQL. A data field without a value in it is said to contain a null value. A null value can occur in two situations: (a) where a value is unknown (e.g., the salary of an employee is unknown or unavailable), and (b) where a value is not meaningful (e.g., in a column representing "commission" for an employee who is not a salesperson and thus is not eligible for a commission).

A number field containing a null value is different from one that contains a value of zero. Null values are displayed as blanks while zero values are displayed as numeric zeros. A null value will evaluate to null in any expression. For example, a null value added to 15 results in a null value; a null value minus a null value yields a null value, not zero. The aggregate function COUNT(*) counts all null and not null rows in a table; COUNT (attribute) counts all rows whose attribute value is not null. Other SQL aggregate functions ignore null values in their computation.

This section consists of examples that illustrate the impact of null values in SQL. Each of the examples is based on the textbook table whose structure and initial contents follow.

```
CREATE TABLE TEXTBOOK
(TX_ISBN      VARCHAR(14)    CONSTRAINT PK_10TB PRIMARY KEY,
TX_TITLE      VARCHAR(22)    CONSTRAINT NN_10TIT NOT NULL,
TX_YEAR       INTEGER(4),
TX_PUBLISHER  CHAR(13));
```

```
SELECT *
FROM TEXTBOOK;
```

Tx_isbn	Tx_title	Tx_year	Tx_publisher
000-66574998	Database Management	1999	Thomson
003-6679233	Linear Programming	1997	Prentice-Hall
001-55-435	Simulation Modeling	2001	Springer
118-99898-67	Systems Analysis	2000	Thomson
77898-8769	Principles of IS	2002	Prentice-Hall
0296748-99	Economics For Managers	2001	
0296437-1118	Programming in C++	2002	Thomson
012-54765-32	Fundamentals of SQL	2004	
111-11111111	Data Modeling	2006	

The behavior of null values in SQL can on occasion be surprising or unintuitive. In order to illustrate differences between how SQL handles a null value (defined as a character string zero characters long) versus a character string that consists of a single blank

space, a null value appears in the **TEXTBOOK.Tx_publisher** column for both the titles Economics For Managers and Fundamentals of SQL[12] while the value in the **TEXTBOOK.Tx_publisher** column for the title Data Modeling consists of a single blank space.

As expected, both the previous query and the following query (labeled Example 1.5.1) display the **TEXTBOOK.Tx_publisher** column for Economics For Managers, Fundamentals of SQL, and Data Modeling as blank values in the **TEXTBOOK.Tx_publisher** column.

Example 1.5.1

```
SELECT TEXTBOOK.TX_PUBLISHER
FROM TEXTBOOK;
```

Result:

```
Tx_publisher

Thomson
Prentice-Hall
Springer
Thomson
Prentice-Hall

Thomson

9 rows selected.[13]
```


A query (see Example 1.5.2) that displays the textbooks where the **TEXTBOOK.Tx_publisher** column contains a not null value excludes the rows associated with the titles Economics For Managers and Fundamentals or' SQL, since the **TEXTBOOK.Tx_publisher** column for each of these textbooks contains a null value. The row associated with the title Data Modeling is not excluded since the value in the **TEXTBOOK.Tx publisher** column for this title is not null (i.e., consists of a single blank space).

Example 1.5.2

```
SELECT TEXTBOOK.TX_TITLE, TEXTBOOK.TX_PUBLISHER
FROM TEXTBOOK
WHERE TEXTBOOK.TX_PUBLISHER IS NOT NULL;
```

Result:

```
Tx_title                Tx_publisher
----------------------- ------------
Database Management     Thomson
Linear Programming      Prentice-Hall
Simulation Modeling     Springer
```

[12] In SQL-92, a character value that is zero characters long is treated as a null value.

[13] Most versions of SQL contain a system variable that allows for the number of rows retrieved by the execution of a query to be displayed. The value of this variable will be shown in conjunction with various examples when appropriate. It is useful here as a way of showing that all rows from the TEXTBOOK table were selected.

```
Systems Analysis        Thomson
Principles of IS        Prentice-Hall
Programming in C++      Thomson
Data Modeling

7 rows selected.
```

The only comparison operators that can be used with null values are IS NULL and IS NOT NULL. If any other operator (e.g., =, >, <>, etc.) is used with a null value, the result is always unknown.[14] In addition, since a NULL represents a lack of data, a null value cannot be equal or unequal to any other value, even another NULL. Examples 1.5.3, 1.5.4, and 1.5.5 illustrate the fact that only IS NULL and IS NOT NULL can be used as comparison operators with null values.

Example 1.5.3

```
SELECT *
FROM TEXTBOOK
WHERE TEXTBOOK.TX_PUBLISHER = NULL;
```


Result:

```
no rows selected
```

Example 1.5.4

```
SELECT *
FROM TEXTBOOK
WHERE TEXTBOOK.TX_PUBLISHER <> NULL;
```

Result:

```
no rows selected
```

While conditional expressions of the form "WHERE X = NULL" and "WHERE X <> NULL" are illegal, SQL does not generate a syntax error. This can create serious problems in cases where "WHERE X IS NULL" would otherwise cause one or more rows to be selected.

Example 1.5.5

```
SELECT *
FROM TEXTBOOK
WHERE TEXTBOOK.TX_PUBLISHER IS NULL;
```

Result:

```
Tx_isbn         Tx_title                 Tx_year Tx_publisher
--------------- ---------------------- ---------- -------------
0296748-99      Economics For Managers       2001
012-54765-32    Fundamentals of SQL          2004
```

The SELECT statement in Example 1.5.6 demonstrates that there are five distinct publishers in the TEXTBOOK table and reflects the fact that a null value in a column can be distinguished from a column that contains a single blank space. The SELECT statement in Example 1.5.7 (with the is not null condition) indicates that the publisher whose name

[14] SQL-92 treats conditions evaluating to unknown as FALSE.

consists of a single blank space can be distinguished from the publisher with a null value for its name.

Example 1.5.6

```
SELECT DISTINCT TEXTBOOK.TX_PUBLISHER
FROM TEXTBOOK;
```

Result:

```
Tx_publisher
-------------
Prentice-Hall
Springer
Thomson


5 rows selected.
```

Example 1.5.7


```
SELECT DISTINCT TEXTBOOK.TX_PUBLISHER
FROM TEXTBOOK
WHERE TEXTBOOK.TX_PUBLISHER IS NOT NULL;
```

Result:

```
Tx_publisher
-------------
Prentice-Hall
Springer
Thomson

4 rows selected.
```

Example 1.5.8 uses the COUNT function to count the number of rows in the TEXTBOOK table and returns the result in a single table with a single column. Recall that when COUNT(*) is used, SQL focuses on the presence of rows rather than values appearing in a column.

Example 1.5.8

```
SELECT COUNT(*)
FROM TEXTBOOK;
```

Result:

```
  COUNT(*)
----------
         9

1 row selected.
```

When the COUNT function refers to a column, it behaves like the other aggregate functions and ignores null values (see Section 11.2.1.4). Thus the query in Example 1.5.9 displays the number of rows in the TEXTBOOK table with something other than a null value in the **TEXTBOOK.Tx_publisher** column.

Example 1.5.9

```
SELECT  COUNT(TEXTBOOK.TX_PUBLISHER)
FROM TEXTBOOK;
```

Result:

```
COUNT(TEXTBOOK.Tx_publisher)

                             7

1 row selected.
```

Numeric functions, such as the COUNT function, are associated with columns of output whose width is equal to the number of characters required to display the name of the function plus its argument(s). Often, a column alias, such as the one used in Example 1.5.10, is used to provide a more descriptive column heading. Observe how the width of the column has been adjusted to display the entire column alias (i.e., heading).

Example 1.5.10


```
SELECT COUNT(TEXTBOOK.TX_PUBLISHER)  AS  "Number of Publishers"
FROM TEXTBOOK;
```

Result:

```
Number of Publishers

                    7
```

In Example 1.5.10, the two rows in the TEXTBOOK table with null publishers are ignored by the COUNT function. This can be verified by the query in Example 1.5.11 which counts the number of distinct not null values in the TEXTBOOK.Tx_publisher column.

Example 1.5.11

```
SELECT COUNT(DISTINCT TEXTBOOK.TX_PUBLISHER)  AS  "Number of Distinct
Publishers"
FROM TEXTBOOK;
```

Result:

```
Number of Distinct  Publishers

                             4

1 row selected.
```

Aggregate functions are frequently applied to groups of rows in a table rather than to all rows in a table. The SELECT statement in Example 1.5.12 reflects the fact that there are five distinct publishers in the TEXTBOOK table and counts the number of rows (i.e., the COUNT (*) function is focusing on the presence of a row) associated with each distinct publisher.

Example 1.5.12

```
SELECT TEXTBOOK.TX_PUBLISHER,  COUNT(*)
FROM TEXTBOOK
GROUP  BY  TEXTBOOK.TX_PUBLISHER;
```

Result:

```
Tx_publisher    COUNT(*)
------------- ----------
                       1
Prentice-Hall          2
Springer               1
Thomson                3
                       2

5 rows selected.
```

Observe that the last row displayed is associated with the two textbooks with a null value in the TEXTBOOK.Tx_publisher column.

The SELECT statement in Example 1.5.13 also recognizes that there are five distinct publishers, but since group functions ignore the presence of null values in the TEXTBOOK.Tx_publisher column, the accumulator set up for the publisher whose value is a null value never has its initial value of zero incremented.

Example 1.5.13

```
SQL> SELECT TEXTBOOK.TX_PUBLISHER, COUNT(TEXTBOOK.TX_PUBLISHER)
FROM TEXTBOOK
GROUP BY TEXTBOOK.TX_PUBLISHER;
```

Result:

```
Tx_publisher  COUNT(TEXTBOOK.Tx_publisher)
------------- ---------------------------
                                         1
Prentice-Hall                            2
Springer                                 1
Thomson                                  3
                                         0

5 rows selected.
```

The SELECT statement in Example 1.5.14 works as expected, since the WHERE clause places the condition that the value in the TEXTBOOK.Tx_publisher column must be not null before that publisher can be used to form a group. The SELECT statement in Example 1.5.15 also works as expected since the WHERE clause focuses only on the publisher whose name consists of a single blank space. Finally, the SELECT statement in Example 1.5.16 serves as a reminder that the only publishers whose name consists of something other than a single blank space are Thomson, Prentice-Hall, and Springer. Recall, using a <> comparison operator in the evaluation of whether a null value differs from a single blank space produces an unknown result (which is treated as false).

Example 1.5.14

```
SELECT TEXTBOOK.TX_PUBLISHER, COUNT(*)
FROM TEXTBOOK
WHERE TEXTBOOK.TX_PUBLISHER IS NOT NULL
GROUP BY TEXTBOOK.TX_PUBLISHER;
```

Result:

```
Tx_publisher    COUNT(*)
------------ ----------
                      1
Prentice-Hall         2
Springer              1
Thomson               3

4 rows selected.
```

Example 1.5.15

```
SELECT TEXTBOOK.TX_PUBLISHER, COUNT(*)
FROM TEXTBOOK
WHERE TEXTBOOK.TX_PUBLISHER = ' '
GROUP BY TEXTBOOK.TX_PUBLISHER;
```

Result:

```
Tx_publisher    COUNT(*)
------------ ----------
                      1

1 row selected.
```


Example 1.5.16

```
SELECT TEXTBOOK.TX_PUBLISHER, COUNT(*)
FROM TEXTBOOK
WHERE TEXTBOOK.TX_PUBLISHER <> ' '
GROUP BY TEXTBOOK.TX_PUBLISHER;
```

Result:

```
Tx_publisher    COUNT(*)
------------ ----------
Prentice-Hall         2
Springer              1
Thomson               3

3 rows selected.
```

The SELECT statements in Examples 1.5.17, 1.5.18, and 1.5.19 illustrate the impact of a null value in queries that involve multiple conditions. In Example 1.5.17, the first condition selects the rows associated with the four not null publishers, while the second condition selects the rows associated with the three publishers Thomson, Prentice-Hall, and Springer. Since OR is used to connect the two conditions, groups are formed and counts accumulated for the four not null publishers.

Example 1.5.17

```
SELECT TEXTBOOK.TX_PUBLISHER,  COUNT(*)
FROM TEXTBOOK
WHERE TEXTBOOK.TX_PUBLISHER IS NOT NULL
OR TEXTBOOK.TX_PUBLISHER <> ' '
GROUP BY TEXTBOOK.TX_PUBLISHER;
```

Result:

```
Tx_publisher    COUNT(*)
------------- ----------
                       1
Prentice-Hall          2
Springer               1
Thomson                3

4 rows selected.
```

In Example 1.5.18, since AND is used to connect the two conditions, groups are formed only for those publishers that satisfy both conditions (i.e., Thomson, Prentice-Hall, and Springer).

Example 1.5.18

```
SELECT TEXTBOOK.TX_PUBLISHER, COUNT(*)
FROM TEXTBOOK
WHERE TEXTBOOK. TX_PUBLISHER IS NOT NULL
AND TEXTBOOK.TX_PUBLISHER <> ' '
GROUP BY TEXTBOOK.TX_PUBLISHER;
```


Result:

```
Tx_publisher    COUNT(*)

Prentice-Hall          2
Springer               1
Thomson                3

3 rows selected.
```

The SELECT statement in Example 1.5.19 illustrates a situation where the first condition selects the rows associated with the three publishers other than the single-space publisher and the second condition selects rows associated with the TEXTBOOK.Tx_publisher column containing a null value. Since OR is used to connect the two conditions, groups are formed and counts accumulated for Thomson, Prentice-Hall, and Springer, and the rows associated with the TEXTBOOK.Tx_publisher column containing a null value.

Example 1.5.19

```
SELECT TEXTBOOK.TX_PUBLISHER, COUNT(*)
FROM TEXTBOOK
WHERE TEXTBOOK.TX_PUBLISHER <> ' '
OR TEXTBOOK.TX_PUBLISHER IS NULL
GROUP BY TEXTBOOK. TX_PUBLI SHER;
```

Result:

```
Tx_publisher    COUNT(*)

Prentice-Hall          2
Springer               1
Thomson                3
                       2

4 rows selected.
```

The SELECT statement in Example 1.5.20 introduces the use of the operator IN to test whether a value is contained within a set of values.

Example 1.5.20

```
SELECT *
FROM TEXTBOOK
WHERE TEXTBOOK.TX_PUBLISHER IN ('Thomson', 'Springer');
```

Result:

```
Tx_isbn       Tx_title                  Tx_year Tx_publisher
------------- ------------------------- ------- -------------
000-66574998  Database Management         1999 Thomson
001-55-435    Simulation Modeling         2001 Springer
118-99898-67  Systems Analysis            2000 Thomson
0296437-1118  Programming in C++          2002 Thomson

4 rows selected.
```

Prefaced with the word NOT, NOT IN tests for whether a value does not appear within a set of values. Example 1.5.21 illustrates how null values only satisfy a WHERE clause that uses IS NULL or IS NOT NULL since the titles Economics For Managers and Fundamentals of SQL do not appear in the results but Prentice-Hall and the single-space publisher do.

Example 1.5.21

```
SELECT * FROM TEXTBOOK
WHERE TEXTBOOK.TX_PUBLISHER NOT IN ('Thomson', 'Springer');
```

Result:

```
Tx_isbn       Tx_title                  Tx_year Tx_publisher
------------- ------------------------- ------- -------------
003-6679233   Linear Programming         1997 Prentice-Hall
77898-8769    Principles of IS           2002 Prentice-Hall
111-11111111  Data Modeling              2006

3 rows selected.
```

11.2.1.6 Pattern Matching in SQL

In the context of the TEXTBOOK table, pattern matching would be useful if we were trying to find all textbooks used in Madeira College with a specific word or phrase in their titles. Examples include (a) all textbooks with the word "Introduction" in the title, (b) all textbooks whose title includes the words "Information System" or "Accounting System," and (c) all textbooks with titles that include "Programming." SQL-92 supports pattern matching through the use of the LIKE operator in conjunction with the two wildcard characters—the percent character (%) and the underscore character (_).[15] The percent character represents *a series of one or more unspecified characters* **while the underscore character represents** *exactly one character.* **Let us continue to use the data associated with the TEXTBOOK table as a vehicle to explore use of the LIKE operator. Given the number of rows and columns associated with**

[15] Although the SQL-92 standard specifies use of the underscore character(_) and percent character (%), some implementations of SQL use other characters to represent a single character or a series of characters.

the TEXTBOOK table, the examples that follow have limited practical application but instead are intended to simply illustrate how the LIKE operator works.

The SELECT statement in Example 1.6.1 searches for all textbooks such that the first two characters of their ISBN begin with the digit 1 and is followed by any alphanumeric character. Observe that while the textbooks with the titles Systems Analysis and Data Modeling seem to satisfy this request, no rows are selected. Likewise, the SELECT statement in Example 1.6.2 searches for all textbooks whose title contains the letter "i" in the second character position. While two textbooks would appear to satisfy this request, no rows are selected.

Example 1.6.1

```
SELECT TEXTBOOK.TX_TITLE, TEXTBOOK.TX_YEAR
FROM TEXTBOOK
WHERE TEXTBOOK.TX_ISBN LIKE '1_';
```

Result:

```
No rows selected.
```


Example 1.6.2

```
SELECT TEXTBOOK.TX_TITLE
FROM TEXTBOOK
WHERE TEXTBOOK.TXJTITLE LIKE '_i';
```

Result:

```
No rows selected.
```

The SELECT statements in Examples 1.6.1 and 1.6.2 return no rows because no textbook has an ISBN number that is exactly two characters long nor is there any textbook title that has the letter "i" in the second character position and is exactly two characters long. As illustrated in Example 1.6.3, revising Example 1.6.2 by adding the % sign as a wildcard character searches for textbook titles with the letter "i" in the second character position but allows the title to have as many as twenty-two characters (the width of the **TEXTBOOK.Tx_title** column).

Example 1.6.3

```
SELECT TEXTBOOK.TX_TITLE
FROM TEXTBOOK
WHERE TEXTBOOK.TX_TITLE LIKE '_i%';
```

Result:

```
Tx_title
----------------------
Linear Programming
Simulation Modeling

2 rows selected.
```

Examples 1.6.4 and 1.6.5 illustrate that the LIKE Operator is case sensitive.

Example 1.6.4

```
SELECT TEXTBOOK.TX_TITLE
FROM TEXTBOOK
WHERE TEXTBOOK.TX_TITLE LIKE 'P%';
```

Result:

```
Tx_title
--------------------
Principles of IS
Programming in C++

2 rows selected.
```

Example 1.6.5

```
SELECT TEXTBOOK.TX_TITLE
FROM TEXTBOOK
WHERE TEXTBOOK.TX_TITLE LIKE 'p%';
```

Result:

```
No rows selected.
```


The SELECT statements in Examples 1.6.6 and 1.6.7 represent rather unusual uses of the LIKE Operator. Example 1.6.6 searches for the titles of all textbooks that contain the letter "e" while Example 1.6.7 displays the titles of all textbooks.

Example 1.6.6

```
SELECT TEXTBOOK.TX_TITLE
FROM TEXTBOOK
WHERE TEXTBOOK.TX_TITLE LIKE '%e%';
```

Result:

```
Tx_title
--------------------
Database Management
Linear Programming
Simulation Modeling
Systems Analysis
Principles of IS
Economics For Managers
Fundamentals of SQL
Data Modeling

8 rows selected.
```

Example 1.6.7

```
SELECT TEXTBOOK.TX_TITLE
FROM TEXTBOOK
WHERE TEXTBOOK.TX_TITLE LIKE '%';
```

Result:

```
Tx_title
--------------------
Database Management
Linear Programming
```

```
Simulation Modeling
Systems Analysis
Principles of IS
Economics For Managers
Programming in C++
Fundamentals of SQL
Data Modeling

9 rows selected.
```

As shown in the SELECT statement in Example 1.6.8, the string'%' cannot match a null value as the two publishers with a null value in the TEXTBOOK.Tx_publisher column do not appear in the results.

Example 1.6.8

```
SELECT *
FROM TEXTBOOK
WHERE TEXTBOOK.TX_PUBLISHER LIKE '%';
```


Result:

```
Tx_isbn        Tx_title                  Tx_year Tx_publisher
-------------- ---------------------- ---------- -------------
000-66574998   Database Management          1999 Thomson
003-6679233    Linear Programming          1997 Prentice-Hall
001-55-435     Simulation Modeling         2001 Springer
118-99898-67   Systems Analysis            2000 Thomson
77898-8769     Principles of IS            2002 Prentice-Hall
0296437-1118   Programming in C++          2002 Thomson
111-11111111   Data Modeling               2006

7 rows selected.
```

CHAR(f) and VARCIIAR(f) are string data types (see Table 10.1 in Chapter 10) commonly used to define fixed-length and variable-length character data. A CHAR(t) data type, where f represents the length of the column, has a maximum size of 255 characters in most DBMSs; blank characters are added to the data should the number of characters be less than (. A VARCHAR(f) data type, where *t* also represents the length of the column, has a maximum size of 2,000 characters in most DBMSs. A VARCHAR(C) data type does not append blank characters to the data if the number of characters is less than *I.* CHAR(0 and VARCHAR(f) data types can produce different results in some comparisons that involve the LIKE Operator. For example, the query in Example 1.6.9 searches for and locates all titles that end with the letter "s".

Example 1.6.9

```
SELECT *
FROM TEXTBOOK
WHERE TEXTBOOK.TX_TITLE LIKE '%s';
```

Result:

```
Tx_isbn        Tx_title                  Tx_year Tx_publisher
-------------- ---------------------- ---------- -------------
118-99898-67   Systems Analysis            2000 Thomson
0296748-99     Economics For Managers      2001

2 rows selected.
```

However, had the **TEXTBOOK.Tx_title** column been defined as a CHAR(22) data type instead of as a VARCHAR(22) data type, only one of the titles, Economics For Managers (a title with an "s" in the 22nd character position) would have been located since the 22nd character position in the title, Systems Analysis, would have contained a single blank space character (i.e., the rightmost "s" in Systems Analysis would have been in the 16th character position with character positions 17-22 containing blank spaces).

SQL allows for the definition of an escape character in cases where either a percent character (%) or an underscore character (_) stand for themselves as part of the search. For example, suppose you need to identify the name of each table in the Madeira College database with an underscore character as part of its table name. This is possible with the following SELECT statement:

Example 1.6.10

```
SELECT TABLE_NAME
FROM USER_TABLES[16]
WHERE TABLE_NAME LIKE '%/_%'
ESCAPE '/';
```


Result:

```
Table_name

GRAD_STUDENT
```

ESCAPE is used here to declare the slash (/) as an escape character so that it can be prefixed to the underscore character in the character string expression used in the LIKE operator.

1122 SQL Queries Based on Binary Operators

Recall that binary operators "operate" on two relations and are of four types: (a) the Cartesian Product operator, (b) set theoretic operators, (c) join operators, and (d) the divide operator. The examples in this section arc based on the Madeira College tables shown in Figure 11.1.

11221 The Cartesian Product Operation

An SQL SELECT statement that references two or more tables and does not include a WHERE clause always involves a Cartesian Product operation. The Cartesian Product operation by itself is generally meaningless. However, all joins actually begin as a Cartesian Product followed by a WHERE condition that selects only the rows that make sense in the context of the question. In other words, a Cartesian Product is useful when followed first by a Selection operation that matches values of attributes coming from the component relations (technically, a Cartesian Product operation followed by a Selection operation is a Join operation) and then by a Projection operation that selects certain columns from the combined result.

[16] USER_TABLES is the name of a data dictionary table comprised of columns that contain data about the various tables that comprise the Madeira College database. Included in this table is a column that records the name of each table.

Example 2.1.1. What is the product of the T and SS relations? *Note:* **In order to illustrate some of the SQL queries based on binary operators, relations SS and T have been created from the larger SECTION and TAKES relations. Relation SS (introduced earlier in conjunction with the Right Outer Join example in Section 11.1.2.3) contains four attributes: Ss_section#, Ss_qtr, Ss_year, and Ss_c_course# and relation T contains T_ss_c_course#, T_grade, and T_s_sid.**

SQL SELECT Statement:

```
SELECT * FROM T CROSS JOIN SS;
```

In the SQL-92 standard, the CROSS keyword, combined with the JOIN keyword, is used in the FROM clause to create a Cartesian Product. Sometimes a Cartesian Product is referred to as a Cross Join. The first 22 rows of the result of this Cartesian Product appear below. Since there are 11 rows in SS and 11 rows in T, a total of 121 rows are produced as a result of the concatenation of SS and T. The result shown constitutes the concatenation of the first row of SS (observe how the final four columns remain the same) with the 11 rows in T. Observe that the next 11 rows of the result shows second row of SS with each of the 11 rows of T.

Result:

T_se_c_course#	T_grade	T_s_sid	Ss_section#	Ss_qtr	Ss_year	Ss_c_course#
22QA375	A	KP78924	101	A	2007	22QA375
22QA375	A	KS39874	101	A	2007	22QA375
22QA375	B	BG66765	101	A	2007	22QA375
22IS330	C	BE76598	101	A	2007	22QA375
22IS330	B	KJ56656	101	A	2007	22QA375
22IS330	A	KP78924	101	A	2007	22QA375
22IS330	A	KS39874	101	A	2007	22QA375
22IS832	A	KS39874	101	A	2007	22QA375
22IS330	A	BE76598	101	A	2007	22QA375
22IS832	B	BG66765	101	A	2007	22QA375
22IS330	C	GS76775	101	A	2007	22QA375
22QA375	A	KP78924	901	A	2006	22IS270
22QA375	A	KS39874	901	A	2006	22IS270
22QA375	B	BG66765	901	A	2006	22IS270
22IS330	C	BE76598	901	A	2006	22IS270
22IS330	B	KJ56656	901	A	2006	22IS270
22IS330	A	KP78924	901	A	2006	22IS270
22IS330	A	KS39874	901	A	2006	22IS270
22IS832	A	KS39874	901	A	2006	22IS270
22IS330	A	BE76598	901	A	2006	22IS270
22IS832	B	BG66765	901	A	2006	22IS270
22IS330	C	GS76775	901	A	2006	22IS270

When a Cartesian Product operation is accompanied by a Selection operation, an Inner Join results. As shown below, using the SQL-92 standard, the words INNER JOIN[17] are used in the FROM clause instead of the words CROSS JOIN, and the ON clause is used to specify the join condition(s).

[17] Use of the word, INNER, is optional.

Example 2.1.2. What are the student IDs of those students who took a section of a course during the year 2006? A'ote: Since the result here explicitly calls for only one column to be displayed, the SECTION and TAKES tables are used in lieu of the abridged versions SS and T used in Example 2.1.1.

SQL SELECT Statement:

```
SELECT TAKES.TK_ST_SID
FROM TAKES INNER JOIN SECTION
ON SECTION.SE_YEAR = 2006
AND TAKES.TK_SE_YEAR = 2006
AND SECTION.SE_SECTION# = TAKES.TK_SE_SECTION#;
```

Result:

```
Tk_st_sid

BE76598

1 row selected.
```


Observe that the ON clause given above effectively constrains the concatenation of a row from TAKES with a row from SECTION to the condition where both the year in TAKES (TAKES.Tk_se_year) and the year in SECTION (SECTION.Se_year) are equal to 2006, and the section* in the SECTION row (SECTION.Se_section#) matches the section* in the TAKES row (TAKES.Tk_se_section#).

Reviewing the content of the SECTION and TAKES tables provides an explanation of this result. Since there are 11 rows in the SECTION table and 11 rows in the TAKES tables, the Cartesian Product operation results in an unnamed table with 121 rows and 14 columns (there are eight columns in SECTION and six columns in TAKES). Since only one row in TAKES contains the year 2006, this Cartesian Product operation generates only four of these 121 rows which have 2006 in both the TAKES.Tk se year and SECT1ON.Se_year columns. Each of these four rows contains the same value in the TAKES.Tk_se_section# column, 101, and the same value in the TAKES.Tk st sid column, BE76598. On the other hand, only one of these four rows contains the section number 101 in the SECTION.Selection* column. Hence, the portion of the Selection operation which requires that TAKES.Tk_se_section# = SECTION.Se section* "selects" only one of these four rows, and the SQL Projection operation displays the TAKES.Tk_st_sid value BE76598.[ks]

11.2.2.2 SQL Queries Involving Set Theoretic Operations

Union, Intersection, and Difference are three set theoretic operators used to merge elements of two union compatible sets together. The examples shown below illustrate the incorporation of these operators in SQL statements that involve the tables R and S

[ks] The actual execution of this SELECT statement would differ from this description. For example, a Selection operation on the TAKES table would occur first and result in "selecting" the one row where TAKES.Tk_se_year = 2006. A Cartesian Product operation would follow concatenating this one row with the 11 rows in the SECTION table. This would be followed by a second Select operation which would "select" the four rows in the SECTION" tabic where SECTION.Se_year = 2006 and where TAKES.Tk se_section# = SECTION.Se_section#. The Projection operation on the TAKES.Tk_st_sid column and the imposition of the DISTINCT qualifier is the final operation executed.

derived from the SECTION table, where R contains those sections offered during a Fall Quarter and S contains those sections offered in a room located in Lindner Hall.[19]

RELATION R

Se_section#	Se_qtr	Se_year	Se_time	Se_maxst	Se_room	Se_co_course#	Se_pr_profid
902	A	2006	H1700	25	Lindner 108	22IS270	SK85977
901	A	2006	W1800	35	Rhodes 611	22IS270	SK85977
101	A	2007	H1700	29	Lindner 108	22IS330	SK85977
101	A	2007	T1015	25		22QA375	HT54347
101	A	2007	W1800		Baldwin 437	20ECES212	RR79345

RELATION S

Se_section#	Se_qtr	Se_year	Se_time	Se_maxst	Se_room	Se_co_course#	Se_pr_profid
902	A	2006	H1700	25	Lindner 108	22IS270	SK85977
101	S	2006	T1045	29	Lindner 110	22IS330	SK85977
102	S	2006	H1045	29	Lindner 110	22IS330	CC49234
101	A	2007	H1700	29	Lindner 108	22IS330	SK85977

Example 2.2.1 (Corresponds to Union Example in Section 11.1.2.2). Display the union of R and S (i.e., those sections offered in either the fall quarter or in a room located in Lindner Hall, or offered in both the fall quarter and in a room located in Lindner Hall).

SQL SELECT Statement:

```
SELECT *
FROM R
   UNION
SELECT *
FROM S;
```

Note that when duplicate rows exist in tables, only one row per set of duplicates is displayed in the result when using the UNION operator unless UNION ALL has been used.

Result:

Se_section#	Se_qtr	Se_year	Se_time	Se_maxst	Se_room	Se_co_course#	Se_pr_profid
101	A	2007	H1700	29	Lindner 108	22IS330	SK85977
101	A	2007	T1015	25		22QA375	HT54347
101	A	2007	W1800		Baldwin 437	20ECES212	RR79345
101	S	2006	T1045	29	Lindner 110	22IS330	SK85977
102	S	2006	H1045	29	Lindner 110	22IS330	CC49234
901	A	2006	W1800	35	Rhodes 611	22IS270	SK85977
902	A	2006	H1700	25	Lindner 108	22IS270	SK85977

Example 2.2.2 (Corresponds to Intersection Example in Section 11.1.2.2). Display the intersection of R and S (i.e., those sections offered in both the fall quarter and in a room located in Lindner Hall).

SQL SELECT Statement:

```
SELECT *
FROM R
```

The tables R and S were created simply as a means to illustrate use of the UNION, INTERSECT and DIFFERENCE operators in an SQL query. Each of the examples in this section can be expressed by a query that refers to just the SECTION table.

```
   INTERSECT
SELECT *
FROM S;
```

Result:

Se_section#	Se_qtr	Se_year	Se_time	Se_maxst	Se_room	Se_co_course#	Se_pr_profid
101	A	2007	H1700	29	Lindner 108	22IS330	SK85977
902	A	2006	H1700	25	Lindner 108	22IS270	SK85977

Example 2.2.3 (Corresponds to Difference Example 1 in Section 11.1.2.2). Display the difference R minus S (i.e., those sections offered in the fall quarter but not in a room located in Lindner Hall).

SQL SELECT Statement:

```
SELECT *
FROM R
   MINUS
SELECT *
FROM S;
```

Result:

11).

Se_section#	Se_qtr	Se_year	Se_time	Se_maxst	Se_room	Se_co_course#	Se_pr_profid
101	A	2007	T1015	25		22QA375	HT54347
101	A	2007	W1800		Baldwin 437	20ECES212	RR79345
901	A	2006	W1800	35	Rhodes 611	22IS270	SK85977

The word MINUS is used in Oracle's SQL. The word EXCEPT is part of the SQL-92 standard.

Example 2.2.4 (Corresponds to Difference Example 2 in Section 11.1.2.2). Using the data in the relations R and S given above, form the difference S minus R (i.e., those sections offered in a room located in Lindner Hall but not in the fall quarter).

SQL SELECT Statement:

```
SELECT *
FROM S
   MINUS
SELECT *
FROM R;
```

Result:

Se_section#	Se_qtr	Se_year	Se_time	Se_maxst	Se_room	Se_co_course#	Se_pr_profid
101	S	2006	T1045	29	Lindner 110	22IS330	SK85977
102	S	2006	H1045	29	Lindner 110	22IS330	CC49234

11.2.2.3 Join Operations

Four types of relational algebra joins were discussed earlier in Section 11.1.2.3. The Equijoin involves join conditions with equality comparisons only and produces a new relation which contains all columns associated with the join condition. On the other hand, a Natural Join omits one of each pair of columns associated with the join condition, thus eliminating the possibility of obtaining a result that contains columns with the same set of values. A Theta Join, the third type of join, is based on a join condition that does not involve an equality comparison.

A fourth type of join is the Outer Join. An Outer Join is used when it is desired to include each row in a relation in the result even if the row does not contain an attribute value satisfying the join condition. Examples of these four types of joins using SQL are illustrated here in the context of the SECTION and TAKES tables.

In the SECTION table, a Section* is assigned to each course offered during a particular quarter and year. This allows Section* 101 to be assigned to the course 22QA375 offered during the Fall quarter in 2007. It is also possible for Section* 101 to be assigned to other courses offered during the Fall quarter in 2007 (e.g., 20ECES212 and 22IS330). Observe that **(Se_section#, Se_qtr, Se_year,** and **Se_co_course#)** constitute the concatenated primary key of SECTION.

The TAKES table records the sections of the courses taken by various students. Observe that the concatenated primary key of TAKES is **(Tk_se_section#, Tk_se_qtr, Tk_se_year, Tk_se_co_course#.** and **Tk_st_sid),** which makes it possible for a student to take two different courses with the same Section* during a particular quarter and year (e.g., **Tk_st_sid** KS39874 takes both 22QA375 and 22IS330 during the fall quarter in 2007).

While it is possible to join TAKES and SECTION over the two common attributes **SECTION.Se_section#** and **TAKES.Tk_se_section#** sharing the same domain, the result of such a join would be meaningless. This is because each row in TAKES would be concatenated not only with the row from SECTION that describes the section of the course taken by the student, but also with the rows from SECTION with the same **SECTION.Se_section#** but associated with a different course than that taken by the student. Using the data in the SECTION and TAKES tables in Figure 11.1, it is left as an exercise for the reader to demonstrate that an Equijoin of TAKES and SECTION on just the common attributes **SECTION.Se_section#** and **TAKES.Tk_se_section#** yields a result that contains 65 rows.

An Example of an Equijoin Operation The most meaningful Equijoin of TAKES and SECTION would involve all of the attributes that share the same domains in TAKES and SECTION. This allows the result of the join to include only rows that match information on a section taken by a student (recorded in TAKES) with the information about that section (recorded in SECTION).

Example **2.3.1.1.** Join the SECTION and TAKES tables over their common attributes.

SQL SELECT Statement:

```
SELECT *
FROM SECTION JOIN TAKES
ON SECTION.SE_SECTION# = TAKES.TK_SE_SECTION#
AND SECTION.SE_QTR = TAKES.TK_SE„QTR
AND SECTION.SE_YEAR = TAKES.TK_SE_YEAR
AND SECTION.SE_CO_COURSE# = TAKES.TK_SE_CO_COURSE#;
```

Had the result of this Equijoin been displayed for each section taken by a student, a complete description of the section would have appeared as well. In addition, the section number taken, the quarter during which the section taken was offered, the year during which the section taken was offered, and the time the section taken was offered would have appeared twice in each row as a result of the Equijoin operation. The following example illustrates how the combination of a Natural Join operation and a Projection operation allow a less cluttered result to be displayed.

Examples of Natural Join Operations **Since the result of an Equijoin results in pairs of attributes with identical values, a Natural Join operation was created to omit the second (and superfluous) attribute(s) in an Equijoin condition. In direct violation of the requirement of the relational data model that attributes have unique names over the entire relational schema, the standard definition of NATURAL JOIN requires that the two join attributes (or each pair of join attributes) have the same name in both relations.**

Example 2.3.2.1. Join the SECTION and TAKES tables over their common attributes (in our case attributes with different names but sharing the same domain).

SQL-92 supports three approaches for representing a Natural Join. One uses the NATURAL JOIN keyword, another uses JOIN ... USING, and the third uses JOIN ... ON. The first two approaches can only be used if the requirement that attributes have unique names over the entire relational schema is not enforced and tables that contain columns with the same name are allowed to exist. Thus in the context of this example, use of the first two approaches would require that in addition to the section#, quarter, year, and course# sharing the same domain, they must also have exactly the same column names in the SECTION and TAKES tables. Since this book adheres strictly to the requirement that attribute (and hence column) names be unique over the entire relational schema, the first two approaches can only be illustrated in general. In addition, in order to display just the distinct column names, the third approach based on the SQL-92 syntax requires that each column name appear in the <column list>.

Approach 1

```
SELECT * FROM tablename1 NATURAL JOIN tablename2
```

Approach 2

```
SELECT * FROM tablename1

JOIN tablename2 USING (columnname_a, columnname_b, ..., columnnamej]
```

Approach 3

```
SELECT SECTION.*, TAKES.TK_GRADE, TAKES.TK_ST_SID
FROM SECTION JOIN TAKES ON
SECTION.SE_SECTION# = TAKES.TK_SE_SECTION#
AND SECTION.SE_QTR = TAKES.TK_SE_QTR
AND SECTION.SE_YEAR = TAKES.TK_SE_YEAR
AND SECTION.SE_CO_COURSE# = TAKES.TK_SE_CO_COURSE#;
```

Result:

Se_section#	Se_qtr	Se_year	Se_time	Se_maxst	Se_room	Se_co_course#	Se_pr_profid	Tk_grade	Tk_st_sid
101	A	2007	T1015	25		22QA375	HT54347	A	KP78924
101	A	2007	T1015	25		22QA375	HT54347	A	KS39874
101	A	2007	T1015	25		22QA375	HT54347	B	BQ66765
101	S	2006	T1045	29	Lindner 110	22IS330	SK85977	C	BE76598
101	A	2007	H1700	29	Lindner 108	22IS330	SK85977	B	KJ56656
101	A	2007	H1700	29	Lindner 108	22IS330	SK85977	A	KP78924
101	A	2007	H1700	29	Lindner 108	22IS330	SK85977	A	KS39874
701	W	2007	M1000	33	Braunstien 211	22IS832	CC49234	A	KS39874
101	A	2007	H1700	29	Lindner 108	22IS330	SK85977	A	BE76598
701	W	2007	M1000	33	Braunstien 211	22IS832	CC49234	B	BQ66765
101	A	2007	H1700	29	Lindner 108	22IS330	SK85977	C	GS76775

Since most queries that involve one or more Join operations also include some sort of Projection operation, in effect virtually all joins take the form of this "variation" of a Natural Join.

Sometimes a join may be specified between a relation and itself. This type of join is often referred to as a Self Join.

Example 2.3.2.2. List the student IDs of those students recorded as having taken more than one course.

```
SELECT  X.TK_ST_SID
FROM  TAKES  X  JOIN  TAKES  Y
ON  X.TK_ST_SID  =  Y.TK_ST_SID
AND  X.TK_SE_CO_COURSE#  <>  Y.TK_SE_CO_COURSE#;
```

In this SELECT statement the TAKES table is referenced twice. To prevent ambiguity, each use of TAKES in the FROM clause has been assigned a temporary name (called a table alias). X and Y serve as the table aliases in this SELECT statement. Whenever a table alias is associated with a table name in the FROM clause, it must also be used any time the table is referenced in the SELECT statement (i.e., in this example when referencing the table in the WHERE condition and also in the column list that follows use of the word SELECT). Whenever a table alias has been introduced in the FROM clause, we cannot use the full table name anywhere else in the SELECT statement.

Since, as we have seen, duplicate rows are not automatically removed in SQL, the execution of the SELECT statement given above generates ten rows (six with an **X.Tk_st_sid** value of KS39874, two with an **X.Tk_st_sid** value of KP78924, and two with an **X.Tk_st_sid** value of BG66765):

```
Tk_st_sid

BG66765
BG66765
KP78924
KP78924
KS39874
KS39874
KS39874
KS39874
KS39874
KS39874
```

This is due to the fact that the join condition is satisfied two times for each of the three rows in TAKES that involve **X.Tk_st_sid** KS39874, one time for each of the two rows in TAKES that involve **X.Tk_st_sid** KP78924, and one time for each of the two rows in TAKES that involve **X.Tk_st_sid** BG66765. Observe that while **X.Tk_st_sid** BE76598 appears in two rows in TAKES, both rows involve the same course (22IS330) taken once in the summer quarter of the year 2006 and again in the fall quarter of the year 2007. The fact that the join condition in the relational algebra expression and in the SQL SELECT statement is not satisfied is the cause of **X.Tk_st_sid** BE76598 not being displayed.

Use of the qualifier DISTINCT in the SELECT statement (see below) eliminates the duplicate rows and produces a result that corresponds to the relational algebra result.

SQL SELECT Statement Revised:

```
SELECT DISTINCT X.TK_ST_SID
FROM TAKES X JOIN TAKES Y
ON X.TK_ST_SID = Y.TK_ST_SID
AND X.TK_SE_CO_COURSE# <> Y.TK_SE_CO_COURSE#;
```

Result:

```
Tk st sid

BG66765
KP78924
KS39874
```

The Natural Join or equijoin operation can also be specified among multiple tables, leading to what is sometimes referred to as an *n-uny join.*

Example 2.3.2.3. Instead of listing the student IDs of those students having taken more than one course, list the names of those students having taken more than one course.

SQL SELECT Statement:


```
SELECT DISTINCT STUDENT.ST_NAME
FROM (TAKES X JOIN TAKES Y
ON X.TK_ST_SID = Y.TK_ST_SID
AND X.TK_SE_CO_COURSE# <> Y.TK_SE_CO_COURSE#)
JOIN STUDENT ON X.TK_ST_SID = STUDENT.ST_SID;
```

This SELECT statement uses the result of the Self Join from Example 2.3.2.2 and joins it with the STUDENT table. Without the use of the qualifier DISTINCT to eliminate duplicate rows, the three names shown below would be displayed a total of ten times (i.e., Gladis Bale twice, Poppy Kramer twice, and Sweety Kramer six times).

Result:

```
St name

Gladis Bale
Poppy Kramer
Sweety Kramer
```

Example 2.3.2.4. For each student taking a course in the fall quarter of 2007, list the student's name, classroom where the course is offered, and course number.

SQL SELECT Statement:

```
SELECT STUDENT.ST_NAME, SECTION.SE_ROOM, TAKES.TK_SE_CO_COURSE#
FROM (STUDENT JOIN TAKES
ON STUDENT.ST_SID = TAKES.TK_ST_SID)
JOIN SECTION ON TAKES.TK_SE_CO_COURSE# = SECTION.SE_CO_COURSE#
AND SECTION.SE_QTR = TAKES.TK_SE_QTR
AND SECTION.SE_YEAR = TAKES.TK_SE_YEAR
AND TAKES.TK_SE_QTR = 'A' AND TAKES.TK_SE_YEAR = 2007;
```

Result:

```
St_name         Se_room         Tk_se_co_course#
--------------- --------------- ---------------
Poppy Kramer                    22QA375
Sweety Kramer                   22QA375
Gladis Bale                     22QA375
```

```
Joumana Kidd    Lindner 108    22IS330
Poppy Kramer    Lindner 108    22IS330
Sweety Kramer   Lindner 108    22IS330
Elijah Baley    Lindner 108    22IS330
Shweta Gupta    Lindner 108    22IS330
```

In effect, prior to the execution of the Projection operation, the SQL SELECT statement first links (i.e., concatenates) each row of TAKES to the corresponding row in SECTION and then links each row in the combined result to the corresponding row in STUDENT. Should both the course number and course name need to be displayed, the COURSE table must be included in the join. As shown below, the SELECT statement required to generate this result takes the result of joining the STUDENT, TAKES, and SECTION tables and joins it with the COURSE table.

SQL SELECT Statement:

```
SELECT STUDENT.ST_NAME, SECTION.SE_ROOM, TAKES.TK_SE_CO_COURSE#, COURSE.CO_
NAME
FROM  ((STUDENT JOIN TAKES
ON STUDENT.ST_SID = TAKES.TK_ST_SID)
JOIN SECTION ON TAKES.TK_SE_CO_COURSE# = SECTION.SE_CO_COURSE#
AND SECTION.SE_QTR = TAKES.TK_SE_QTR
AND SECTION.SE_YEAR = TAKES.TK_SE_YEAR
AND TAKES. TK_SE_QTR = 'A' AND TAKES. TK_SE_YEAR = 2007)
JOIN COURSE ON SECTION.SE_CO_COURSE# = COURSE.CO_COURSE#;
```

Result:

```
St_name         Se_room         Tk_se_co_course# Co_name
--------------- --------------- ---------------- ---------------------
Poppy Kramer                    22QA375          Operations Research
Sweety Kramer                   22QA375          Operations Research
Gladis Bale                     22QA375          Operations Research
Joumana Kidd    Lindner 108     22IS330          Database Concepts
Poppy Kramer    Lindner 108     22IS330          Database Concepts
Sweety Kramer   Lindner 108     22IS330          Database Concepts
Elijah Baley    Lindner 108     22IS330          Database Concepts
Shweta Gupta    Lindner 108     22IS330          Database Concepts
```

The Theta Join Operation **While occurring infrequently in practical applications, Theta Joins (or Non-Equijoins) that do not involve equality conditions are possible as long as the join condition involves attributes that share the same domain.**

Example 2.3.3.1. Instead of doing an Equijoin of SECTION and TAKES when an equality condition involving each of their common attributes exists, do a Join of SECTION and TAKES when an inequality condition exists for each of their common attributes.

The following SQL SELECT statement displays the columns in SECTION and TAKES involved in the join condition. On each row, observe how adjacent pairs of columns reflect inequalities for the four columns involved in the join condition.

SQL SELECT Statement:

```
SELECT SECTION.SE_SECTION#, TAKES.TK_SE_SECTION#,
       SECTION . SE_YEAR, TAKES . TK_SE_YEAR,
       SECTION,SE_QTR, TAKES.TK_SE_QTR,
       SECTION.SE_CO_COURSE#, TAKES.TK_SE_CO_COURSE#
FROM SECTION JOIN TAKES
ON SECTION. SE_SECTION# <> TAKES . TK_SE_SECTION#
```

```
AND SECTION.SE_YEAR <> TAKES.TK_SE_YEAR
AND SECTION.SE_QTR <> TAKES.TK_SE_QTR
AND SECTION.SE_CO_COURSE# <> TAKES.TK_SE_CO_COURSE#
ORDER BY SECTION.SE_SECTION#;
```

Result:

Se_section#	Tk_se_section#	Se_year	Tk_se_year	Se_qtr	Tk_se_qtr	Se_co_course#	Tk_se_co_course#
101	701	2006	2007	S	W	22IS330	22IS832
101	701	2006	2007	S	W	22IS330	22IS832
102	101	2006	2007	S	A	22IS330	22QA375
102	101	2006	2007	S	A	22IS330	22QA375
102	101	2006	2007	S	A	22IS330	22QA375
102	701	2006	2007	S	W	22IS330	22IS832
102	701	2006	2007	S	W	22IS330	22IS832
701	101	2007	2006	W	S	22IS832	22IS330
901	701	2006	2007	A	W	22IS270	22IS832
901	701	2006	2007	A	W	22IS270	22IS832
902	701	2006	2007	A	W	22IS270	22IS832
902	701	2006	2007	A	W	22IS270	22IS832

11.2.2.4 Outer Join Operations

In relational algebra, Outer Joins can be used when there is a need to keep all the tuples in R, or those in S, or those in both relations in the result of the Join, whether or not they have matching tuples in the other relation.

Left Outer Join **The Left Outer Join operation is used when we want to retain all rows in the leftmost table (i.e., the first table to be listed) regardless of whether corresponding rows exist in the other table. SQL uses the LEFT OUTER JOIN[20] keywords to create a Left Outer Join.**

Example 2.4.1.1. For each section taken by a student, display the section number, quarter, year, course number, grade, student ID, and student name. Include the names of all students (i.e., even those who have never taken a course).

SQL SELECT Statement:

```
SELECT TAKES.*, ST_NAME
FROM STUDENT LEFT OUTER JOIN TAKES
ON STUDENT.ST_SID = TAKES.TK_ST_SID;
```

Result:

Tk_se_section#	Tk_se_qtr	Tk_se_year	Tk_se_co_course#	Tk_grade	Tk_st_sid	St_name
101	A	2007	22QA375	A	KP78924	Poppy Kramer
101	A	2007	22QA375	A	KS39874	Sweety Kramer
101	A	2007	22QA375	B	BG66765	Gladis Bale
101	S	2006	22IS330	C	BE76598	Elijah Baley
101	A	2007	22IS330	B	KJ56656	Joumana Kidd
101	A	2007	22IS330	A	KP78924	Poppy Kramer

[20] Use of the word OUTER is not required to create either a LEFT, RIGHT, or FULL outer join.

```
101 A    2007 22IS330    A    KS39874    Sweety Kramer
701 W    2007 22IS832    A    KS39874    Sweety Kramer
101 A    2007 22IS330    A    BE76598    Elijah Baley
701 W    2007 22IS832    B    BG66765    Gladis Bale
101 A    2007 22IS330    C    GS76775    Shweta Gupta
                                         Tim Duncan
                                         Troy Hudson
                                         Rick Fox
                                         Jenny Aniston
                                         Daniel Olive
                                         Diana Jackson
                                         Jenna Hopp
                                         Wanda Seldon
                                         David Sane
                                         Vanessa Fox
```

In SQL-92, the words LEFT OUTER JOIN are used to designate a Left Outer Join operation. The use of the LEFT JOIN keywords means that if the table listed on the left side of the join condition given in the ON clause has an unmatched row, it should be matched with a null row and displayed in the results.

Observe how the first three rows in TAKES are concatenated with those rows in STUDENT that correspond to the students taking the course 22QA375 to produce the first three rows in the result shown above. On the other hand, the second row in STUDENT (the one for Daniel Olive) has been concatenated with the row of null values appended to the end of the TAKES table to produce the 16th row in the result. The result reveals that ten of the students have never taken a course.

One way to verify that the ten rows with null values for the attributes in TAKES represent the ten students who have not taken a course is to take the difference between the STUDENT and the TAKES relations over those attributes that share the same domain.

```
SELECT STUDENT.ST_SID
FROM STUDENT
  MINUS
SELECT TAKES.TK_ST_SID
FROM TAKES;
```

Result:

```
St_sid

AJ76998
DT87656
FR45545
FV67733
HJ45633
HT67657
JD35477
OD76578
SD23556
SW56547
```

The student IDs shown above can be replaced by a list of student names through the use of the following nested subquery. Subqueries are discussed in Section 11.2.3.

```
SELECT STUDENT.ST_NAME
FROM STUDENT
WHERE STUDENT.ST_SID IN
    (SELECT STUDENT.ST_SID
    FROM STUDENT
       MINUS
    SELECT TAKES.TK_ST_SID
    FROM TAKES);
```

Result:

```
St_name

Jenny Aniston
Tim Duncan
Rick Fox
Vanessa Fox
Jenna Hopp
Troy Hudson
Diana Jackson
Daniel Olive
David Sane
Wanda Seldon
```


Right Outer Join **Semantically, a Right Outer Join is the same as a Left Outer Join. The difference is that the required table is the rightmost table listed.**

Example 2.4.2.1. For each textbook, display all available information about the book and its usage in courses. Include textbooks that have never been used.

SQL SELECT Statement:

```
SELECT *
FROM USES RIGHT OUTER JOIN TEXTBOOK
ON USES.US_TX_ISBN = TEXTBOOK.TX„ISBN;
```

Result:

Us_co_course#	Us_tx_isbn	Us_pr_empid	Tx_isbn	Tx_title	Tx_year	Tx_publisher
22IS832	000-66574998	S43278	000-66574998	Database Management	1999	Thomson
22IS270	000-66574998	SS43278	000-66574998	Database Management	1999	Thomson
22IS270	77898-8769	SS43278	77898-8769	Principles of IS	2002	Prentice-Hall
22IS270	77898-8769	SK85977	77898-8769	Principles of IS	2002	Prentice-Hall
22IS270	77898-8769	CC49234	77898-8769	Principles of IS	2002	Prentice-Hall
20ECES212	0296437-1118	CC49234	0296437-1118	Programming in C++	2002	Thomson
22QA375	0296437-1118	SJ65436	0296437-1118	Programming in C++	2002	Thomson
22IS330	003-6679233	BC65437	003-6679233	Linear Programming	1997	Prentice-Hall
18ECON123	0296748-99	CM65436	0296748-99	Economics For Managers	2001	
22IS330	118-99898-67	SS43278	118-99898-67	Systems Analysis	2000	Thomson
22IS832	118-99898-67	SK85977	118-99898-67	Systems Analysis	2000	Thomson
22QA888	001-55-435	HT54347	001-55-435	Simulation Modeling	2001	Springer
			111-11111111	Data Modeling	2006	
			012-54765-32	Fundamentals of SQL	2004	

In SQL-92, the words RIGHT OUTER JOIN are used to designate a Right Outer Join operation. The use of the RIGHT JOIN keywords means that if the table listed on the right side of the join condition given in the ON clause has an unmatched row, it should be matched with a null row and displayed in the results.

Full Outer Join **A Full Outer Join operation is used when we want all rows in both the left and right tables included even though no corresponding rows exist in the other table.**

Example 2.4.3.1. Join the GRAD.STUDENT and TAKES tables making sure that each row from each table appears in the result.

Only syntaxes beginning with SQL-92 support the full outer join operation directly. This is accomplished through use of the FULL OUTER JOIN keywords.

SQL SELECT Statement:

```
SELECT *
FROM GRAD_STUDENT FULL OUTER JOIN TAKES
ON GRAD_STUDENT.GS_ST_SID = TAKES.TK_ST_SID;
```

Result:

Gs_st_sid	Gs_thesis	Gs_ugmajor	Tk_se_section#	Tk_se_qtr	Tk_se_year	Tk_se_co_course#	Tk_grade	Tk_st_sid
BG66765	N	Archeology	101	A	2007	22QA375	B	BG66765
BE76598	Y	Marketing	101	S	2006	22IS330	C	BE76598
KJ56656	Y	History	101	A	2007	22IS330	B	KJ56656
BE76598	Y	Marketing	101	A	2007	22IS330	A	BE76598
BG66765	N	Archeology	701	W	2007	22IS832	B	BG66765
GS76775	N	Archeology	101	A	2007	22IS330	C	GS76775
DT87656	N	Physics						
SW56547	Y	Finance						
AJ76998	Y	Child Care						
JD35477	N	Mathematics						
HJ45633	Y	History						
			101	A	2007	22QA375	A	KP78924
			101	A	2007	22QA375	A	KS39874
			101	A	2007	22IS330	A	KP78924
			101	A	2007	22IS330	A	KS39874
			701	W	2007	22IS832	A	KS39874

Note that the union of a Left Outer Join and a Right Outer Join is equal to the results of a Full Outer Join.

11.2.2.5 SQL and the Semi-Join and Semi-Minus Operations

Recall that the Semi-Join operation defines a relation that contains the tuples of R that participate in the Join of R with S. In other words, a Semi-Join is defined to be equal to the Join of R and S, projected back on the attributes of R. As such, a Semi-Join can be handled in SQL by using the Projection and Join operations.

Example 2.5.1 (Corresponds to Semi-Join Example 1 in Section 11.1.2.5). List the complete details of all courses for which sections are being offered in the fall 2007 quarter.

SQL SELECT Statement:

```
SELECT COURSE.*
FROM COURSE JOIN SECTION
ON COURSE.CO_COURSE# = SECTION.SE_CO_COURSE#
AND SE_YEAR = 2007 AND SE_QTR = 'A';
```

Result:

Co_name	Co_course#	Co_credit	Co_college	Co_hrs	Co_dpt_dcode
Database Concepts	22IS330	U	Business	4	7
Operations Research	22QA375	U	Business	2	3
Programming in C++	20ECES212	G	Engineering	3	6

A Semi-Minus operation occurs in cases where there are tuples in relation R that have no counterpart in relation S. A Semi-Minus operation can be handled in SQL through use of a combination of Join, Projection, and Minus operations.

Example 2.5.2 (Corresponds to Semi-Minus Example 1 in Section 11.1.2.5). List complete details of all courses for which sections are not offered in the fall 2007 quarter.

SQL SELECT Statement:

```
SELECT * FROM COURSE
MINUS
SELECT COURSE.* FROM COURSE JOIN SECTION
ON COURSE.CO_COURSE# = SECTION.SE_CO_COURSE#
AND SECTION.SE_YEAR = 2007 AND SECTION.SE_QTR = 'A';
```


Result:

Co_name	Co_course#	Co_credit	Co_college	Co_hrs	Co_dpt_dcode
Database Principles	22IS832	G	Business	3	7
Financial Accounting	18ACCT801	G	Education	3	4
Intro to Economics	15ECON112	U	Arts and Sciences	3	1
Intro to Economics	18ECON123	U	Education	4	4
Optimization	22QA888	G	Business	3	3
Principles of IS	22IS270	G	Business	3	7
Supply Chain Analysis	22QA411	U	Business	3	3
Systems Analysis	22IS430	G	Business	3	7

11.2.3 Subqueries

A complete SELECT statement embedded within another SELECT statement is called a subquery. In data retrieval, subqueries may be used (a) in the SELECT list of a SELECT statement, (b) in the FROM clause of a SELECT statement, (c) in the WHERE clause of a SELECT statement, and (d) in the ORDER BY clause of a SELECT statement. As illustrated in Section 10.2.1, subqueries can also be used in an INSERT...SELECT...FROM statement as well as the SET clause of an UPDATE statement. The output of a subquery can consist of a single value (a single-row subquery) or several rows of values (a multiple-row subquery). There are two types of subqueries: (a) uncorrelated subqueries where the subquery is executed first and passes one or more values to the outer query, and (b) correlated subqueries where the subquery is executed once for every row retrieved by the outer query.

11.2.3.1 Multiple-Row Uncorrelated Subqueries

Multiple-row subqueries are nested queries that can return more than one row of results to the parent query. Three multiple-row operators are used with multiple-row queries (IN, ALL, and ANY).

The IN and NOT IN Operators **When used in conjunction with a subquery, the IN operator evaluates if rows processed by the outer query are equal to any of the values returned by**

the subquery (i.e., it creates an OR condition). Example 3.1.1.1 illustrates how the IN operator can be used to display the course number, course name, and college of those courses for which sections have been offered. In this query, the subquery is executed first and returns the set of values (22QA375, 22IS270, 22IS330, 22IS832, and 20ECES212). The main query then displays the course number, name, and college for these courses. Note that a subquery of the form SELECT DISTINCT SECTION.CO_COURSE# FROM SECTION would have produced exactly the same result.

Example 3.1.1.1

```
SELECT COURSE.CO_COURSE#, COURSE.CO_NAME, COURSE.CO_COLLEGE
FROM COURSE
WHERE COURSE.CO_COURSE# IN
       (SELECT SECTION.SE_CO_COURSE#
       FROM SECTION);
```

Result:


```
Co_course# Co_name                Co_college

20ECES212  Programming in C++     Engineering
22IS270    Principles of IS       Business
22IS330    Database Concepts      Business
22IS832    Database Principles    Business
22QA375    Operations Research    Business
```

The NOT IN operator is the opposite of the IN operator and indicates that the rows processed by the outer query are not equal to any of the values returned by the subquery. Example 3.1.1.2 displays the course number, course name, and college of those courses for which sections have not been offered.

Example 3.1.1.2

```
SELECT COURSE.CO_COURSE#, COURSE.CO_NAME, COURSE.CO_COLLEGE
FROM COURSE
WHERE COURSE.CO_COURSE# NOT IN
       (SELECT SECTION.SE_CO„COURSE#
       FROM SECTION);
```

Result:

```
Co_course# Co_name                Co_college

15ECON112  Intro to Economics     Arts and Sciences
18ECON123  Intro to Economics     Education
22QA411    Supply Chain Analysis  Business
22QA888    Optimization           Business
18ACCT801  Financial Accounting   Education
22IS430    Systems Analysis       Business
```

The comparison operators =, <>, >, >=, <, and <= are *single-row operators.* **Observe the error message generated when the multiple-row operator IN is replaced by the single-row operator = (equals sign) in Example 3.1.1.3. This error is caused by the fact that there arc several course number-section number combinations in the TAKES table associated with a grade of A'. Observe what happens, however, when the single-row operator = is replaced by IN.**

The purpose of Example 3.1.1.3 is to display the section number and course number for which at least one grade of "A" has been assigned.

Example 3.1.1.3

```
SELECT DISTINCT SECTION.SE_SECTION#,  SECTION.SE_CO_COURSE#
FROM SECTION
WHERE (SECTION. SE_SECTION#, SECTION. SE_CO_COURSE#) =
        (SELECT TAKES . TK_SE_SECTION# , TAKES . TK_SE_CO_COURSE#
        FROM TAKES

        WHERE TAKES . TK_GRADE = ' A ' ) ;

(SELECT  TAKES.TK_SE_SECTION#,TAKES.TK_SE_CO_COURSE#

ERROR at line 4:  single-row subquery returns more than one row
```

Example 3.1.1.3 (Corrected)

```
SELECT DISTINCT SECTION.SE_SECTION#,  SECTION.SE_CO_COURSE#
FROM SECTION
WHERE (SECTION.SE_SECTION#,  SECTION.SE_CO_COURSE#) IN
        (SELECT TAKES.TK_SE_SECTION#,  TAKES.TK_SE_CO_COURSE#
        FROM TAKES
        WHERE TAKES.TK_GRADE = 'A');
```


Result:

```
Se_section# Se_co_course#
----------- -------------
        101 22IS330
        101 22QA375
        701 22IS832
```

The remaining examples in this section illustrate the use of the ANY and ALL operators in the context of the PROFESSOR table.

The ALL and ANY Operators **The ALL and ANY operators can be combined with the comparison operators =, <>, >, >=, <, and <= to treat the results of a subquery as a set of values, rather than as individual values. ANY specifies that the condition be true for *at least one value* from the set of values. ALL, on the other hand, specifies that the condition be true for *all values* in the set of values. Table 11.2 summarizes the use of the ALL and ANY operators in conjunction with other comparison operators.**

Table 11.2 Use of ANY and ALL operators in subqueries

Operator	Description
> ALL	Greater than the highest value returned by the subquery
>= ALL	Greater than or equal to the highest value returned by the subquery
< ALL	Less than the lowest value returned by the subquery
<= ALL	Less than or equal to the lowest value returned by the subquery
> ANY	Greater than the lowest value returned by the subquery

Table 11.2 Use of ANY and ALL operators in subqueries (continued)

Operator	Description
>= ANY	Greater than or equal to the lowest value returned by the subquery
< ANY	Less than the highest value returned by the subquery
<= ANY	Less than or equal to the highest value returned by the subquery
= ANY	Equal to any value returned by the subquery (same as the IN operator)

Example 3.1.2.1. Display the names and salaries of those professors who earn more than all professors in department number 3.

SQL SELECT Statement:

```
SELECT PROFESSOR.PR_NAME, PROFESSOR.PR_SALARY
FROM PROFESSOR
WHERE PROFESSOR.PR_SALARY > ALL
        (SELECT PROFESSOR.PR_SALARY
        FROM PROFESSOR
        WHERE PROFESSOR.PR_DPT_DCODE = 3);
```


Result:

```
Pr_name          Pr_salary

Mike Faraday         92 000
Marie Curie          99000
John Nicholson       99000
```

The following query includes Chelsea Bush and Tony Hopkins in the result since their salary is equal to the highest value returned by the subquery.

```
SELECT PROFESSOR.PR_NAME, PROFESSOR.PR_SALARY
FROM PROFESSOR
WHERE PROFESSOR.PR_SALARY >= ALL
        (SELECT PROFESSOR.PR_SALARY
        FROM PROFESSOR
        WHERE PROFESSOR.PR_DPT_DCODE = 3);
```

Result:

```
Pr_name          Pr_salary

Mike Faraday         92 000
Chelsea Bush         77000
Tony Hopkins         77 000
Marie Curie          99000
John Nicholson       99000
```

Example 3.1.2.2. Display the names and salaries of those professors who earn less than all professors in department number 7.

```
SELECT PROFESSOR.PR_NAME, PROFESSOR.PR_SALARY
FROM PROFESSOR
WHERE PROFESSOR.PR_SALARY < ALL
        (SELECT PROFESSOR.PR_SALARY
        FROM PROFESSOR
        WHERE PROFESSOR.PR_DPT_DCODE = 7);
```

Result:

```
Pr_name          Pr_salary
---------------- ----------
Ram Raj               44000
Prester John          44000
Laura Jackson         43000
```

Example 3.1.2.3. Rev ise the query in Example 3.1.2.2 and display the names and salaries of those professors with a salary that is less than or equal to that of the lowest paid professor in department number 7.

```
SELECT PROFESSOR.PR_NAME, PROFESSOR.PR_SALARY
FROM PROFESSOR
WHERE PROFESSOR.PR_SALARY <= ALL
       (SELECT PROFESSOR.PR_SALARY
       FROM PROFESSOR
       WHERE PROFESSOR.PR_DPT_DCODE = 7);
```

Result:

```
Pr_name          Pr_salary
---------------- ----------
John Smith            45000
Ram Raj               44000
Prester John          44000
Laura Jackson         43000
Cathy Cobal           45000
Jeanine Troy          45000
```

Example 3.1.2.4. Revise the query in Example 3.1.2.3 to exclude display of any employees in department number 7.

their

```
SELECT PROFESSOR.PR_NAME, PROFESSOR.PR_SALARY
FROM PROFESSOR
WHERE  PROFESSOR.PR_DPT_DCODE <> 7
AND PROFESSOR.PR_SALARY <= ALL
       (SELECT PROFESSOR.PR_SALARY
       FROM PROFESSOR
       WHERE PROFESSOR.PR_DPT_DCODE = 7);
```

Result:

```
Pr_name          Pr_salary
---------------- ----------
John Smith            45000
Ram Raj               44000
Prester John          44000
Laura Jackson         43000
Jeanine Troy          45000
```

Since < ANY returns all rows with a salary less than highest salary associated with department 3, the query in Example 3.1.2.5 displays the rows for all professors with a salary less than $77,000.

Example 3.1.2.5

```
SELECT PR_NAME, PR_SALARY
FROM PROFESSOR
WHERE PROFESSOR.PR^SALARY < ANY
```

```
(SELECT PROFESSOR.PR_SALARY
FROM PROFESSOR
WHERE PROFESSOR.PR_DPT_DCODE = 3);
```

Result:

Pr_name	Pr_salary
John Smith	45000
Kobe Bryant	66000
Ram Raj	44000
Prester John	44000
Alan Brodie	76000
Jessica Simpson	67000
Laura Jackson	43000
Jack Nicklaus	67000
Sunil Shetty	64000
Katie Shef	65000
Cathy Cobal	45000
Jeanine Troy	45000
Mike Crick	69000

On the other hand, since <= ANY returns all rows with a salary less than or equal to the highest salary associated with department 3, the query in Example 3.1.2.6 also displays the rows for both Chelsea Bush and Tony Hopkins.

Example 3.1.2.6

```
SELECT PR_NAME, PR_SALARY
FROM PROFESSOR
WHERE PROFESSOR.PR_SALARY <= ANY
       (SELECT PROFESSOR.PR_SALARY
       FROM PROFESSOR
       WHERE PROFESSOR.PR_DPT_DCODE = 3);
```

Result:

Pr_name	Pr_salary
John Smith	45000
Kobe Bryant	66000
Ram Raj	44000
Prester John	44000
Chelsea Bush	77000
Tony Hopkins	77000
Alan Brodie	76000
Jessica Simpson	67000
Laura Jackson	43000
Jack Nicklaus	67000
Sunil Shetty	64000
Katie Shef	65000
Cathy Cobal	45000
Jeanine Troy	45000
Mike Crick	69000

Since > ANY returns all rows with a salary greater than the lowest salary associated with professors who work in department 3, the query in Example 3.1.2.7 displays all rows except for the professor with the lowest salary (i.e., Laura Jackson) and the two professors who have a null salary.

Example 3.1.2.7

```
SELECT * PR_NAME, PR_SALARY
FROM PROFESSOR
WHERE PROFESSOR.PR_SALARY > ANY
        (SELECT PROFESSOR. PR_SALARY
        FROM PROFESSOR
        WHERE PROFESSOR.PR_DPT_DCODE = 3);
```

Result:

Pr_name	Pr_salary
John Smith	45000
Mike Faraday	92000
Kobe Bryant	66000
Ram Raj	44000
Prester John	44000
Chelsea Bush	77000
Tony Hopkins	77000
Alan Brodie	76000
Jessica Simpson	67000
Marie Curie	99000
Jack Nicklaus	67000
John Nicholson	99000
Sunil Shetty	64000
Katie Shef	65000
Cathy Cobal	45000
Jeanine Troy	45000
Mike Crick	69000

As expected, since >= ANY returns all rows with a salary greater than or equal to the lowest salary associated with department 3, all rows are returned in Example 3.1.2.8 except for those associated with the professors who have null salaries.

Example 3.1.2.8

```
SELECT PR_NAME, PR_SALARY
FROM PROFESSOR
WHERE PROFESSOR.PR_SALARY >= ANY
        (SELECT PROFESSOR.PR_SALARY
FROM PROFESSOR
WHERE PROFESSOR.PR_DPT_DCODE = 3);
```

Result:

Pr_name	Pr_salary
John Smith	45000
Mike Faraday	92000
Kobe Bryant	66000
Ram Raj	44000
Prester John	44000
Chelsea Bush	77000
Tony Hopkins	77000
Alan Brodie	76000
Jessica Simpson	67000
Laura Jackson	43000
Marie Curie	99000
Jack Nicklaus	67000

```
John Nicholson        99000
Sunil Shetty          64000
Katie Shef            65000
'Cathy Cobal          45 000
Jeanine Troy          45 0 00
Mike Crick            69000
```

As illustrated in Example 3.1.2.9, = ANY produces the same result as the IN operator. Note how the query in Example 3.1.2.9a restricts the rows displayed to those not associated with department 3.

Example 3.1.2.9

```
SELECT PR_NAME, PR_SALARY
FROM PROFESSOR
WHERE PROFESSOR.PR_SALARY = ANY
       (SELECT PROFESSOR.PR_SALARY
       FROM PROFESSOR
       WHERE PROFESSOR.PR_DPT_DCODE = 3);
```


Result:

```
Pr_name          Pr_salary

Laura Jackson        43 000
Jessica Simpson      67000
Jack Nicklaus        67000
Alan Brodie          76000
Chelsea Bush         77000
Tony Hopkins         77 0 00
```

Example 3.1.2.9a

```
SELECT PR_NAME, PR_SALARY
FROM PROFESSOR
WHERE PROFESSOR.PR_SALARY = ANY
       (SELECT PROFESSOR. PR_SALARY
       FROM PROFESSOR
       WHERE PROFESSOR.PR_DPT_DCODE = 3)
       AND PROFESSOR.PR_DPT_DCODE <> 3;
```

Result:

```
Pr_name          Pr_salary

Jack Nicklaus        67000
```

Although ANY and ALL are most commonly used with subqueries that return a set of numeric values, it is also possible to use them in conjunction with subqueries that return a set of character values.

The MAX and MIN Functions The MAX and MIN functions can be used in place of > ANY, < ANY, > ALL, and < ALL when the WHERE clause of the outer query involves a numeric value. Examples 3.1.3.1 through 3.1.3.3 are equivalent to Examples 3.1.2.1 through 3.1. 2.3 but use the MAX and MIN function instead of ANY or ALL.

Example 3.1..3.1 (Compare with Example 3.1.2.1)

```
SELECT PR_NAME, PR_SALARY
FROM PROFESSOR
```

```
WHERE PROFESSOR.PR_SALARY >
       (SELECT MAX (PROFESSOR. PR_SALARY)
       FROM PROFESSOR
       WHERE PROFESSOR.PR_DPT_DCODE = 3)
```

Result:

```
Pr_name          Pr_salary

Mike Faraday          92000
Marie Curie           99000
John Nicholson        99000
```

Example 3.1.3.2 (Compare with Example 3.1.2.2)

```
SELECT PR_NAME, PR_SALARY
FROM PROFESSOR WHERE
PROFESSOR. PR_SALARY <
       (SELECT MIN(PROFESSOR.PR_SALARY)
       FROM PROFESSOR
       WHERE PROFESSOR. PR_DPT_DCODE = 7);
```


Result:

```
Pr name          Pr_salary

Ram Raj              44000
Prester John         44000
Laura Jackson        43000
```

Example 3.1.3.3 (Compare with Example 3.1.2.3)

```
SELECT PR_NAME, PR_SALARY
FROM PROFESSOR
WHERE PROFESSOR.PR_SALARY <=
       (SELECT MIN(PROFESSOR.PR_SALARY)
       FROM PROFESSOR
       WHERE PROFESSOR.PR_DPT_DCODE = 7);
```

Result:

```
Pr_name          Pr_salary

John Smith           45000
Ram Raj              44000
Prester John         44000
Laura Jackson        43000
Cathy Cobal          45000
Jeanine Troy         45000
```

Subqueries in the FROM Clause **It is also possible to have nested subqueries in the FROM clause and in effect treat the subquery itself as if it were the name of a table.[21] This approach is often used when the subquery is a multiple-column subquery. As an example, suppose we are interested in listing all professors with a salary that is equal to or exceeds the average salary of all professors in their department.**

[21] Such a "temporary table" is more formally called an inline view.

Example 3.1.4.1. Display all professors with a salary that is equal to or exceeds the average salary of all professors in their department.

SQL SELECT Statement:

```
SELECT A.PR_NAME, A.PR_DPT_DCODE, A.PR_SALARY, B."Department Average"
FROM PROFESSOR A
     JOIN(SELECT PROFESSOR.PR_DPT_DCODE, AVG(PROFESSOR.PR_SALARY) AS "Department Average"
     FROM PROFESSOR
     GROUP BY PROFESSOR.PR_DPT_DCODE) B
     ON A.PR_DPT_DCODE = B.PR_DPT_DCODE
     AND A.PR_SALARY >= B."Department Average";
```

Observe how the shaded subquery in essence creates a temporary table that records the average salary of the professors in each department. The syntax calls for the table alias B to be located outside the parenthetical expression of the subquery since the execution of the subquery yields a temporary (i.e., virtual) table. The Join operation uses the PROFESSOR table and concatenates a row from PROFESSOR (table alias A) with a row from the temporary table created by the subquery (table alias B) when (a) the department number of the row from A matches the department number of a row from B, and (b) the salary of the professor in the row from A exceeds the average salary of the professors in his or her department.

Result:

Pr_name	Pr_dpt_dcode	Pr_salary	Department Average
Mike Faraday	1	92000	67666.6667
Chelsea Bush	3	77000	68000
Tony Hopkins	3	77000	68000
Alan Brodie	3	76000	68000
Marie Curie	4	99000	88333.3333
John Nicholson	4	99000	88333.3333
Ram Raj	6	44000	44000
Prester John	6	44000	44000
Sunil Shetty	7	64000	58000
Katie Shef	7	65000	58000
Mike Crick	9	69000	57000

Subqueries in the Column List The column list of a query can also include a subquery expression. A subquery in the column list must return a single value.

Example 3.1.5.1. Display all professors with a salary that is equal to or exceeds the average salary of all professors in their department along with the amount by which the average is exceeded.

SQL SELECT Statement:

```
1.  SELECT A.PR_NAME, A.PR_SALARY, A.PR_DPT_DCODE,
2.  ROUND((SELECT AVG(B.PR_SALARY)
3.  FROM PROFESSOR B
4.  WHERE A.PR_DPT_DCODE = B.PR_DPT_DCODE),0) AS "Avg Dept Salary",
5.  ROUND((A.PR_SALARY - (SELECT AVG(B.PR_SALARY) FROM PROFESSOR B
6.  WHERE A.PR_DPT_DCODE = B.PR_DPT_DCODE)),0) AS "Deviation"
7.  FROM PROFESSOR A
8.  WHERE A.PR_SALARY IS NOT NULL
9.  AND ROUND((A.PR_SALARY - (SELECT AVG(B.PR_SALARY) FROM PROFESSOR B
10. WHERE A.PR_DPT_DCODE = B.PR_DPT_DCODE)),0) > 0
11. ORDER BY "Deviation" DESC;
```

In order to explain this query, each line has been numbered. Two SELECT statements, each of which is the same, appear in the column list. The first (shown in *italics* on lines 2-4) determines the average salary for the professors in the department for a given professor, while the second (shown highlighted on lines 5 and 6) recalculates this average salary and uses it to determine the amount of the deviation between the salary of the professor and the average salary for the professors in the department. The WHERE clause in the main query (a) excludes from consideration those professors with a null salary (see line 8), and (b) includes only those professors whose salary exceeds that of their average salary in their department (note that the average salary of all professors in the professors' department is calculated a third time). The ORDER BY clause on line 11 allows the result to be displayed in descending order by the amount of the deviation between the salary of the professor and the average salary of the professors in their department.

Result:

Pr_name	Pr_salary	Pr_dpt_dcode	Avg Dept Salary	Deviation
Mike Faraday	92000	1	67667	24333
Mike Crick	69000	9	57000	12000
Marie Curie	99000	4	88333	10667
John Nicholson	99000	4	88333	10667
Chelsea Bush	77000	3	68000	9000
Tony Hopkins	77000	3	68000	9000
Alan Brodie	76000	3	68000	8000
Katie Shef	65000	7	58000	7000
Sunil Shetty	64000	7	58000	6000

Subqueries in the HAVING Clause **In addition to appearing in the WHERE clause, the FROM clause, and in the column list, a subquery can also be used in the HAVING clause. As an example, consider the following:**

Example 3.1.6.1. Display the name and average professor salary (for all departments) whose average salary exceeds the average salary paid to all professors at Madeira College.

SQL SELECT Statement:

```
SELECT DEPARTMENT.DPT_NAME, AVG(PROFESSOR.PR_SALARY)
FROM DEPARTMENT JOIN PROFESSOR
ON DEPARTMENT.DPT_DCODE = PROFESSOR.PR_DPT_DCODE
GROUP BY DEPARTMENT.DPT_NAME
HAVING AVG(PROFESSOR.PR_SALARY) >
    (SELECT AVG(PROFESSOR.PR_SALARY) FROM PROFESSOR);
```

Result:

Dpt_name	AVG(PROFESSOR.Pr_salary)
Economics	78000
QA/QM	68000

The SELECT statement in the HAVING clause acts as a filter that insures the selection of only those departments (i.e., groups) with an average salary greater than the average salary of the entire college.

11.2.3.2 Multiple-Row Correlated Subqueries

A correlated subquery can be used if it is necessary to check if a nested subquery returns ,no rows. Correlated subqueries make use of the EXISTS operator, which returns the value of true if a set is non-empty.

Example 3.2.1. Display the names of professors who have offered at least one section.

```
SELECT  PROFESSOR.PR_NAME
FROM  PROFESSOR
WHERE  EXISTS
        (SELECT *
       FROM  SECTION
       WHERE  PROFESSOR.PR_EMPID = SECTION.SE_PR_PROFID);
```

Result:

```
Pr_name

Ram Raj
Tony Hopkins
Katie  Shef
Cathy  Cobal
```


A correlated nested subquery is processed differently from an uncorrelated nested subquery. Instead of the execution of the subquery serving as input to its parent query (i.e., the outer query), in a correlated subquery the subquery is executed once for each row in the outer query. In addition, execution of the subquery stops and the EXISTS condition of the main query is declared true for a given row should the condition in the subquery be true. For example, using the data in the PROFESSOR and SECTION tables, for Ram Raj the execution of the subquery stops when the fourth row of the PROFESSOR table is evaluated against the seventh row of the SECTION table because **PROFESSOR.Pr_empid = SECTION.Se_pr_empid** at this point. Thus, in essence each value of **PROFESSOR.Pr name** is treated as a constant during the evaluation. If the NOT EXISTS operator were used instead of the EXISTS operator, the names of professors who have not offered a section would be displayed.

The NOT EXISTS operator can be used as a way to express the DIVIDE operator in SQL. Recall the example in Section 11.1.2.4 which poses the question: What are the course numbers and course names of those courses offered in all quarters during which sections are offered? An SQL statement that produces an answer to this query is given below. The individual lines have been numbered to facilitate discussion of the execution of the query.

SQL SELECT Statement:

```
1.  SELECT COURSE.CO_COURSE#,  COURSE.CO_NAME
2.  FROM COURSE
3.  WHERE NOT EXISTS
4.          (SELECT DISTINCT A.SE_QTR
5.          FROM SECTION A
6.          WHERE NOT EXISTS
7.                  (SELECT *
8.                  FROM SECTION B
9.                  WHERE COURSE.CO_COURSE# = B.SE_CO_COURSE#
10.                 AND A. SE_QTR = B. SE_QTR) ) ;
```

Result:

```
Co_course#  Co_name

22QA375     Operations Research
```

Both the subquery that begins on line 4 and the subquery that begins on line 7 refer to the SECTION table. To prevent ambiguity, these two uses of the SECTION table have been assigned the aliases of A and B, respectively. Since the two uses of the NOT EXISTS operator may make this query difficult to understand, let's begin by assuming the first row retrieved from the COURSE table as part of the execution of lines 1-3 defines COURSE.CO_COURSE# as 15ECON112. Replacing COURSE.CO_COURSE# in line 9 with '15ECON112' causes the execution of lines 4-10 to generate the result shown below.

```
4.  (SELECT DISTINCT A.SE_QTR
5.  FROM SECTION A
6.  WHERE NOT EXISTS
7.  (SELECT *
8.  FROM SECTION B
9.  WHERE '15ECON112' = B.SE_CO_COURSE#
10. AND A.SE_QTR = B.SE_QTR));
```


Result:

```
Se_qtr

A
S
U
W
```

Since Course* 15ECON112 does not appear at all in the SECTION table, the NOT EXISTS condition is true for each of the four quarters. On the other hand, when Course* 22QA375 replaces Course* 15ECON112 line 9, the NOT EXISTS condition is false for all four quarters (note that a section of Course* 22QA375 is offered during each quarter in the SECTION table) and thus "no rows selected" is the result when lines 4-10 are executed.

```
4.  (SELECT DISTINCT A.SE_QTR
5.  FROM SECTION A
6.  WHERE NOT EXISTS
7.  (SELECT *
8.  FROM SECTION B
9.  WHERE '22QA375' = B.SE_CO_COURSE#
10. AND A.SE_QTR = B.SE_QTR));
```

Result:

```
No rows selected.
```

In other words, the NOT EXISTS condition in line 3 is true for Course* 22QA375. This SQL formulation corresponds to the following informal statement: "Display the course numbers of those courses such that there does not exist a quarter during which the course is not offered." It is left as an exercise for the reader to determine the result when lines 4-10 are executed for other courses (e.g., Course* 22IS330).

In SQL, an aggregate function takes as input a set of values, one from each row in a group of rows, and returns one value as the result. As illustrated in Section 11.2.1.4, the COUNT function, one of the most commonly used aggregate functions, counts the non-NULL values in a column. Other aggregate functions include retrieving the sum, average, maximum, and minimum of a series of numeric values. Another type of request involves the grouping of rows in a table or tables by the value of some of their attributes and then applying an aggregate function independently to each group.

Example 3.3.1. Using the data in the STUDENT and TAKES tables, count the number of sections taken by each student. Be sure to include those students who have never taken a class. The individual lines have been numbered to facilitate the discussion of the execution of the query.

SQL SELECT Statement:


```
1.  SELECT TAKES.TK_ST_SID, STUDENT.ST_NAME, COUNT(*) AS "Sections Taken"
2.  FROM STUDENT JOIN TAKES
3.  ON STUDENT.ST_SID = TAKES.TK_ST_SID
4.  GROUP BY TAKES.TK_ST_SID, STUDENT.ST_NAME
5.  UNION
6.  SELECT STUDENT.ST_SID, STUDENT.ST_NAME, 0
7.  FROM STUDENT
8.  WHERE STUDENT.ST_SID NOT IN
9.  (SELECT TAKES.TK_ST_SID FROM TAKES)
10. ORDER BY "Sections Taken" DESC;
```

In addition to illustrating the use of a grouping operation and an aggregate function, this query also contains a join, a nested subquery, and a union. Execution of the query begins with lines 1-4, which count the number of sections taken by those students who have taken at least one class. Note that a value of COUNT(*) is obtained for each combination of a TAKES.Tk st_sid, and STUDENT.St_name. On the other hand lines 6-10, when executed, begin with the execution of line 9 and identify those students who have not taken a section. For each qualifying student, the numeric literal zero (0)[22] is displayed along with the value of STUDENT.St_sid, STUDENT.St_name. Since the content of columns 1 and 2 on lines 1 and 6 share the same domain and COUNT(*) and 0 also share the same domain (i.e., both are numeric values), the UNION operation on line 5 can take place. When a UNION operation takes place in a query, the sorting operation (i.e., represented by the ORDER BY clause on line 10) applies to the collective results from all SELECT statements involved in the UNION.

Result:

Tk_st_sid	St_name	Sections Taken
KS39874	Sweety Kramer	3
BE76598	Elijah Baley	2
BG66765	Gladis Bale	2
KP78924	Poppy Kramer	2
GS76775	Shweta Gupta	1
KJ56656	Joumana Kidd	1

In addition to listing column names, expressions (e.g., see Section **11.2.1.2** and **11.2.1.3),** functions (e.g., **COUNTC)),** and SELECT statements (e.g., see Section **11.2.3.1),** the SELECT list of a SELECT statement may also contain either ;i numeric constant (e.g., the numeric literal **0)** or a string constant.

AJ76998	Jenny Aniston	0
DT87656	Tim Duncan	0
FR45545	Rick Fox	0
FV67733	Vanessa Fox	0
HJ45633	Jenna Hopp	0
HT67657	Troy Hudson	0
JD35477	Diana Jackson	0
OD76578	Daniel Olive	0
SD23556	David Sane	0
SW56547	Wanda Seldon	0

The same result could be obtained using the Left Outer Join shown below, since the value of COUNT(TAKES.Tk_st_sid) is zero when a row for a student not enrolled in a section is concatenated with the row of null values in TAKES.

SQL SELECT Statement:

```
SELECT STUDENT.ST_SID, STUDENT.ST_NAME, COUNT(TAKES.TK_ST_SID)
AS "Sections Taken"
FROM STUDENT LEFT OUTER JOIN TAKES
ON STUDENT.ST_SID = TAKES.TK_ST_SID
GROUP BY STUDENT.ST_SID, STUDENT.ST_NAME
ORDER BY "Sections Taken" DESC;
```


Example 3.3.2. Display the maximum, minimum, total, and average salary for the professors affiliated with each department. In addition, count the number of professors in each department as well as the number of professors in each department with a not null salary.

SQL SELECT Statement:

```
SELECT DEPARTMENT.DPT_NAME AS "Dept Name", DEPARTMENT.DPT_DCODE
AS "Dept Code",
MAX(PROFESSOR.PR_SALARY) AS "Max Salary", MIN(PROFESSOR.PR_SALARY)
AS "Min Salary",
SUM (PROFESSOR.PR_SALARY) AS "Total Salary",
ROUND(AVG(PROFESSOR.PR_SALARY),0) AS "Avg Salary", COUNT(*) AS "Size",
COUNT(PROFESSOR.PR_SALARY) AS "# Sals"
FROM DEPARTMENT JOIN PROFESSOR
ON DEPARTMENT.DPT_DCODE = PROFESSOR.PR_DPT_DCODE
GROUP BY DEPARTMENT.DPT_NAME, DEPARTMENT.DPT_DCODE
ORDER BY "# Sals" DESC;
```

Result:

Dept Name	Dept Code	Max Salary	Min Salary	Total Salary	Avg Salary	Size	# Sals
Economics	4	99000	67000	265000	88333	3	3
QA/QM	3	77000	43000	340000	68000	5	5
Economics	1	92000	45000	203000	67667	3	3
IS	7	65000	45000	174000	58000	3	3
Philosophy	9	69000	45000	114000	57000	3	2
Mathematics	6	44000	44000	88000	44000	3	2

Since department names are not unique but required as part of the output, grouping must be done on both DEPARTMENT.Dpt_name and DEPARTMENT.Dpt_dcode. In addition, instead of grouping by DEPARTMENT.Dpt dcode, grouping could have been by DEPARTMENT.Dpt_college had the name of the college housing the department had been required as opposed to the department code.

CHAPTER 12

Advanced Data Manipulation Using SQL

SQL for data manipulation is covered extensively in the previous chapter. In this chapter, we move on to study some of the advanced features of SQL. In Chapter 6, it is pointed out that while it is possible to specify all the integrity constraints as a part of the conceptual modeling process, some cannot be expressed explicitly or implicitly in the schema of the data model. Chapter 10 presented implementation of a majority of the schema-based or declarative constraints (such as domain constraints, key constraints, entity integrity constraints, and referential integrity constraints). It is also pointed out in Chapter 6 that some business rules pertaining to a valid state of a database or legal transitions of a database state may require procedural intervention. While application programs can be used to handle such procedural constraints, most DBMS products offer general-purpose procedural language support capable of implementing these constraints within the database. Assertions and triggers are SQL-92 facilities for capturing sophisticated declarative constraints (table-level constraints) and procedural constraints, respectively. In a database environment, views play a number of roles. As discussed in Chapter 6, views are "virtual tables" created using SQL/DDL. Since the script used in the creation of each of these database objects, i.e., ASSERTION, TRIGGER, VIEW, also requires an understanding of the SQL data manipulation statements, these topics are covered in this chapter.

The SQL-92 standard includes a number of built-in functions that can be used in an SQL statement anywhere that a constant of the same data type can be used. Section 12.2 covers a number of these functions while Section 12.3 focuses more on functions that facilitate the manipulation of dates and times. Discussion of SQL for data manipulation and retrieval concludes in Section 12.4 with a series of examples that apply a number of the SQL features introduced in this chapter and in Chapter 11.

121 Assertions, Triggers, and Views

This section revisits SQL/DDL because the DDL constructs ASSERTION and VIEW presented here embed the data retrieval construct of SQL/DML, namely, the SELECT statement discussed in Chapter 11.

1211 Specifying an Assertion in SQL

There are times when a user-specified business rule may entail definition of a constraint not covered by the declarative constraints illustrated in Chapter 10 (for example, a constraint to be satisfied by a table as opposed to individual rows of a table or a constraint that spans multiple tables). The SQL-92 standard offers a means to specify such a "general" constraint via what is called a declarative assertion using a CREATE ASSERTION construct. Let

us review an example to understand the utility of this construct using the PATIENT, MEDICATION, and ORDERS tables from Chapter 10. For convenience of the reader, Figure 10.1 is reproduced here as Figure 12.1. In addition, the SQL/DDL script from Box 3 in Section 10.1.1.1 is reproduced here as Figure 12.2.

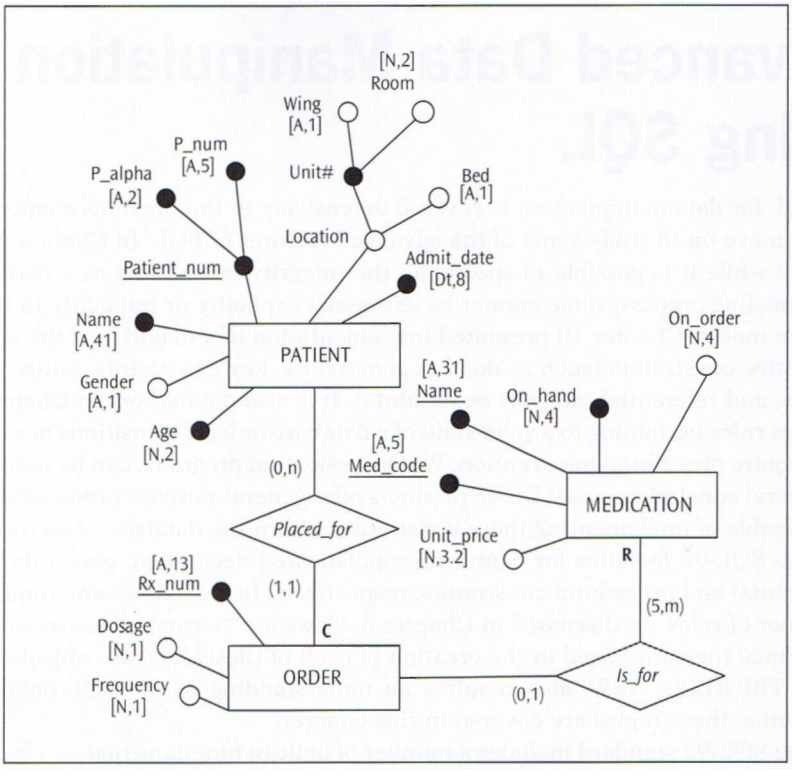


Figure 12.1a ER diagram: An excerpt from a hypothetical medical information system

> Constraint	Gender	IN ('M', 'F')
> Constraint	Age	IN (1 through 90)
> Constraint	Bed	IN ('A', 'B')
> Constraint	Unit_price	< 4.50
> Constraint	(Qty_onhand + Qty_onorder)	IN (1000 through 3000)
> Constraint	Dosage	DEFAULT 2
> Constraint	Dosage	IN (1 through 3)
> Constraint	Frequency	DEFAULT 1
> Constraint	Frequency	IN (1 through 3)

Figure 12.1b Semantic integrity contstraints for the Fine-granular Design-Specific ER model

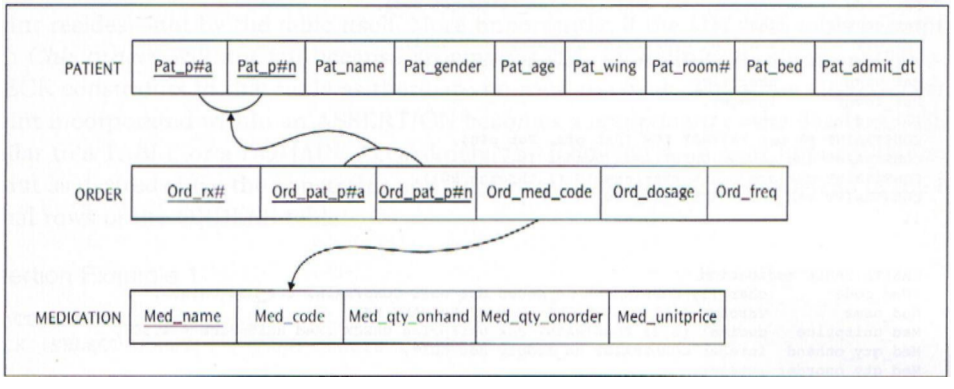


Figure 12.1c Relational schema for the ERD in Figure 12.1a

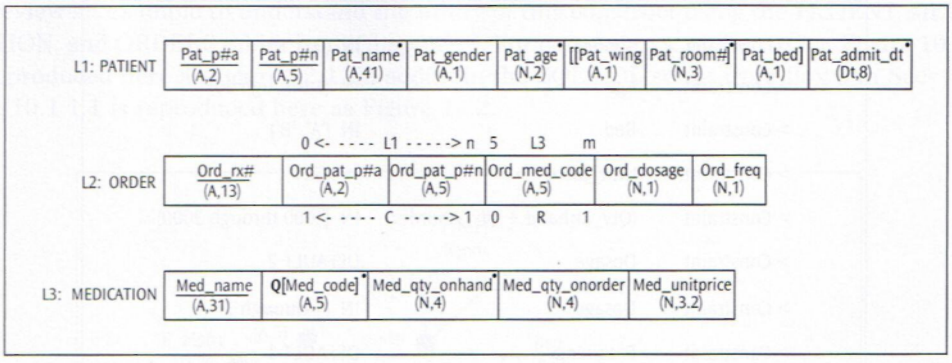


Figure 12.1d An information-preserving logical schema for the ERD in Figure 12.1a


```
CREATE TABLE patient
(Pat_p#a        char (2),
Pat_p#n         char (5),
Pat_name        varchar (41) constraint nn_Patnm not null,
Pat_gender      char (1),
Pat_age         smallint constraint nn_Patage not null,
Pat_admit_dt    date constraint nn_Patadmdt not null,
Pat_wing        char (1),
Pat_room#       integer,
Pat_bed         char (1),
CONSTRAINT pk_pat PRIMARY KEY (Pat_p#a, Pat_p#n),
CONSTRAINT chk_gender CHECK (Pat_gender IN ('M', 'F')),
CONSTRAINT chk_age CHECK (Pat_age IN (1 through 90)),
CONSTRAINT chk_bed CHECK (Pat_bed IN ('A', 'B'))
);


CREATE TABLE medication
(Med_code        char (5) CONSTRAINT nn_medcd not null CONSTRAINT unq_med UNIQUE,
Med_name         varchar (31) CONSTRAINT pk_med  PRIMARY KEY,
Med_unitprice    decimal (3,2) CONSTRAINT chk_unitprice CHECK (Med_unitprice < 4.50),
Med_qty_onhand   integer CONSTRAINT nn_medqty not null,
Med_qty_onorder integer,
CONSTRAINT chk_qty CHECK ((Med_qty_onhand + Med_qty_onorder) BETWEEN 1000 AND 3000)
);


CREATE TABLE orders
(Ord_rx#         char (13) CONSTRAINT pk_ord  PRIMARY KEY,
Ord_pat_p#a      char (2) CONSTRAINT nn_ord_pat_p#a not null,
Ord_pat_p#n      char (5) CONSTRAINT nn_ord_pat_p#n not null,
Ord_med_code     char (5) CONSTRAINT fk_med REFERENCES medication (Med_code)
ON DELETE RESTRICT ON UPDATE RESTRICT,
Ord_dosage       smallint DEFAULT 2 CONSTRAINT chk_dosage CHECK (Ord_dosage BETWEEN 1 AND 3),
Ord_freq         smallint DEFAULT 1 CONSTRAINT chk_freq CHECK (Ord_freq IN (1, 2, 3)),
CONSTRAINT fk_pat FOREIGN KEY (Ord_pat_p#a, Ord_pat_p#n)
REFERENCES patient (Pat_p#a, Pat_p#n) ON DELETE CASCADE ON UPDATE CASCADE
);
```

Figure 12.2 Chapter 10, Box 3 reproduced

Suppose a business rule says that at any given time there must be at least 100 orders present in the system; otherwise, the in-house pharmacy may not be cost-justifiable. A syntactically correct CREATE TABLE script specifying this rule is:

```
CREATE TABLE orders
 (Ord_rx#                char  (13)  CONSTRAINT pk_ord   PRIMARY KEY,
Ord_pat_p#a              char  (2)   CONSTRAINT nn_ord_pat_p#a not null,
Ord_pat_p#n              char  (5)   CONSTRAINT nn_ord_pat_p#n not null,
Ord_med_code             char  (5)   CONSTRAINT fJc_med REFERENCES medication  (Med_code)
ON DELETE RESTRICT ON UPDATE RESTRICT,
Ord_dosage               smallint  CONSTRAINT  chk_dosage  CHECK   (Ord_dosage
BETWEEN 1 AND 3) ,
Ord_freq                 smallint CONSTRAINT chk_freq CHECK  (Ord_freq IN  (1,  2,  3)),
CONSTRAINT  Chk_orders  CHECK  (SELECT COUNT  (*)  FROM orders  >=  100),
CONSTRAINT  fk_pat  FOREIGN  KEY  (Ord_pat_p#a,  Ord_pat_p#n)
REFERENCES patient  (Pat_p#a, Pat_p#n) ON DELETE CASCADE ON UPDATE CASCADE
) ;
```

The highlighted CHECK constraint *Chk_orclers* is intended to impose the business rule stated. Unfortunately. *Chk_orders* does not achieve the intended goal because a CHECK constraint in a table is expected to be satisfied by every row in the table where the constraint resides—not by the table itself. More importantly, if the ORDERS table is empty, then *Chk_orders* will *not* fail because an empty table, by definition, always satisfies all CHECK constraints in that table as there are no rows to check. The same CHECK constraint incorporated within an ASSERTION becomes a component of the database schema similar to a TABLE or a DOMAIN. Accordingly, the following statement imposes the constraint as desired since the constraint now applies to the ORDERS table instead of the individual rows of the ORDERS table.

Assertion Example 1

```
CREATE ASSERTION Chk_orders
CHECK  (SELECT COUNT  (*)  FROM orders >= 100);
```

The general syntax for the CREATE ASSERTION construct is:

```
CREATE ASSERTION assertion_name CHECK  (conditional-expression);
```

where:

- *assertion_name* is a user-supplied name for the declarative assertion
- *conditional-expression* is an expression of arbitrary complexity referring to one or more base tables in the database

Recall from Chapter 10 that enforcement of *total participation* of a parent in a relationship (i.e., min > 0) requires a general constraint specification mechanism. *A general constraint* may be described as one that applies to arbitrary combinations of columns in arbitrary combinations of base tables (Date and Darwen, 1997). In that case, we ought to be able to enforce this constraint using a declarative assertion. For example, the ER diagram and the information-preserving logical schema shown in Figure 12.1 indicates that every medication should be included in at least 5 orders. This cannot be implemented through a table creation DDL. A declarative assertion that defines this constraint is of the following form.

Assertion Example 2

```
CREATE   ASSERTION    Chk_ordr_per_med
CHECK  (NOT EXISTS
   (SELECT *
  FROM medication
  WHERE medication.Med_code NOT IN
         (SELECT Ord_med_code
        FROM orders
        GROUP BY Ord_med_code
        HAVING COUNT  (*)  >= 5))
 );
```

This example seeks to generate a list of medications that violate the condition specified and checks for the return of an empty set as the result. Thus, the assertion is violated if the result of the conditional expression (i.e., the query) is not an empty set. A critical question here is what happens if at the time of specifying the assertion *Chk_ordr_per_med* (a) the ORDERS table is empty, or (b) it already has data that violates the condition specified in the assertion. According to the SQL-92 standard, if a new constraint is defined and the existing database state does not satisfy the constraint, the constraint fails (i.e., is not created). At this point, it is the responsibility of the database designer to discover the cause of the violation and either amend the constraint or rectify the data in the database.

The participation of PATIENT in the *Placed Jor* relationship with ORDERS is optional (see Figure 12.1a or 12.Id). This constraint is in force by default. Suppose, for some strange reason, a business rule dictates that there *must* always be some patients who have not placed any orders for medication. First of all, this rule cannot be shown in the ERD diagram and should therefore be included in the list of semantic integrity constraints in Figure 12.1b. That said, how is such a constraint enforced? Let us evaluate the declarative assertion in the next example.

Assertion Example 3

```
CREATE   ASSERTION    Chk_no_ordr_pats
CHECK  (NOT EXISTS
      (SELECT *
     FROM patient
     WHERE  (Pat_p#a,  Pat_p#n)  NOT IN
           (SELECT Ord_pat_p#a,  Ord_pat_p#n
          FROM orders))
);
```

This assertion seeks to generate a list of patients that violate the condition specified and checks for the return of an empty set as the result. Thus, the assertion is violated if the result of the conditional expression (i.e., the query) is not an empty set.

As another illustration of the utility of a declarative assertion, observe that Unit# of the PATIENT entity type in the ER diagram (see Figure 12.1a) is a composite attribute. This information is preserved in the logical schema (Figure 12.Id) by enclosing the atomic attributes Pat_wing and Pat_room# in brackets [|. While it is impossible to map this information to the tabic definition, is it possible to capture the business rule that Unit* and therefore [Pat_wing, Pat_room#] is a *mandatory* composite attribute via a declarative assertion?

The answer is a qualified "Yes"—i.e., it is possible to specify that **Pat_wing** and **Pat_room#** together cannot have a null value, while individually each of these two columns may have a null value in any row of the PATIENT table. It is not possible, however, to specify that **[Pat_wing, Pat_room#]** represents a composite attribute since, by definition, there is no such thing as composite attribute/column in a relational database.

The declarative assertion that prohibits the occurrence of null values in both **Pat_wing** and **Pat_room#** at the same time in the same row of the PATIENT table is:

Assertion Example 4

```
CREATE  ASSERTION  Chk_unit#
CHECK  (NOT EXISTS
      (SELECT *
     FROM patient
     WHERE Pat_wing IS NULL AND Pat_room# IS NULL)
);
```

Finally, let us examine this declarative assertion:

Assertion Example 5


```
CREATE   ASSERTION    Chk_costcontrol
CHECK  ((Med_unitprice * Med_qty_onhand)  < 5000
);
```

The assertion can be viewed as making sure that locked-up capital in stocking a medication is controlled at below S5,000. Do we need a declarative assertion to enforce this business rule? Not at all. This CHECK constraint is applicable at the row level of the MEDICATION table and hence can be defined as a table-level constraint.

Just as a declarative assertion can be created, it can also be deleted when no longer needed using the syntax:

```
DROP   ASSERTION   Assertion_name;
```

For example,

```
DROP ASSERTION Chk_costcontrol;
```

Note that there are no behavior options like CASCADE or RESTRICT associated with the DROP statement for an ASSERTION. Also, SQL-92 does not provide an ALTER construct for ASSERTIONS.

In general, implementing declarative assertions by a DBMS is not as efficient as implementing a CHECK constraint on a column, row, or domain. Therefore, it is advisable to use declarative assertions only when other declarative means will not work. Another fundamental problem with a declarative assertion is that like the other declarative constraints in a database schema, it offers only one option—that is, aborting the operation that violates the assertion. It may be useful if the DBMS offers other options whereby an action can be executed automatically (e.g., sending a message to the user) when the assertion is violated.

12.12 Triggers in SQL

A trigger allows specification of certain types of active rules. The concept of triggers is rooted in active databases. In simple terms, an active database must react to external events

when the events occur. The model used for specifying active database rules is called the Event-Condition-Action (ECA) model. A trigger is an element in the database schema that implements the ECA model where:

a. an event is an external request for an operation on the database (e.g., insert a row into a table),
b. a condition is a Boolean expression that must evaluate TRUE in order for the action to be executed, and
c. an action is a procedure that stipulates what needs to be done when the trigger is fired (e.g., insert a row, display a message).

In general, the possible events are INSERT and DELETE rows in a base table or UPDATE specific columns. The action can be executed BEFORE or AFTER the event based on whether the condition is to be tested in the database state that exists before or after the event. The trigger granularity defines what constitutes an event. For instance, a row-level granularity implies that a change to a single row is an event, while a statement-level granularity indicates that the statement (e.g., INSERT, DELETE, UPDATE) is the event regardless of the number of rows affected by the execution of the trigger.

Interestingly triggers existed in early versions of SQL, but are not part of the SQL-92 standard. However, most DBMS products support some level of trigger processing. Triggers are part of the SQL-99 standard. This section contains examples of simple triggers created and executed using an Oracle9i database platform. None of these examples illustrates the many nuances that complicate the application of triggers. For more information, the reader is directed to the reference material in the selected bibliography at the end of this chapter (Kifer, et. al, 2005; Gulutzan and Pelzer, 1999).

12.1.2.1 Trigger Fundamentals

The syntax for creating a trigger is as follows:

```
CREATE [OR REPLACE] TRIGGER trigger_name
[BEFORE|AFTER]
[DELETE|INSERT|UPDATE [OF column]]
ON table_name
[ FOR EACH { ROW | STATEMENT} ]
[WHEN condition]
statement-list
```

There are 12 basic types of triggers. A trigger's type is defined by the type of triggering transaction, and by the level at which the trigger is executed. The 12 basic types of triggers are derived from the following three categories: row-level triggers, statement-level triggers, and BEFORE and AFTER triggers. Before and after triggers execute immediately before or after inserts, updates, and deletions. Since a trigger can be (a) either a row-level or statement-level trigger, and (b) can execute immediately before or after inserts, updates, and deletes, the following 12 possible configurations arise.

BEFORE INSERT row	AFTER UPDATE row
BEFORE INSERT statement	AFTER UPDATE statement
AFTER INSERT row	BEFORE DELETE row
AFTER INSERT statement	BEFORE DELETE statement
BEFORE UPDATE row	AFTER DELETE row
BEFORE UPDATE statement	AFTER DELETE statement

Row-level triggers execute once for each row in a transaction, whereas statement-level triggers execute once for each transaction. For example, if a single transaction updates 100 rows in a.table, a row-level trigger on the table would be executed 100 times (once either before or after the update of a row) while a statement-level trigger would be executed only one time (either before or after the execution of the updates to the 100 rows).

12.1.2.2 The AFTER DELETE Trigger

A common use of a trigger is to create a historical record of a transaction. The following example illustrates the creation of a database trigger that serves as an audit trail by inserting a row into the PATIENT_AUDIT table whenever a patient is deleted from the clinic (i.e., discharged).

AFTER DELETE Trigger Example.

```
CREATE TRIGGER PATIENT_AFT_DEL_ROW
AFTER DELETE ON PATIENT
FOR EACH ROW
BEGIN
INSERT INTO PAT I ENT_AUD IT VALUES ( : OLD . PAT_P#A, : OLD . PAT_P#N, CURRENT.
DATE);
END;
```


The following comments discuss specific aspects involved in the creation of this trigger.

1. **The PATIENT_AUDIT table contains three columns: Pau_p#a, Pau_p#n, and Pau_datetime. The CREATE TABLE statement shown below was used in the creation of the table.**

   ```
   CREATE TABLE patient_audit
   (Pau_p#a  char(2),
   Pau_p#n  char(5),
   Pau_datetime  date);
   ```

2. **Observe that the name of the trigger (PATIENT_AFT_DEL_ROW) reflects its nature. The component, PATIENT, suggests its association with the PATIENT table, _AFT indicates that the trigger is an after trigger, _DEL indicates that the trigger involves a delete operation, and _ROW indicates that the trigger is a row-level trigger. It is important that the content of the remainder of the trigger be consistent with the name of the trigger if the name of the trigger is to accurately reflect the nature of the trigger.**
3. **As indicated in the syntax, in the creation of a trigger, the name of the trigger must be followed by the BEFORE or AFTER keyword, indicating whether the trigger should be fired before or after the operation that causes it to be fired. Immediately following the BEFORE or AFTER keyword is the action (or actions) with which the trigger is associated. This can be INSERT, UPDATE, or DELETE, or any combination of these separated by OR. This trigger is to be executed after the deletion of each row from the PATIENT table.**
4. **The FOR EACH ROW keyword defines the behavior of the trigger when it is fired by a statement affecting multiple rows. The default behavior is to fire the trigger only once, regardless of the number of rows affected (i.e., a statement-level trigger is assumed unless the FOR EACH ROW keyword is used).**

Every trigger must contain a series of one or more statements[1] bracketed by the words BEGIN and END.[2] In this example there is only one statement, an INSERT statement that inserts a row into the PATIENT_AUDIT table. The :OLD keyword is used in the INSERT statement as a prefix to refer to the old value of the **PATIENT.Pau_p#a** and **PATIENT.Pau_p#n** columns. Although not used in this example since a delete operation cannot result in a new value for a column, the :NEW keyword is used as a prefix in both insert and update operations to refer to the new value of a column. CURRENT_DATE in the INSERT statement is an SQL-92 function that records the current date. Section 12. 3 contains additional information about the CURRENT_DATE and other SQL-92 functions.

To illustrate the execution of the trigger, observe the content of the PATIENT and PATIENT_AUDIT tables before and after the deletion of the row for the patient Bill Davis.

Content of Tables Prior to Deletion

```
SELECT * FROM PATIENT;

Pat_pTfa  Pat_p#n  Pat_name        Pat_gender     Pat_age  Pat_admit_dt  Pat_wing   Pat_room#  Pat_bed

DB        77642    Davis, Bill     M                  27  2007-07-07    B               108 B
GD        72222    Grimes, David                      44  2007-07-12

SELECT * FROM PATIENT_AUDIT;

no rows selected
```

Deletion of Patient Bill Davis

```
DELETE FROM PATIENT WHERE PATIENT.PAT_NAME LIKE "SDavis, Bill%';
1 row deleted.
```

Contents of Tables After Deletion

```
SELECT * FROM PATIENT;

Pat_p#a   Pat_p#n  Pat_name          Pat^gender     Pat_age  Pat_admit_dt  Pat_wing   Pat_room#  Pat_bed

GD        72222    Grimes, David                      44  2007-07-12

SELECT * FROM PATIENT_AUDIT;

Pau p#a   Pau_p#n   Pau_datetime

DB        77642    2006-07-22
```

[1] In Oracle9i, these statements can be a combination of SQL and PL/SQL statements. This section contains only enough information about PL/SQL to explain the examples. For a comprehensive discussion of PL/SQL, refer to Sunderraman (1999) or books published by Oracle Press.

[2] The words BEGIN and END enclose the executable portion of what is known as a PL/SQL block.

12.1.2.3 The BEFORE INSERT Trigger

The following is an example of a trigger that disallows the insertion of an order for a patient who has already received three orders for the same medication (a business rule that cannot be specified by a declarative constraint when defining the ORDERS table). The individual lines have been numbered to facilitate the discussion but otherwise are not part of the trigger.

BEFORE INSERT Trigger Example

```
I.   CREATE OR REPLACE TRIGGER ORDERS_BEF_INS_ROW
2.   BEFORE INSERT ON ORDERS FOR EACH ROW
3.   DECLARE
4.       NO_ORDERS NUMBER;
5.       TEMP_MED_NAME  VARCHAR(20);
6.   BEGIN
7.       SELECT COUNT(*)  INTO NO_ORDERS
8.       FROM ORDERS
9.       WHERE ORDERS.ORD_PAT_P#A =  :NEW.ORD_PAT_P#A
10.      AND ORDERS.ORD_PAT_P#N -  :NEW.ORD_PAT_P#N
II.      AND ORDERS . ORD_MED_CODE = : NEW. ORD_MED_CODE;
12.           IF NO_ORDERS >= 3 THEN
13.              SELECT MED_NAME INTO TEMP_MED_NAME FROM MEDICATION
14.              WHERE MED_CODE = : NEW. ORD_MED_CODE;
15.              RAISE_APPLICATION_ERROR (-20000, 'Patient '||;NEW.ORD_PAT_P#A||:NEW.ORD_PAT_P#N||
16.              ' has had too much ' | |TEMP_MED_NAME) ;
17.           END IF;
18.  END;
```


1. **As indicated in the trigger syntax at the beginning of Section 12.1.2.1, the OR REPLACE clause on line 1 is optional. It is included here because triggers, like many application programs, can be created and thus become part of the database schema while still containing logic errors necessitating their re-creation. Since an ALTER TRIGGER statement does not exist, the CREATE OR REPLACE clause allows an existing trigger in need of modification to be dropped (i.e., deleted from the database schema) and recreated from scratch. Had CREATE TRIGGER instead of CREATE OR REPLACE TRIGGER been used, a DROP TRIGGER statement would have to have been used first to drop the erroneous trigger before a CREATE TRIGGER statement could have been used to recreate it.**
2. **This trigger is divided into two sections. The DECLARE section on lines 3-5 defines two variables-[3] (NO_ORDERS and TEMP_MED_NAME) that will subsequently be referenced by the executable statements contained on lines 6-18.**
3. **The first SELECT statement on lines 7-11 searches the ORDERS table and counts the number of orders that involve the medication code for the patient for whom an attempt is being made to insert a new order into the ORDERS table. Since the SELECT statement here is embedded in a host language (in this case PL/SQL), the INTO clause must be used to allow the value returned by the COUNTf*) function to be referenced in statements written in the host language. Observe that the host language variable, NO_ORDERS, that appears in the INTO clause is subsequently referenced by the IF statement on line 12.**

^ These variables (ire technically referred to as host variables.

4. :NEW.ORD_PAT_P_P#A, :NEW.ORD_PAT_P#N, and :NEW.ORD_MED_GODE referenced on lines 9-11 are the values assigned to the **Ord_pat_p#a, Ord_pat_p#n** and **Ord_med_code** columns (columns 2, 3, and 4 of the ORDERS table) in the INSERT statement that is the subject of this trigger. The prefix :NEW is used to refer to the new value associated with the row that caused the execution of the trigger.
5. Lines 12-17 contain a host language IF statement that checks to see if the patient has already received three orders for the same medication (i.e., it checks to see if the host language variable NO_ORDERS is greater than or equal to 3). If the condition is true, then the SELECT statement on lines 13 and 14 retrieves the name of the medication from the MEDICATION table. RAISE_APPLICATION_ERROR is an Oracle procedure which allows custom error-messages to be issued. Its syntax consists of (a) an error number that is a negative integer in the range -20000 to -20999, and (b) an error message. Note how the error message makes use of the patient number associated with the INSERT statement that is the subject of the trigger plus the name of the medication retrieved by the SELECT statements on lines 13 and 14.

To illustrate the execution of the trigger, observe the content of the PATIENT, MEDICATION, and ORDERS tables before and after attempts to insert new rows into the ORDERS table.

Content of Tables Prior to Insertion Attempts

```
SELECT * FROM PATIENT;

Pat_p#a  Pat_p#n  Pat_name        Pat_gender    Pat_age Pat_admit_dt Pat_wing  Pat_room# Pat_bed
-------- -------- --------------- ---------- ---------- ------------ -------- ---------- -------
DB       77642    Davis, Bill     M                  27 2007-07-07   B               108 B
GD       72222    Grimes, David                      44 2007-07-12

SELECT * FROM MEDICATION;

Med_code Med_name            Med_qty_onhand Med_qty_onorder Med_unitprice
-------- ------------------- -------------- --------------- -------------
KEF      Keflin                         400             700             3
VAL      Valium                         500             500          3.33
ASP      Aspirin                       1200             100            .1

SELECT * FROM ORDERS;

Ord_rx#      Ord_pat_p#a  Ord_pat_p#n  Ord_med_code Ord_dosage   Ord_freq
------------ ------------ ------------ ------------ ---------- ----------
104          DB           77642        ASP                   3          1
105          DB           77642        ASP                   2          1
106          DB           77642        ASP                   2          1
108          GD           72222        KEF                   2          1
```

Insertion Attempts

```
INSERT INTO ORDERS VALUES ('109', 'GD', '72222', 'KEF', 2, 3)/

1 row created.


INSERT INTO ORDERS VALUES ('110', 'DB' , '77642', 'ASP', 3, 1);

ERROR-20000: Patient DB77642 has had too much Aspirin


INSERT INTO ORDERS VALUES ('111', 'DB', '77642', 'KEF', 3, 1);

1 row created.
```

Content of Tables After Insertion Attempts

```
SELECT * FROM PATIENT;

Pat_p#a  Pat_p#n  Pat_name         Pat_gender    Pat_age Pat_admit_dt Pat_wing  Pat_room# Pat_bed
-------- -------- ---------------- ---------- ---------- ------------ -------- ---------- -------
DB       77642    Davis, Bill      M                  27 2007-07-07   B               108 B
GD       72222    Grimes, David                       44 2007-07-12

SELECT * FROM MEDICATION;

Med_code Med_name            Med_qty_onhand Med_qty_onorder Med_unitprice
-------- ------------------- -------------- --------------- -------------
KEF      Keflin                         400             700             3
VAL      Valium                         500             500          3.33
ASP      Aspirin                       1200             100            .1

SELECT * FROM ORDERS;

Ord_rx#      Ord_pat_p#a  Ord_pat_p#n  Ord_med_code Ord_dosage   Ord_freq
------------ ------------ ------------ ------------ ---------- ----------
104          DB           77642        ASP                   3          1
105          DB           77642        ASP                   2          1
106          DB           77642        ASP                   2          1
108          GD           72222        KEF                   2          1
109          GD           72222        KEF                   2          3
111          DB           77642        KEF                   3          1
```


12.1.2.4 Combining Trigger Types and Customizing Error Conditions

Triggers for insert, update, and delete statements on a table can be combined into a single trigger provided they are all at the same level (row-level or statement-level). The following example shows a statement-level trigger that is executed whenever an insertion, deletion, or update is attempted on the PATIENT table. Its purpose is to check two system conditions: that the day of the week is neither Saturday nor Sunday (i.e., the PATIENT table is available for only querying on these days), and that the username of the employee performing the insertion, update, or deletion begins with the letters "ACCTG". Once again, the individual lines have been numbered to facilitate the discussion of those statements or features not covered in the two previous examples.

BEFORE INSERT, UPDATE, OR DELETE Trigger Example

```
1.    CREATE OR REPLACE TRIGGER PATIENT_BEF_INS_UPD_DEL
2.     BEFORE INSERT OR UPDATE OR DELETE ON PATIENT
3.    DECLARE
4.      WEEKEND_ERROR EXCEPTION;
5.      NOT_ACCTG_USER EXCEPTION;
6.    BEGIN
7.      IF TO_CHAR(CURRENT_DATE, 'DY') = 'SAT' OR
8.        TO_CHAR(CURRENT_DATE, 'DY') = 'SUN' THEN
9.      RAISE WEEKEND_ERROR;
10.     END IF;
11.     IF SUBSTR(USER,1,5) <> UPPER{'ACCTG') THEN
12.       RAISE NOT_ACCTG_USER;
13.     END IF;
14.   EXCEPTION
15.       WHEN WEEKEND_ERROR THEN
16.          RAISE_APPLICATION_ERROR(-20001, ' PATIENT table not available on weekends');
17.       WHEN NOT_ACCTG_USER THEN
18.          RAISE_APPLICATION_ERROR(-20002, 'Not authorized to access PATIENT table');
19.   END;
```


1. **Line 2 illustrates the use of the OR keyword to demonstrate the use of this trigger to fire before each possible action on the PATIENT table. As a statement-level trigger, this trigger is fired before the execution of an action on one row or to many rows in the PATIENT table.**
2. **The DECLARE section on lines 3-5 is used to define two host language (PL/SQL) variables with an exception data type. The exception data type allows the variables WEEKEND_ERROR and NOT_ACCTG_USER to function as user-defined exception handlers that will subsequently be referenced by the executable statements on lines 6-19.**
3. **The TO_CHAR (d [,fmt]) function is used in Oracle to extract different parts of a date/time and convert them to a character string using the format specified by the format element *fmt.* The purpose of the TO_CHAR (CURRENT. DATE, 'DY') function on line 7 is to extract from the current date the day of the week and compare its value with 'SAT'. Section 12.3 contains additional information about the TO_CIIAR function.**
4. **The RAISE statement on line 9 stops normal execution of the PL/SQL block and transfers control to the exception handler \VEEKEND ERROR on lines 15 and 16.**
5. **The keyword, USER, on line 11 refers to the name of the current Oracle user. Its use with the SUBSTR function makes it possible for the userid to be checked. The UPPER function allows for a character string or column to be converted to upper case. The similar function LOWER converts a character string or column to lower case. Functions such as the UPPER and SUBSTR function are also discussed in Section 12.2.**
6. **A PL/SQL block can consist of up to three sections:**
   - **an optional declaration section that begins with the DECLARE header,**
   - **an executable section which starts with the word BEGIN, and**
   - **an optional exception section which consists of series of statements to respond to exceptions.**

The AFTER DELETE trigger example in Section 12.1.2.2 consists of only one section—the required executable section and the BEFORE INSERT trigger example in Section 12.1.2.3 contains both the required executable section and the optional declare section. Lines 14-18 of this trigger illustrate an example of a trigger with all three sections. The exception section is made up of one or more exception handlers which consist of a WHEN clause specifying an exception name and a sequence of statements to be executed to handle the error when the exception is raised. The execution of this sequence of statements completes the execution of the block (and therefore the trigger). The RAISE_APPLICATION_ERROR procedure is used to display customized error messages should one of the two system conditions be violated.

The following series of INSERT statements test various conditions associated with this trigger.

Content of PATIENT Table Prior to Tests

```
SELECT * FROM PATIENT;

Pat_p#a  Pat_p#n  Pat_name         Pat_gender    Pac_age Pat_adinit_dt Pat_wing   Pat_room Pat_bed

DB       77642    Davis, Bill      M                27 2007-07-07   B              108 B
GD       72222    Grimes, David                     44 2007-07-12
```


Test 1. Assume Date of Saturday, August 25, 2007 and Accounting User

```
INSERT INTO PATIENT VALUES ('SJ', '12345', 'Su, John', 'M', 30, '2007-08-25', NULL, NULL, NULL);
INSERT INTO PATIENT VALUES ('SJ', '12345', 'Su, John', 'M\ 30, '2007-08-25', NULL, NULL, NULL)

ERROR at line 1:
ERROR-20001: PATIENT table not available on weekends
```

Test 2. Assume Date of Friday, August 24, 2007 and Non-Accounting User

```
INSERT INTO PATIENT VALUES ('LB', •23451', 'Li, Belle', 'F', 27, '2007-08-25', NULL, NULL, NULL);
INSERT INTO PATIENT VALUES ('LB', •23451', 'Li, Belle', 'F', 27, '2007-08-25', NULL, NULL, NULL)

ERROR at line 1:
ERROR-20002: Not authorized to access PATIENT table
```

Test 3. Assume Date of Friday, August 24, 2007 and Accounting User

```
INSERT INTO PATIENT VALUES ('LS', '30232', 'Li, Sue', 'F', 23, '2007-08-25', NULL, NULL, NULL);

1 row created.

SELECT * FROM PATIENT;

Pat_p#a  Pat_p#n  Pat_namc         Pat_gender    Pat_age Pat_admit_dt Pat_wing   Pat_room Pat_bed

DB       77642    Davis, Bill      M                27 2007-07-07   B              108 B
LS       30232    Li, Sue          F                23 2007-08-25
GD       72222    Grimes, David                     44 2007-07-12
```

While individual triggers are usually relatively easy to understand, when combined with declarative integrity checking, trigger execution can be complex because the actions of one SQL statement may cause several triggers to fire. Mannino (2007, pp. 414-415) provides a helpful description of Oracle's simplified trigger execution procedure.

1. **Execute the applicable BEFORE statement-level triggers**
2. **For each row affected by the SQL data manipulation statement:**
   - **2.1 Execute the applicable BEFORE row-level triggers**
   - **2.2 Perform the data manipulation operation on the row**
   - **2.3 Perform integrity constraint checking**
   - **2.4 Execute the application AFTER row-level triggers.**
3. **Perform deferred integrity constraint checking.[4]**
4. **Execute the applicable AFTER statement-level triggers.**

The remainder of this section constitutes a simple illustration of how declarative constraints and triggers work together to enforce data integrity. Let's begin by assuming that PATIENT, MEDICATION, and ORDERS tables exist as defined by the three earlier CREATE TABLE statements and that the three tables contain the following data.


```
SELECT * FROM PATIENT;
```

Pat_p#a	Pat_p#n	Pat_name	Pat_gender	Pat_age	Pat_admit_dt	Pat_wing	Pat_room	Pat_bed
DB	77642	Davis, Bill	M	27	2007-07-07	B	108	B
LS	12345	Li, Sue	F	23	2007-08-25			
ZZ	06912	Zhang, Zhaoping	F	35	2007-08-11			
GD	72222	Grimes, David		44	2007-07-12			

```
SELECT * FROM MEDICATION;
```

Med_code	Med_name	Med_qty_onhand	Med_qty_onorder	Med_unitprice
KEF	Keflin	400	700	3
VAL	Valium	501	500	3.33
ASP	Aspirin	1200	100	.1

```
SELECT * FROM ORDERS;
```

Ord_rx#	Ord_pat_p#a	Ord_pat_p#n	Ord_med_code	Ord_dosage	Ord_freq
104	DB	77642	ASP	3	1
105	DB	77642	ASP	2	1
106	DB	77642	ASP	2	1
108	GD	72222	KEF	2	1
109	GD	72222	KEF	2	3
111	DB	77642	KEF	3	1

[4] Deferred integrity constraint checking involves enforcing integrity constraints at the end of a transaction rather than after each data manipulation statement. Recall from Section 6.7.1.2.3. that using mutual-referencing to enforce a **1:1** relationship with total participation of each entity type necessitates deferring the execution of at least one of the two referential constraints until run time.

Next, let's create the following after insert trigger on the ORDERS table reducing the quantity on hand of the medication associated with a new order by the number of doses ordered. Once again, the individual lines have been numbered to facilitate the discussion of the trigger and the introduction of new PL/SQL statements.

```
1.  CREATE OR REPLACE TRIGGER ORDERS_AFT_INS_ROW
2.  AFTER INSERT ON ORDERS FOR EACH ROW
3.  DECLARE
4.  TOTAL_AVAILABLE NUMBER;
5.  NAME_OF_MEDICATION VARCHAR(20);
6.  BEGIN
7.      UPDATE MEDICATION
8.      SET MEDICATION.MED_QTY_ONHAND = MEDICATION.MED_QTY_ONHAND - :NEW.ORD_DOSAGE
9.      WHERE MEDICATION.MED_CODE = :NEW.ORD_MED_CODE;
10.
11.     SELECT MEDICATION.MED_QTY_ONHAND, MED_NAME INTO TOTAL_AVAILABLE, NAME_OF_MEDICATION
12.     FROM MEDICATION
13.     WHERE MEDICATION.MED_C0DE = :NEW.ORD_MED_CODE;
14.     DBMS_OUTPUT.PUT_LINE ('Quantity on hand for '||NAME_OF_MEDICATION||' is now '||
15.           TOTAL_AVAI LABLE || ' doses.');
16.
17.  END;
```


1. The DECLARE section on lines 3-5 defines two host language (PL/SQL) variables (TOTAL_AVAILABLE and NAME_OF_MEDICATION) that will subsequently be referenced on lines 11, 14, and 15.
2. The UPDATE statement on lines 7-9 updates the quantity on hand for the medication referenced in the INSERT statement that causes the trigger to be fired. The :NEW.ORD_DOSAGE and :NEW.ORD_MED_CODE represent the code for the medication code and dosage amount in the INSERT statement. Upon execution of this UPDATE statement, the quantity on hand for the medication code is reduced by the value stored in :NEW.ORD_DOSAGE.
3. After updating the MEDICATION table, the SELECT statement on lines 11-13 references the row just updated, and the quantity on hand and name of the medication are stored in the PL/SQL variables TOTAL_AVAILABLE and NAME_OF_MEDICATION. Lines 14 and 15 contain a I)BMS_OUTPUT.PUT_LINE statement. The purpose of the DBMS_OUTPUT.PUT_LINE statement is to display the content of PL/SQL variables and constants and is used in this trigger to display the updated quantity on hand for the medication ordered.

A second trigger was also created and is shown below. The purpose of this statement-level after insert or update or delete trigger on the MEDICATION table is simply to display a message that records a successful insertion, update, or deletion to the MEDICATION table. The words INSERTING, UPDATING (and also DELETING) arc reserved words in PL/SQL that represent valid transaction types.

```
CREATE OR REPLACE TRIGGER MEDICATION_AFT_INS_UPD_DEL
AFTER INSERT OR UPDATE OR DELETE ON MEDICATION
,BEGIN
   IF INSERTING THEN
   DBMS_OUTPUT.PUT_LINE  ('SUCCESSFUL INSERTION INTO MEDICATION TABLE');
   ELSE
     IF UPDATING THEN
         DBMS_OUTPUT.PUT_LINE  ('SUCCESSFUL UPDATE OF MEDICATION TABLE');
     ELSE
         DBMS_OUTPUT. PUT_LINE  ('SUCCESSFUL DELETION TO MEDICATION TABLE');
     END IF;
     END IF;
END;
```

Thus, at this point, a total of five triggers exist in the database schema:

- **A row-level after delete trigger on the PATIENT table (see Section 12.1.2.2)**
- **A row-level before insert trigger on the ORDERS table (see Section 12.1.2.3)**
- **A statement-level before, insert, or update trigger on the PATIENT table (see Section 12.1.2.4)**
- **A row-level after insert trigger on the ORDERS table (defined towards the beginning of this section)**
- **A statement-level before, insert, or update trigger on the MEDICATION table (defined immediately above)**

Test 1. Patient Sue Li places an order for two doses of the medication Valium.

```
INSERT INTO ORDERS VALUES ('112', 'LS', '12345', 'VAL', 2, 1);
INSERT INTO ORDERS VALUES ('112', 'LS', '12345', 'VAL', 2, 1)
*
ERROR at line 1:
ORA-02290: check constraint (CHAPTER11.CHK_QTY) violated
ORA-06512: at "CHAPTER11.ORDERS_AFT_INS_ROW", line 5
ORA-04088: error during execution of trigger 'CHAPTER11.ORDERS_AFT_INS_ROW'
```

A statement-level before, update, and delete trigger exists on the PATIENT table but not on the ORDERS table. Had a similar trigger existed on the ORDERS table, a check would have been made to make sure that the day of the week was neither Saturday nor Sunday and that the user name begins with the letters 'ACCTG'. Thus in this case, no applicable BEFORE statement-level trigger exists. However, a row-level before insert trigger on the ORDERS table does exist and evaluates whether or not Sue Li has already received three orders for the medication Valium. Since she has not, the insert operation is allowed to begin and a new row is inserted in the ORDERS table. At this point the row-level AFTER INSERT Trigger on the ORDERS table is executed and an UPDATE statement is issued to update the MEDICATION.Med_qty_onhand for MEDICATION.Med_code VAL. This update statement causes the declarative CHK_QTY constraint that requires MEDICATION. Med_qty_onhand + MEDICATION.Med_qty_onorder to be between 1000 and 3000 to be

evaluated and the transaction fails as a result of the violation of this constraint. At the end of the failed transaction, the content of the MEDICATION and ORDERS tables is as follows:

```
SELECT * FROM MEDICATION;

Med_code Med_name            Med_qty_onhand Med_qty_onorder Med_unitprice
-------- ------------------- -------------- --------------- -------------
KEF      Keflin                         400             700             3
VAL      Valium                         501             500          3.33
ASP      Aspirin                       1200             100            .1

SELECT * FROM ORDERS;

Ord_rx#      Ord_pat_p#a  Ord_pat_p#n  Ord_med_code Ord_dosage   Ord_freq
------------ ------------ ------------ ------------ ---------- ----------
104          DB           77642        ASP                   3          1
105          DB           77642        ASP                   2          1
106          DB           77642        ASP                   2          1
108          GD           72222        KEF                   2          1
109          GD           72222        KEF                   2          3
111          DB           77642        KEF                   3          1
```


Test 2. Patient Sue Li places an order for one dose of the medication Valium.

```
INSERT INTO ORDER_PATIENT VALUES ('113', 'LS', '12345', 'VAL', 1, 1);
SUCCESSFUL UPDATE OF MEDICATION TABLE
Quantity on hand for Valium is now 500 doses.

1 row created.
```

Once again, no applicable BEFORE statement-level trigger exists and the BEFORE row-level trigger on the ORDERS table verifies that Sue Li has not received three orders for the medication Valium. This allows the insert operation to begin and a new row is inserted in the ORDERS table. At this point, the row-level AFTER INSERT Trigger on the ORDERS table is executed and an UPDATE statement is issued to update the MEDICATION.Med_qty_onhand for MEDICATION.Med_code VAL. This update statement does not cause the declarative CIIK_QTY constraint to be violated and thus the transaction is successful, and DBMS_OUTPUT.PUT_LINE statement indicates that the quantity on hand for Valium is now 500 doses. Since no deferred constraint checking is required, the applicable AFTER statement-level trigger on the MEDICATION table is executed. At the end of this transaction, the content of the MEDICATION and ORDERS tables is as follows:

```
SELECT * FROM MEDICATION;

Med_code Med_name            Med_qty_onhand Med_qty_onorder Med_unitprice
-------- ------------------- -------------- --------------- -------------
KEF      Keflin                         400             700             3
VAL      Valium                         500             500          3.33
ASP      Aspirin                       1200             100            .1

SELECT * FROM ORDERS;

Ord_rx#      Ord_pat_p#a  Ord_pat_p#n  Ord_med_code Ord_dosage   Ord_freq
------------ ------------ ------------ ------------ ---------- ----------
104          DB           77642        ASP                   3          1
```

```
105     DB      77642     ASP     2     1
106     DB      77642     ASP     2     1
108     GD      72222     KEF     2     1
109     GD      72222     KEF     2     3
111     DB      77642     KEF     3     1
113     LS      12345     VAL     1     1
```

Test 3. Increase the price of each medication by 10 percent.

```
UPDATE MEDICATION SET MED_UNITPRICE = MED_UNITPRICE*1.10;
SUCCESSFUL UPDATE OF MEDICATION TABLE
```

No constraints prohibit updating the prices of these medications since the Chk_unitprice constraint only requires MEDICATION.Med_unitprice to be less than $4.50. Note, however, the applicable AFTER statement-level trigger on the MEDICATION table is executed once again. At the end of this transaction, the content of the MEDICATION table is as follows.

```
SELECT * FROM MEDICATION;
```


Med_code	Med_name	Med_qty_onhand	Med_qty_onorder	Med_unitprice
KEF	Keflin	400	700	3.3
VAL	Valium	500	500	3.66
ASP	Aspirin	1200	100	.11

Had an attempt been made to increase the price of each medication by 50 percent, the presence of the Chk_unitprice constraint would have caused the transaction to fail.

```
UPDATE MEDICATION SET MED_UNITPRICE = MED_UNITPRICE * 1.50;
UPDATE MEDICATION SET MED_UNITPRICE = MED_UNITPRICE * 1.50
*
ERROR at line 1:
check constraint (CHAPTER11.CHK_UNITPRICE) violated

SELECT * FROM MEDICATION;
```

Med_code	Med_name	Med_qty_onhand	Med_qty_onorder	Med_unitprice
KEF	Keflin	400	700	3.3
VAL	Valium	500	500	3.66
ASP	Aspirin	1200	100	.11

As stated above, this illustration is simplified and does not consider the possibility of overlapping triggers (two or more triggers with the same timing, granularity, and table). In addition, data manipulation statements in a trigger itself also complicate the description given above and may cause other triggers to fire. Kifer, et al. (2005) and Mannino (2007) each contain a good discussion of these and other issues associated with triggers.

12.1.3 Specifying Views in SQL/DDL

Views were introduced in Chapter 6 (see Section 6.4). This section outlines SQL support for defining and implementing views. An SQL view is a *single* table that is derived based on a relational expression involving one or more other base tables and/or views. In contrast to a base

table that has physical existence as a file, a view does not have a physical form. It is an "empty shell" in a physical sense and so is referred to as a virtual table. A view is essentially a lens to look at data stored in base tables. Therefore, there are specific limitations in updating data in a database through views, while there are no such limitations on retrieving data (i.e., querying) through views. The tables that serve as the input to the relational expression defining a view are called defining tables. A view in a database environment is a convenience. A user can issue queries on a view as if it is a single table retrieval task instead of approaching the query as a multiple table retrieval operation involving a complex relational expression. Executing the retrieval operation itself entails converting the view definition (i.e., the relational expression defining the view) into equivalent operations on the base tables which are done by the DBMS. To that extent, there is a performance overhead associated with view processing. Therefore, indiscriminate use of views has database efficiency and performance implications.

A view is defined using the SQL/DDL statement, CREATE VIEW, with syntax as follows:

```
CREATE VIEW view_name [ (comma delimited list of columns) ]
AS relational expression [ WITH [CASCADED [ LOCAL ] CHECK OPTION ];
```

where:

- *view_name* is a user supplied name for the view
- *comma delimited list of columns* is the unqualified names of the columns of the view
- *relational expression* defines the scope of the view in terms of an SQL query
- CHECK OPTION is an integrity constraint applicable only to SQL-updatable views

Several examples of defining a view in SQL follow.

View Example 1. *A simple view that results from a combination of the relatioiial algebra operations of SELECTION and PROJECTION looks like:*

```
CREATE VIEW senior_citizen AS
  SELECT patient.Pat_name, patient.Pat_age, patient.Pat_gender
  FROM patient
  WHERE patient.Pat_age > 64;
```

The view definition above constructs a view structure that contains the three columns **Pat_name, Pat_age,** and **Pat_gender** from the PATIENT table and yields a result that contains only the rows of patients that are 65 or older from the PATIENT table. The name of the view (virtual table) is senior_citizen and the ordered column names in the view are, by default, the same as the unqualified name of the columns from source table—i.e., **Pat_name, Pat_age, Pat_gender.** The option to explicitly specify column names for the view is shown clearly in the syntax for view definition. The only time it is mandatory to specify the column name of a view is when the column results from applying a function or arithmetic operations, as in the next example.

View Example 2. *A view that shows the gender (male or female) and the number of patients in each gender that are 65 or older.*

```
CREATE VIEW senior_stat (V_gender, V_#ofpats) AS
  SELECT patient.Pat_gender, count (*)
  FROM patient
  WHERE patient.Pat_age > 64
  GROUP BY patient.Pat_gender;
```

View examples 1 and 2 use a query on a single table as the relational expression to derive a view. The third example involves more than one table in the relational expression, but the data retrieved for the view is from a single table.

View Example 3. *A view that exhibits a list of medications that are not ordered by any patient—the medication name and the quantity on hand are included in the list.*

```
CREATE VIEW unused_med AS
  SELECT medication.Med_name, medication.Med_code,
  medication.Med_qty_onhand
  FROM medication
  WHERE medication.Med_code NOT IN
    (SELECT orders.Ord_med_code FROM orders)
WITH CHECK OPTION;
```


The CHECK OPTION in a view definition applies exclusively to SQL-updatable tables and cannot be specified in a view definition if the view is not SQL-updatable. When included in the view definition, the CHECK OPTION makes sure that any row insertion or update of a column value in the view does not violate the view-defining condition. Thus, a CHECK OPTION should always be specified for an SQL-updatable view; yet, SQL does not enforce such a rule (Date and Darwen, 1997). The CASCADED I LOCAL choice is relevant only when a view definition is specified using another view instead of from base table(s). Note that any update on a view derived directly from a base table(s) is most definitely checked against the integrity constraints of the base table(s) irrespective of the specification of the CHECK OPTION in the view definition.

The next example entails retrieval of data for a view from more than one table.

View Example 4. *A view that lists patient number, names of all the medications used by the patient along with dosage and frequency.*

```
CREATE VIEW used_med AS
  SELECT orders.Ord_pat_p#a, orders.Ord_pat_p#n,
  medication.Med_name, orders.Ord_dosage, orders.Ord_frequency
  FROM medication, orders
  WHERE medication.Med_code = orders.Ord_med_code;
```

As noted earlier, views are mainly intended as a convenient mechanism for information retrieval. SQL offers very limited support for updating base tables via views. In general, updating a database through a view is usually (but not always) feasible when the view is defined on a *single table without any aggregate functions.* Since an update of a view that involves a join of multiple base tables can be mapped to update operations on the underlying tables in multiple ways, it introduces a lot of complications and ambiguities. SQL standards for view update, in general, are quite restrictive and complicated. Therefore, we do not treat this topic in any depth in this book. The reader is directed to one of the references (e.g., Date, 2004) in the selected bibliography of this chapter for further discussion of this topic.

12.1.4 The Division Operation

Recall that the Division operation is useful when there is a need to identify tuples in one relation that match all tuples in another relation, and is described in Chapter 11 as capable of being expressed as a sequence of Projection. Cartesian product, and Difference operations. Let's consider this description in the context of the following question which originally appeared in Chapter 11, referring to the Madeira College tables: *Wliat are the course numbers and course names of those courses offered in all quarters during which sections are offered?* **The following section shows how SQL views can be used to simulate the Division operation. For convenience of the reader, the content of the COURSE and SECTION tables follows.**

COURSE Table

Co_name	Co_course#	Co_credit	Co_college	Co_hrs	Co_dpt_dcode
Intro to Economics	15ECON112	U	Arts and Sciences	3	1
Operations Research	22QA375	U	Business	2	3
Intro to Economics	18ECON123	U	Education	4	4
Supply Chain Analysis	22QA411	U	Business	3	3
Principles of IS	22IS270	G	Business	3	7
Programming in C++	20ECES212	G	Engineering	3	6
Optimization	22QA888	G	Business	3	3
Financial Accounting	18ACCT801	G	Education	3	4
Database Concepts	22IS330	U	Business	4	7
Database Principles	22IS832	G	Business	3	7
Systems Analysis	22IS430	G	Business	3	7

SECTION Table

Se_section#	Se_qtr	Se_year	Se_time	Se_maxst	Se_room	Se_co_course#	Se_pr_profid
101	A	2003	T1015	25		22QA375	HT54347
901	A	2002	W1800	35	Rhodes 611	22IS270	SK85977
902	A	2002	H1700	25	Lindner 108	22IS270	SK85977
101	S	2002	T1045	29	Lindner 110	22IS330	SK85977
102	S	2002	H1045	29	Lindner 110	22IS330	CC49234
701	W	2003	M1000	33	Braunstien 211	22IS832	CC49234
101	A	2003	W1800		Baldwin 437	20ECES212	RR79345
101	U	2003	T1015	33		22QA375	HT54347
101	A	2003	H1700	29	Lindner 108	22IS330	SK85977
101	S	2003	T1015	30		22QA375	HT54347
101	W	2003	T1015	20		22QA375	HT54347

12.1.4.1 Use of Views to Simulate the Division Operation

The Division operation that appears in Chapter 11 can be simulated by creating two views, Tl and T 2 , in SQL. View Tl creates a virtual table that contains the quarters during which sections of courses are offered, while view T2 begins by forming the product of the SECTION table and the Tl view (shown below with duplicate tuples crossed out) and then subtracts the projection of the SECTION table, which records the sections offered during various quarters.

```
CREATE VIEW T1 AS SELECT DISTINCT SECTION.SE_QTR FROM SECTION;

CREATE VIEW T2 AS
```

```
SELECT SECTION.SE_CO_COURSE#, T1.SE_QTR
FROM SECTION CROSS JOIN T1
MINUS
'SELECT SECTION.SE_CO_COURSE#, SECTION.SE_QTR
FROM SECTION;
```

Se_co_course#	Se_qtr
22QA375	A
22IS270	A
~~22IS270~~	~~A~~
22IS330	A
~~22IS330~~	~~A~~
22IS832	A
20ECES212	A
~~22QA375~~	~~A~~
~~22IS330~~	~~A~~
~~22QA375~~	~~A~~
~~22QA375~~	~~A~~
22QA375	S
22IS270	S
~~22IS270~~	~~S~~
22IS330	S
~~22IS330~~	~~S~~
22IS832	S
20ECES212	S
~~22QA375~~	~~S~~
~~22IS330~~	~~S~~
~~22QA375~~	~~S~~
~~22QA375~~	~~S~~
22QA375	U
22IS270	U
~~22IS270~~	~~U~~
22IS330	U
~~22IS330~~	~~U~~
22IS832	U
20ECES212	U
~~22QA375~~	~~U~~
~~22IS330~~	~~U~~
~~22QA375~~	~~U~~
~~22QA375~~	~~U~~
22QA375	W
22IS270	W
~~22IS270~~	~~W~~
22IS330	W
~~22IS330~~	~~W~~
22IS832	W
20ECES212	W
~~22QA375~~	~~W~~
~~22IS330~~	~~W~~
~~22QA375~~	~~W~~
~~22QA375~~	~~W~~

← Projection of product of SECTION table and T1 view. Duplicates resulting from courses offered more than once in a given quarter are crossed out. Note that 20 unique Se_co_course# and Se_qtr combinations exist.

Se_co_course#	Se_qtr
22QA375	A
22IS270	A
22IS270	A

← Projection on SECTION table over Se_co_course# and Se_qtr.

```
22IS330       S
22IS330       S
22IS832       W
20ECES212   • A
22QA375       U
22IS330       A
22QA37 5      S
22QA375       W
```

```
Se_co_course# Se_qtr   Content of T2 view recording quarters during which each
_____________________course is not offered.
20ECES212     S
20ECES212     U
20ECES212     W
22IS270       S
22IS270       U
22IS270       W
22IS330       U
22IS832       A
22IS832       S
22IS832       U
```


Using T2, the course(s) offered during all quarters can be determined using the following nested subquery. The individual lines have been numbered to facilitate the discussion of the execution of the query.

```
1.  SELECT COURSE.C0_C0URSE#, COURSE.C0_NAME
2.  FROM COURSE
3.  WHERE COURSE.CO_COURSE# IN
4.        (SELECT SECTION.SE_CO_COURSE# FROM SECTION
5.       WHERE SECTION.SE_C0_COURSE# NOT IN
6.             (SELECT T2 . SE_CO_COURSE#
7 .             FROM T2) ) ;

Co_course# Co_name

22QA375    Operations Research
```

A nested subquery[5] is a complete query nested in the SELECT, FROM, HAVING, or WHERE clause of another query. Nested subqueries of the type shown here are executed from the bottom up.[6] In other words, the result of the query on lines 6 and 7 is used in the WHERE clause in the query on lines 4 and 5 and the result of the query on lines 4 and 5 is used in the WHERE clause in the query on lines 1 through 3. It is required that T2.Se_co_course# on line 6 share the same domain as SECTION.Se_co_course# referenced in line 5, and likewise that SECTION.Se_co_course# on line 4 share the same domain as COURSE.Co_course# referenced on line 3.

Execution of this query begins by retrieving the values of course numbers associated with view T2 (20ECES212, 22IS270, 22IS330, 22IS832). Next, the query on lines 4 and 5 identifies values of the course numbers in the SECTION table that are not associated with

[5] Subqueries are discussed in greater detail in Section **11.2**.3.

[6] The subqueries in this example take the form of an uneorrelated subquery. The execution of a correlated subquery is discussed in Section **11.2**.3.**2.**

T2. Finally, the query on lines 1 through 3 verifies that the value(s) of the course numbers retrieved by the query on lines 4 and 5 actually exist in the COURSE table.

12.1.4.2 A Second Query to Simulate the Division Operation

The following SQL SELECT statement can also be used to display the course(s) offered during all quarters. Three steps are required to execute this statement. First, the number of distinct SECTION.Se_qtr values in the SECTION table is determined by the SELECT statement:

```
(SELECT DISTINCT(COUNT(DISTINCT(SECTION.SE_QTR))) FROM SECTION)
```

Second, the COURSE and SECTION tables are joined on their course number attributes that share the same domain. This join yields a total of 11 rows. Next, the rows associated with the result of the join are logically grouped by the combination of COURSE.Co_course# and COURSE.Co_name with the HAVING used to identify the subset of groups we want to consider. Finally, since one course (course number 22QA375) is associated with all four quarters, only one course number is displayed, that for course number 22QA375.

SQL SELECT Statement:

```
SELECT DISTINCT COURSE.CO_COURSE#, COURSE.CO_NAME
FROM COURSE JOIN SECTION
ON COURSE.CO_COURSE# = SECTION.SE_CO_COURSE#
HAVING COUNT(*) =
       (SELECT DISTINCT(COUNT(DISTINCT(SECTION.SE_QTR)))
       FROM SECTION)
GROUP BY COURSE.CO_COURSE#, COURSE.CO_NAME;
```

```
Co_course#  Co_name

22QA3 7S    Operations Research
```

Section 11.2.3.2 in Chapter 1 1 contains a third approach for representing the Division operation in an SQL SELECT statement.

12.2 SQL-92 Built-in Functions

A built-in function can be used in an SQL expression anywhere that a constant of the same data type can be used. A large number of built-in functions are supported by popular SQL implementations. Four of these functions (i.e., CURRENT_DATE(), TO_CHAR(), SUBSTR(), UPPERQ) were used in the triggers discussed in Sections 12.1.2.2 and 12.1.2.4. As Groff and Weinberg (2002) report, the SQL-92 standard incorporates what have been judged to be the most useful of these built-in functions, in many cases with slightly different syntax. Column 1 of Table 12.1 contains the names of the most widely-used of the 20 built-in functions that comprise the SQL-92 standard.[7]

[7] For a complete list of SQL-92 functions, see Groff and Weinberg (2002).

Table 12.1 Selected SQL-92 built-in functions and their Oracle and MySQL equivalents

SQL-92 Function	Oracle Implementation	MySQL Implementation	Sections Containing Useful Examples in Chapter 12
CASE	DECODE	Function not available	12.4
CHAR_LENGTH	LENGTH	LENGTH**	12.2.2
Concatenation	CONCAT	CONCAT**	12.2.1
CURRENT_DATE	CURRENT_DATE	CURRENT_DATE or CURDATE()	12.3
CURRENT_TIME	CURRENT_DATE (With Mask)	CURRENT_TIME or CURTIME()	12.3
CURRENT_USER	USER	CURRENT_USER	12.1.2.4
Date/Time Conversions	TO_CHAR, TO_DATE	See below*	12.3
EXTRACT	EXTRACT	EXTRACT	
LOWER and UPPER	LOWER and UPPER	LOWER and UPPER	12.1.2.4
POSITION	INSTR	INSTR** or LOCATE**	12.2.5
SUBSTRING	SUBSTR	SUBSTRING**	12.2.1
TRANSLATE	TRANSLATE	Function not available	12.2.4
TRIM	LTRIM and RTRIM	LTRIM and RTRIM**	12.2.3

*MySQL supports a series of functions to extract date and time elements.
**Both the Oracle and MySQL implementation of these functions are covered in this text to illustrate the similarities and differences in how they are implemented across products. For additional information on these as well as on other SQL-92 functions not covered in this book, the reader is encouraged to refer to the considerable product-specific documentation available online and in books.

This section covers the use of many of these functions using a variety of short examples in the context of the SQL SELECT statement. In order to illustrate some of the differences in syntax and functionality between products, the examples employ the syntax of the Oracle and MySQL implementations of SQL. Note that Oracle's SQL requires use of the FROM keyword in every SQL SELECT statement. Thus, many of the Oracle SQL examples in this section make use of the DUAL table. The DUAL table has one column, DUMMY GHAR(l), and one row with a value of 'X'. As we will see, the DUAL table is useful when a SELECT statement is issued to display data that does not exist in a table. Tt is particularly useful when you want to display a numeric or character literal in a SELECT statement. On the other hand, MySQL's SQL does not require use of the FROM keyword.

Columns 2 and 3 of Table 12.1 contain names of the SQL-92 standard built-in functions used by Oracle and MySQL while column 4 contains the section numbers in Chapter 12 where examples of the use of these functions can be found.

The purpose of illustrating both the Oracle and MySQL implementation of some of the SQL-92 functions in this section is to sensitize the reader to the differences in syntax and sometimes functionality across database platforms. In short, the material in this chapter, along with the material in Chapters 10 and 11, is intended to be a highly useful but not necessarily standalone reference to SQL- As such, the reader may need to supplement the material in this textbook with product-specific documentation.

1221 The SUBSTRING Function

The purpose of the SUBSTRING function is to extract a substring from a given string. The format of the SUBSTRING function in the SQL-92 standard is:

SUBSTRING *(source* FROM *n* FOR *len)*

where:

- ***n* indicates the character position where the search begins**
- ***len* represents the length of the search.**

Oracle implements the SUBSTRING function with the SUBSTR (char, m [,n]) function which returns a portion of *char,* beginning at character *m, n* characters long (if *n* is omitted, to the end of *char).* The first position of *char* is 1. Floating point numbers passed as arguments to SUBSTR are automatically converted to integers. MySQL implements the substring function as SUBSTRING (string FROM start FOR length).

The SELECT statement in SUBSTRING Example *1* goes into the character string ABCDEFG' beginning at character position 3 and returns the next four characters thus displaying 'CDEF.'

SUBSTRING Example 1

```
SELECT SUBSTR('ABCDEFG',3,4) "Substring" FROM DUAL;
SELECT SUBSTRING('ABCDEFG' FROM 3 FOR 4);
```

RESULT:' CDEF

As shown in Examples 2 and 3, if the position where the search begins is not an integer, the function's value is truncated using Oracle SQL and rounded using MySQL.

SUBSTRING Example 2

```
SELECT SUBSTR('ABCDEFG',3.1,4) "Substring" FROM DUAL;
SELECT SUBSTRING('ABCDEFG' FROM 3.1 FOR 4);
```

RESULT: CDEF

SUBSTRING Example 3

```
SELECT SUBSTR('ABCDEFG',3.7,4) "Substring" FROM DUAL;
SELECT SUBSTRING ('ABCDEFG' FROM 3.7 FOR 4);
```

'' The output of eaeh function is accompanied by a column heading. Since the format used to display column headings varies by function and by product, only the value returned by the function is displayed in this section. In addition, unless the value returned by the function differs between Oracle and MySQL, only one result is shown.

```
Oracle RESULT:  CDEF
MySQL RESULT:   DEFG
```

The position where the search begins can also be a negative number. In this case, characters beginning with the rightmost characters in the string are stripped off. As expected, the SELECT statement in Example 4 illustrates that a value of -5 for the starting position of the search produces the same result as when the starting position of the search has a value of 3.

SUBSTRING Example 4

```
SELECT SUBSTR('ABCDEFG',-5,4) "Substring" FROM DUAL;
SELECT SUBSTRING ('ABCDEFG ' FROM -5 FOR 4);
```

```
RESULT:  CDEF
```

As shown in Example 5, if the number of characters to be searched is omitted, the number of characters returned extends to the end of the string.

SUBSTRING Example 5


```
SELECT SUBSTR('ABCDEFG',-1)  FROM DUAL;
SELECT SUBSTRING('ABCDEFG' FROM -1);
```

```
RESULT:  G
```

Observe that if the position where the search begins is a negative value and the number of characters to be stripped off is greater than the absolute value of *value where the search begins,* **the number of characters returned extends only to the end of the string (see Example 6).**

SUBSTRING Example 6

```
SELECT SUBSTR ('ABCDEFG',-1, 3) FROM DUAL;
SELECT SUBSTRING('ABCDEFG' FROM -1 FOR 3);
```

```
RESULT:  G
```

However, as indicated in Examples 7 and 8, if the number of characters to be stripped off is zero or negative, a null value is displayed for the result.

SUBSTRING Example 7

```
SELECT SUBSTR ('ABCDEFG',-1, 0) FROM DUAL;
SELECT SUBSTRING)'ABCDEFG' FROM -1 FOR 0);
```

```
RESULT:  null value
```

SUBSTRING Example 8

```
SELECT SUBSTR ('ABCDEFG',-1, -5) FROM DUAL;
SELECT SUBSTRING('ABCDEFG' FROM -1 FOR -5);
```

```
RESULT:  null value
```

In Oracle, the SUBSTR function is often used in conjunction with the concatenation operator (II). Example 9a illustrates their use together in displaying the name and phone number of all professors with phone numbers that end with two digits ranging between 45 and 65.

SUBSTRING Example 9a

```
SELECT PROFESSOR.PR_NAME,
'('||SUBSTR(PROFESSOR.PR_PHONE,1,3)||')'||
SUBSTR(PROFESSOR.PR_PHONE,4,3)||'-'|
SUBSTR(PROFESSOR.PR_PHONE,7,4)  "Phone" FROM PROFESSOR
WHERE SUBSTR(PROFESSOR.PR_PHONE,  9,2) BETWEEN '45' and '65';
```

RESULT:

```
Pr_name          Phone

John Smith       (523)556-7645
John B Smith     (523)556-7556
Sunil Shetty     (523)556-6764
Katie Shef       (523)556-8765
Cathy Cobal      (523)556-5345
Jeanine Troy     (523)556-5545
Tiger Woods      (523)556-5563
```


MySQL, on the other hand, does not support the concatenation operator (II) concatenating strings. Instead, concatenation requires use of the concatenation function.[9] The CONCAT function takes the form:

```
CONCAT(string [,string ...])
```

and thus allows for any number of string arguments. Example 9b contains a MySQL query with the same functionality as that shown in Example 9a.

SUBSTRING Example 9b

```
SELECT PROFESSOR.PR_NAME,
CONCAT(' (',SUBSTRING(PROFESSOR.PR_PHONE FROM 1 FOR 3),')',
SUBSTRING(PROFESSOR.PR_PHONE  FROM  4  FOR  3),'-',
SUBSTRING(PROFESSOR.PR_PHONE FROM 7 FOR 4)) AS "Phone"
FROM PROFESSOR
WHERE SUBSTRING(PROFESSOR.PR_PHONE FROM 9 FOR 2) BETWEEN '45' AND '65';
```

1222 The CHARJ-ENGTH (char) Function[10]

The SQL-92 CHAR_LENGTII *(char)*[11] function returns the length of the character string *char*. Both Oracle and MySQL use the LENGTH *(char)* syntax to implement the CHAR_LENGTH function in the SQL-92 standard. The LENGTH function returns a numeric value.

[9] Oracle also supports the CONCAT function. However, unlike MySQL's CONCAT function, the Oracle CONCAT function can only combine two string arguments. **Hence,** use of the concatenation operator (II) is preferred. However, nesting of one CONCAT function within another CONCAT function can be used to combine two string arguments.

[10] The LENGTH *(char)* Function is called the CHARJLENGTH ***{string)*** function in the SO>92 standard. It represents the length of a character string.

[11] The LENGTH ***(char)*** Function is the LEN ***(char)*** Function in SQL Server.

LENGTH Example 1

```
SELECT LENGTH ('ABCDEFG') FROM DUAL;
SELECT LENGTH ('ABCDEFG');
```

```
RESULT:  7
```

Using the TEXTBOOK table, the SELECT statements in Examples 2 and 3 illustrate the difference when the LENGTH function is applied to a column defined as a VARCHAR data type (TEXTBOOK.Tx_title) versus one defined as a CHAR data type (TEXTBOOK.Tx_publisher).

LENGTH Example 2

```
SELECT TEXTBOOK.TX_TITLE, TEXTBOOK.TX_PUBLISHER, LENGTH(TEXTBOOK.TX_TITLE)
FROM TEXTBOOK;
```

RESULT:

Tx_title	Tx_publisher	LENGTH(TEXTBOOK.Tx_title)
Database Management	Thomson	19
Linear Programming	Prentice-Hall	18
Simulation Modeling	Springer	19
Systems Analysis	Thomson	16
Principles of IS	Prentice-Hall	16
Economics For Managers		22
Programming in C++	Thomson	18
Fundamentals of SQL		19
Data Modeling		13

LENGTH Example 3

```
SELECT TEXTBOOK.TX_TITLE, TEXTBOOK.TX_PUBLISHER,
LENGTH(TEXTBOOK.TX_PUBLISHER)
FROM TEXTBOOK;
```

Oracle RESULT

Tx_title	Tx_publisher	LENGTH(TEXTBOOK.Tx_publisher)
Database Management	Thomson	13
Linear Programming	Prentice-Hall	13
Simulation Modeling	Springer	13
Systems Analysis	Thomson	13
Principles of IS	Prentice-Hall	13
Economics For Managers		
Programming in C++	Thomson	13
Fundamentals of SQL		
Data Modeling		13

MySQL RESULT

Tx_title	Tx_publisher	LENGTH(TEXTBOOK.Tx_publisher)
Database Management	Thomson	7
Linear Programming	Prentice-Hall	13
Simulation Modeling	Springer	8

```
Systems Analysis        Thomson          7
Principles of IS        Prentice-Hall   13
Economics For Managers                  NULL
Programming in C++      Thomson          7
Fundamentals of SQL                     NULL
Data Modeling                           0
```

Observe how the MySQL LENGTH function ignores trailing blanks in calculating the length of a CHAR data type, and returns a null value for the textbooks whose publisher consists of a null value, and a value of 0 for the textbooks whose publisher consists of a single space (as a result of ignoring trailing blanks).

1223 The TRIM Function

The SQL-92 standard includes three basic TRIM functions for trimming characters from a string. TRIM (LEADING unwanted FROM string) trims off any leading occurrences of *unwanted;* TRIM (TRAILING unwanted FROM string) trims off any trailing occurrences of *unwanted;* and TRIM (BOTH unwanted FROM string) trims both leading and trailing occurrences of *unwanted.* TRIM is particularly useful with a CHAR data type in cases where it is desirable to remove unwanted blank spaces from the end of the string.

Both Oracle and MySQL support LTRIM and RTRIM functions. LTRIM (string [,unwanted]) removes unwanted characters from the beginning of a *string* while RTRIM (string [,unwanted]) removes *unwanted* characters from the end of a *string.* As illustrated in Trim Example 1, the *unwanted* argument is a string that contains the characters to be trimmed, and defaults to a single space. In MySQL you can trim only spaces[12] whereas in Oracle you can remove many characters at once by listing them all in the *unwanted* string. However, MySQL does offer TRIM (LEADING...), TRIM (TRAILING...), and TRIM (BOTH ...) functions that are part of the SQL-92 standard.

TRIM Example 1

```
SELECT LTRIM('xxxLAST WORD') FROM DUAL;
SELECT LTRIM(' LAST WORD');
```

RESULT: LAST WORD

The SELECT statement in TRIM Example 2 trims the leftmost 'x's from the character string 'xxxXxxLASTWORD'. Note that MySQL requires the use of the TRIM (LEADING ...) function to trim the leading 'x's'.

TRIM Example 2

```
SELECT LTRIM ('xxxXxxLAST WORD', 'x') FROM DUAL;
SELECT TRIM(LEADING 'x' FROM 'xxxXxxLAST WORD');
```

RESULT: XxxLAST WORD

All forms of the TRIM function are case sensitive. Thus the SELECT statement in the TRIM Example 3 does not result in trimming any characters from the string 'xxxXxx-LAST WORD'.

[12] SQL Server and DB2 also permit only spaces to be trimmed.

TRIM Example 3

```
SELECT LTRIM ( ' XxxXxxLAST WORD ' , ' X ' ) FROM DUAL ;
SELECT TRIM(LEADING 'X' FROM 'xxxXxxLAST WORD') ;
```

RESULT: xxxXxxLAST WORD

TRIM Example 4 illustrates the difference between Oracle's implementation of the TRIM function and the MySQL implementation. Note how the Oracle LTRIM function not only trims the word "Systems" from the beginning of the textbook title "Systems Analysis," it also trims the leading "S" from all titles that begin with the letter "S" (i.e., Simulation Modeling). This is because in Oracle's LTRIM function it is important to note that any character string that begins with any of the characters included in *unwanted* will be trimmed.

TRIM Example 4—Oracle Version

```
SELECT TEXTBOOK.TX_TITLE, LTRIM(TEXTBOOK.TX_TITLE,
'Systems') "Trimmed Title"
FROM TEXTBOOK;
```


Oracle Result:

```
Tx_title                Trimmed Title

Database Management     Database Management
Linear Programming      Linear Programming
Simulation Modeling     Simulation Modeling
Systems Analysis         Analysis
Principles of IS        Principles of IS
Economics For Managers  Economics For Managers
Programming in C++      Programming in C++
Fundamentals of SQL     Fundamentals of SQL
Data Modeling           Data Modeling
```

TRIM Example 4—MySQL Version

```
SELECT TEXTBOOK.TX_TITLE,
TRIM(LEADING 'Systems'FROM TEXTBOOK.TX_TITLE)  "Trimmed Title"
FROM TEXTBOOK;
```

MySQL Result:

```
Tx_title                Trimmed Title

Database Management     Database Management
Linear Programming      Linear Programming
Simulation Modeling     Simulation Modeling
Systems Analysis         Analysis
Principles of IS        Principles of IS
Economics For Managers  Economics For Managers
Programming in C++      Programming in C++
Fundamentals of SQL     Fundamentals of SQL
Data Modeling           Data Modeling
```

Suppose the trim function is to be applied to the character string "Systemsim". How would the results differ from that shown in TRIM Example 4 for Oracle's LTRIM function and the MySQL TRIM (LEADING ...) function?

The SELECT statements in TRIM Examples 5 through 7 provide additional details on how the Oracle LTRIM function works. No MySQL equivalents are shown here because in -each case, use of the MySQL TRIM (LEADING ...) function results in no characters being trimmed.

Since there are no characters prior to the character string "LAST WORD," other than those in the set "xX", all of the leading x's and X's are trimmed in TRIM Example 5.

TRIM Example 5

```
SELECT LTRIM ('xxxXxxLAST WORD', 'xX') FROM DUAL;
```

```
RESULT:   LAST WORD
```

Since the leading x's in char are in the set 'yx', they are trimmed and thus not displayed in the result shown in TRIM Example 6.

TRIM Example 6

```
SELECT LTRIM ('xxxXxxLAST WORD', 'yx') FROM DUAL;
```



```
RESULT: XxxLAST WORD
```

On the other hand, as illustrated by the SELECT statement in TRIM Example 7, since the leftmost character(s) in *unwanted* do not include a V or 'X', no characters are trimmed in the result.

TRIM Example 7

```
SELECT LTRIM ('xxxXxxLAST WORD', 'yX') FROM DUAL;
```

```
RESULT:   xxxXxxLAST WORD
```

The RTRIM function operates on the rightmost characters in a string in the same way that LTRIM operates on the leftmost characters in a string. It is important to be careful when using the RTRIM function in conjunction with a CHAR data type. Trim Examples 8 and 9 illustrate the difference between the application of this function to a VARCHAR versus CHAR data type column.

Since a VARCHAR data type stores no trailing blanks, the rightmost 'g' in the titles "Linear Programming" and "Simulation Modeling" are trimmed by the Oracle RTRIM and MySQL TRIM (TRAILING ...) functions in Trim Example 8.

TRIM Example 8

```
SELECT RTRIM(TEXTBOOK.TX_TITLE, 'g') "Trimmed Result" FROM TEXTBOOK;
SELECT TRIM(TRAILING 'g' FROM TEXTBOOK.TX_TITLE) "Trimmed Result" FROM TEXTBOOK;
```

```
RESULT:
Trimmed Result
--------------------
Database Management
Linear Programmin
Simulation Modelin
Systems Analysis
```

```
Principles of IS
Economics For Managers
Programming in C++
Fundamentals of SQL
Data Modelin
```

Trim Example 9 illustrates how the RTRIM function can be used by Oracle to trim (i.e., remove) unwanted blank spaces at the end of a CHAR data type. Column 3 of the Oracle result below confirms that the TEXTBOOK.Tx_publisher column in the TEXTBOOK table is defined as a CHAR(13) data type while column 5 verifies that the RTRIM function used in column 4 removed all trailing blank spaces from the end of all not-null publishers. Despite the definition of the TEXTBOOK.Tx_publisher column as a CHAR(13) data type, the LENGTH function in MySQL ignores the trailing blanks when determining the length of the character string. Thus the MySQL version of TRIM Example 9 does not require use of the TRIM function. Once again, observe how the MySQL LENGTH function returns a value of 0 for the length of the "single-space" publisher of the Data Modeling title.

TRIM Example 9—Oracle Version

```
SELECT TEXTBOOK.TX_TITLE, TEXTBOOK.TX_PUBLISHER,
LENGTH)TEXTBOOK.TX_PUBLISHER) "Length Pub",
RTRIM(TEXTBOOK.TX_PUBLISHER) "Trimmed Pub",
LENGTH(RTRIM(TEXTBOOK.TX_PUBLISHER)) "Trimmed Length"
FROM TEXTBOOK;
```

Oracle Result

Tx_title	Tx_publisher	Length Pub	Trimmed Pub	Trimmed Length
Database Management	Thomson	13	Thomson	7
Linear Programming	Prentice-Hall	13	Prentice-Hall	13
Simulation Modeling	Springer	13	Springer	8
Systems Analysis	Thomson	13	Thomson	7
Principles of IS	Prentice-Hall	13	Prentice-Hall	13
Economics For Managers				
Programming in C++	Thomson	13	Thomson	7
Fundamentals of SQL				
Data Modeling		13		

TRIM Example 9—MySQL Version

```
SELECT TEXTBOOK.TX_TITLE, TEXTBOOK.TX_PUBLISHER,
LENGTH(TEXTBOOK.TX_PUBLISHER) "Length Pub",
TX_PUBLISHER "Trimmed Pub",
LENGTH(TEXTBOOK.TX_PUBLISHER) "Trimmed Length"
FROM TEXTBOOK;
```

MySQL Result

Tx_title	Tx_publisher	Length Pub	Trimmed Pub	Trimmed Length
Database Management	Thomson	7	Thomson	7
Linear Programming	Prentice-Hall	13	Prentice-Hall	13
Simulation Modeling	Springer	8	Springer	8
Systems Analysis	Thomson	7	Thomson	7
Principles of IS	Prentice-Hall	13	Prentice-Hall	13
Economics For Managers				

```
Programming in C++      Thomson          7 Thomson          7
Fundamentals of SQL
Data Modeling                            0                  0
```

12.2.4 The TRANSLATE Function

The SQL-92 TRANSLATE function is used to translate characters into a string. Oracle's implementation of the TRANSLATE function has the format:

```
TRANSLATE (char, from_string, to_string)
```

The function searches *char,* **replacing each occurrence of a character found in** *Jrom_string* **with the corresponding character from** *tojstring.* **Characters that are in** *char* **but not in** *fromjstring* **are left untouched whereas characters in** *from_string* **but not in** *tojstring* **are deleted. For example, the following TRANSLATE function could be used to extract the identifier of the department from each course number.**

```
SELECT CO_COURSE#, TRANSLATE (CO_C0URSE#,
'ABCDEFGHIJKLMNOPQRSTUVWXYZ 0123456789',
'ABCDEFGHIJKLMNOPQRSTUVWXYZ') "Department"
FROM COURSE;
```



```
RESULT:
Co_course# Department

15ECON112  ECON
22QA375    QA
18ECON123  ECON
22QA411    QA
22IS270    IS
20ECES212  ECES
22QA888    QA
18ACCT801  ACCT
22IS330    IS
22IS832    IS
22IS430    IS
```

None of the characters in COURSE.Co_course# are left untouched because each character is part of *from_string.* **However since the characters '0123456789' in** *from_string* **do not appear in** *tojstring* **ABCDEFGHIJKLMNOPQRSTUVWXYZ', the digits in COURSE.Co_course are not returned by the TRANSLATE function.**

As indicated in Table 12.1, the TRANSLATE function is not available in MySQL.

12.2.5 The POSITION Function

The SQL-92 POSITION ***(target*** **IN source) function returns the position where the** *target* **string appears within the** *source* **string. Both the target string and the source string are character strings that have the same character set. The POSITION** *(target* **IN** *source)* **function returns a numeric value as follows:**

- **If the** *target* **string is of length zero (i.e., it is a null value), the result returned is one.**

- **Otherwise, if the *target* string occurs as a substring within the *source* string, the result returned is one greater than the number of characters in the *source* string that precede the first such occurrence.**
- **Otherwise, the result is zero.**

Both Oracle and MySQL implement the POSITION function with an INSTR function. The Oracle INSTR function has the following format:

```
INSTR (source, target [, position [, occurrence]])
```

The *position* argument is used to specify the starting position for the search in *source* and *occurrence* makes it possible for a specific occurrence to be found. If *position* is negative, the search begins from the end of the string.

The MySQL INSTR function meets, but does not go beyond the functionality of the SQL-92 standard. It takes the form: INSTR *(source, target)* and locates the first occurrence of the *target* in the source. MySQL also supports a LOCATE function of the form:

```
LOCATE (target, source [, position])
```

As is the case in Oracle's INSTR function, the position argument is used to specify a starting character position other than 1. However, like the MySQL INSTR function, the LOCATE function locates only the first occurrence of the *target* in the source.

The SELECT statement in INSTR Example 1 locates the character position of the second occurrence of the character 'S' in the character string 'MISSISSIPPI' beginning at character position 5. When the value of *occurrence* is changed to a 1 (see INSTR Example 2), observe that a different'S' is located. Note that the MySQL LOCATE function can be used in INSTR Example 2 but not in INSTR Example 1. When the value of *position* is changed to 4 (see INSTR Example 3), both the Oracle INSTR and MySQL LOCATE function can be used to obtain the result.

INSTR Example 1

```
SELECT INSTR ('MISSISSIPPI','S',5,2) FROM DUAL;
```

RESULT: 7

INSTR Example 2

```
SELECT INSTR ('MISSISSIPPI','S', 5 , 1) FROM DUAL;
SELECT LOCATE('S', 'MISSISSIPPI',5) ;
```

RESULT: 6

INSTR Example 3

```
SELECT INSTR ('MISSISSIPPI', 'S', 4, 1) FROM DUAL;
SELECT LOCATE('S','MISSISSIPPI',4);
```

RESULT: 4

The SELECT statement in INSTR Example 4 locates all textbooks that contain the character string 'ing' somewhere in the title. Since both *position* and *occurrence are* omitted, their values are assumed to be equal to 1.

INSTR Example 4

```
SELECT TEXTBOOK.TX_TITLE, INSTR(TEXTBOOK.TX_TITLE, 'ing') "Position of ing"
'FROM TEXTBOOK

WHERE INSTR(TEXTBOOK.TX_TITLE, 'ing') > 0;
```

RESULT:

```
Tx_title                Position of ing
Linear Programming                  16
Simulation Modeling                 17
Programming in C++                   9
Data Modeling                       11
```

Observe that the value of the INSTR function returned for all other titles is zero. Since the first occurrence of the character string 'ing' in the title was the requirement, both the Oracle and MySQL INSTR function could be used to satisfy the information request.

12.2.6 Combining the INSTR and SUBSTR Functions

The INSTR and SUBSTR functions are often combined. The purpose of the query in the following example is to begin with second word of each textbook title and display all titles with the character string 'ing' in the title. The individual lines have been numbered to facilitate discussion of the execution of the query.

```
1.  SELECT TEXTBOOK.TX_TITLE,
2.  INSTR(SUBSTR(TEXTBOOK.TX_TITLE, INSTR(TEXTBOOK.TX_TITLE, ' ')+1),'ino')
3.                          "ing string in word 2",
4.  INSTR(SUBSTR(TEXTBOOK.TX_TITLE,1),'ing')
5.                          "ing string in overall title"
6.  FROM TEXTBOOK
7.  WHERE INSTR(SUBSTR(TEXTBOOK.TX_TITLE, INSTR(TEXTBOOK.TX_TITLE, ' ')+!).'ing') > 0;
```

RESULT:

```
Tx_title             ing string in word 2 ing string in overall title
-------------------- -------------------- ---------------------------
Linear Programming                      9                          16
Simulation Modeling                     6                          17
Data Modeling                           6                          11
```

The WHERE clause shown on line 7 governs the titles displayed when the query is executed. The **INSTR(TEXTBOOK.TX_TITLE,' ')** function is evaluated first and returns the character position of the first blank space character in the title of each textbook (i.e., it is responsible for skipping over the first word of the title). By adding 1 to the value returned by the INSTR function, the character position of the first character in the second word of the title is obtained. The **SUBSTR(TEXTBOOK.TX_TITLE, INSTR(TEXTBOOK.TX_TITLE, ' ')+1)** function is evaluated next. Since a value of *n* is not provided, all remaining characters beginning with the second word of the title are selected. Finally, the outer INSTR function looks for the first occurrence of the character string 'ing' in the second or remaining words in the title. The values displayed in columns 2 and 3 indicate the values returned by the INSTR function when asked to find the character position of the character string 'ing' starting with the second word of the title (column 2) and when asked to find the character position of the character string 'ing' starting with the first word of the title (column 3).

Since the first occurrence of the 'ing' character string is the subject of the search, the MySQL INSTR and SUBSTRING functions can also be used to produce the results.

MySQL Equivalent

```
1.  SELECT TEXTBOOK.TX_TITLE,
2.  INSTR(SUBSTRING!TEXTBOOK.TX_TITLE FROM INSTR(TEXTBOOK.TX_TITLE, '  •)+1),'ing')
3.                          "ing string in word 2",
4.  INSTR(SUBSTRING!TEXTBOOK.TX_TITLE FROM 1),'ing')
5.                          "ing string in overall title"
6.  FROM TEXTBOOK
7.  WHERE INSTR(SUBSTRING)TEXTBOOK.TX_TITLE FROM INSTR(TEXTBOOK.TX_TITLE, '  ')+!),'ing') > 0;
```

12.3 Some Brief Comments on Handling Dates and Times

Examples 1.2.7-1.2.9 in Section 11.2.1.2 of Chapter 11 illustrate how mathematical operations can be included in the column-list of a query. In addition to operations involving columns and constants defined as numeric, as illustrated in Example 1.3.5 in Section 11.2.1.3, mathematical operations can also be performed on dates.

All DBMS vendors offer SQL functions to handle dates and times. Unfortunately, the implementation of date/time data types is far from standardized across vendors. This problem occurs because the ANSI SQL-92 standard defines the support of date data types, but does not say how those data types should be stored (Rob and Coronel, 2004). Given these differences, the material in this section is based on Oracle and contains examples that work with Oracle dates and times. Please refer to the SQL reference material for your database platform for the syntax associated with features comparable to those discussed here.

As indicated in Table 10.1, a date data type in SQL-92 is ten characters long in the format yyyy-mm-dd. While MySQL makes use of this format, Oracle uses as its default date format dd-mon-yy where "dd" represents a two-digit day, "mon" a three-letter month abbreviation, and "yy" a two-digit year (e.g., the date July 29, 2007 would be represented as 29-jul-07). The Oracle default format can be changed to that of the SQL-92 default by the following SQL statement.

```
ALTER SESSION SET NLS_DATE_FORMAT =  'yyyy-mm-dd';[13]
```

The remainder of the examples in this section make use of the SQL-92 format for representing a date.

Although referenced as a non-numeric field, a date is actually stored internally in a numeric format that includes the century, year, month, day, hour, minute, and second. Although dates appear as non-numeric fields when displayed, calculations can be performed with dates because they are stored internally as numeric data in accordance with the Julian calendar. A Julian date represents the number of days that have elapsed between a specified date and January 1, 4712 B.C. As a result, the query in Example 1.3.5 in Section 11.2.1.3 calculates the age (in years) of each professor in department 3 when hired.

SQL SELECT Statement:

```
SELECT PROFESSOR. PRJUAME,


TRUNC((PROFESSOR.PR_DATEHIRED  -  PROFESSOR.PR_BIRTHDATE)/365.25,0)
```

[13] NLS is an acronym lor National Language Support.

```
"Age When Hired"
FROM PROFESSOR

WHERE PROFESSOR.PR_DPT_DCODE = 3;
```

RESULT:

```
Pr_name          Age When Hired
Chelsea Bush                46
Tony Hopkins                47
Alan Brodie                 56
Jessica Simpson             40
Laura Jackson               26
```

The CURRENT_DATE function is part of the SQL-92 standard and is used to record the current date and time. For example, the following SELECT Statement calculates the age (in days) and the age (in years) for each professor in department 3.

SQL SELECT Statement:

```
SELECT PROFESSOR.PR_NAME,
CURRENT_DATE - PROFESSOR.PR_BIRTHDATE "Age in Days",
TRUNC((CURRENT_DATE - PROFESSOR.PR_BIRTHDATE)/365.25,0)  "Age in Years"
FROM PROFESSOR

WHERE PROFESSOR.PR_DPT_DCODE = 3
```


RESULT:

```
Pr_name          Age in Days Age in Years
Chelsea Bush      21876.8535           59
Tony Hopkins      20698.8535           56
Alan Brodie       22839.8535           62
Jessica Simpson   18598.8535           50
Laura Jackson     11971.8535           32
```

The reason why the age in days contains a fractional component is due to fact that when referenced in a query, the CURRENT_DATE function retrieves not only the current date but also the time of day. Thus the previous query was executed when 85.35 percent of a 24 hour day had elapsed (i.e., the query was executed at approximately 8:29 pm).

Oracle uses the TO_CHAR and TOJDATE functions with dates and times. The TO_CHAR function is used to extract the different parts of a date/time and convert them to a character string[14] while the TO_DATE function is used to convert character strings to a valid date format. Both functions make use of a format element (also known as a format mask). Table 12.2 contains a sample of format elements commonly used in conjunction with the TO_CIIAR and TO_DATE functions. Table 12.3 shows the number format elements used with the TO_CHAR function.

[14] The TO_CHAR function can also be used to convert a number to a formatted character string. Format masks used in conjunction with numbers appear in Table **12.3.**

Table 12.2 Selected date and time format elements used with the TO_CHAR and TO_DATE functions

Element	Description	Example
MONTH, Month, or month	Name of the month spelled out—padded with blank spaces to a total width of nine spaces; case follows format.	JULY, July, or july (5 spaces follows each representation of July)
MON, Mon, or mon	Three-letter abbreviation of the name of the month; case follows format.	JUL, Jul, or jul
MM	Two-digit numeric value of the month.	7
D	Numeric value of the day of the week.	Monday = 2
DD	Numeric value of the day of the month.	23
DAY, Day, or day	Name of the day of the week spelled out—padded with blank spaces to a length of nine characters.	MONDAY, Monday, or monday (3 spaces follows each representation of Monday)
fm	"Fill mode." When this element appears, subsequent elements (such as MONTH) suppress blank padding leaving a variable-length result.	fmMonth, yyyy produces a date such as March, 2007
DY	Three-letter abbreviation of the day of the week.	MON, Mon, or mon
YYYY	The four-digit year.	2007
YY	The last two digits of the year.	07
YEAR, Year, or year	Spells out the year; case follows year.	TWO THOUSAND SEVEN
BC or AD	Indicates B.C. or A.D.	2007 A.D.
AM or PM	Meridian indicator	10:00 AM
J	Julian date. January 1, 4712 B.C. is day 1.	July 27, 2007 is Julian date 2454309
SS	Seconds (value between 0 and 59)	21
MI	Minutes (value between 0 and 59)	32
HH	Hours (value between 1 and 12)	9
HH24	Hours (value between 0 and 23)	13

Table 12.3 Selected number format elements used with the TO_CHAR function

Element	Description	Example
9	Series of 9s indicates width of display (with insignificant leading zeros not displayed).	99999
0	Displays insignificant leading zeros.	0009999
$	Displays a floating dollar sign to prefix value.	$99999
.	Indicates number of decimals to display.	999.99
,	Displays a comma in the position indicated.	9,999

The following query illustrates the use of the TO_CHAR function to display the date of birth and salary of each professor in department 3 using the default format and an alternative format.


```
SELECT PROFESSOR.PR_NAME,
PROFESSOR.PR_BIRTHDATE "SQL-92 Date Format",
TO_CHAR(PROFESSOR.PR_BIRTHDATE,'DD-MON-YYYY') "Alternate Format",
PROFESSOR.PR_SALARY "Default Format",
TO_CHAR(PROFESSOR.PR_SALARY, '$99,999.00') "Alternate Format"
FROM PROFESSOR
WHERE PROFESSOR.PR_DPT_DCODE = 3
```

RESULT:

Pr_name	SQL-92 Date Format	Alternate Format	Default Format	Alternate Format
Chelsea Bush	1946-09-03	03-SEP-1946	77000	$77,000.00
Tony Hopkins	1949-11-24	24-NOV-1949	77000	$77,000.00
Alan Brodie	1944-01-14	14-JAN-1944	76000	$76,000.00
Jessica Simpson	1955-08-25	25-AUG-1955	67000	$67,000.00
Laura Jackson	1973-10-16	16-OCT-1973	43000	$43,000.00

When inserting a date in a table, Oracle assumes a default time of 12:00 AM (midnight). Should it be necessary to associate a time other than 12:00 AM with a date, the TO_DATE function can be used. For example, suppose we wish to insert the date and time of admission for each new patient into the PATIENT table. The INSERT statement that appears below illustrates how this could be done. Prior to and following the INSERT statement are two SELECT statements. The first two SELECT statements display the name and date of admission of each patient prior to the insertion of the new patient. Note that the first SELECT statement displays the date of admission using the default date format while the second displays the date of admission using a format mask that includes the time portion of the date of admission. The use of the TO_DATE function in the INSERT statement for patient Zhaoping Zhang allows both the date and time of her admission to be recorded.

Name and Date of Admission of Patients in PATIENT Table Prior to Insertion

```
SELECT PATIENT.PAT_NAME, PATIENT.PAT_ADMIT_DT "Date of Admission"
FROM PATIENT;
```

```
Pat_name           Date of Admission
------------------ -----------------
Davis, Bill        2007-07-07
Li, Sue            2007-08-25
Grimes, David      2007-07-12
```

```
SELECT PATIENT.PAT_NAME, TO_CHAR(PAT_ADMIT_DT, 'fmMonth dd, yyyy HH:
MI AM') "Date of Admission"
FROM PATIENT;
```

```
Pat_name           Date of Admission
------------------ --------------------------
Davis, Bill        July 7, 2007 12:00 AM
Li, Sue            August 25, 2007 12:00 AM
Grimes, David      July 12, 2007 12:00 AM
```


Insertion of New Patient

```
INSERT INTO PATIENT VALUES ('ZZ', '06912', 'Zhang, Zhaoping', 'F', 35,
TO_DATE('2007-08-11 10:15 AM', 'YYYY-MM-DD HH:MI AM'), NULL, NULL, NULL);
```

```
1 row created.
```

Name and Date of Admission of Patients in PATIENT Table After Insertion

```
SELECT PATIENT.PAT_NAME, PATIENT.PAT_ADMIT_DT "Date of Admission"
FROM PATIENT;
```

```
Pat_name           Date of Admission
------------------ -----------------
Davis, Bill        2007-07-07
Li, Sue            2007-08-25
Zhang, Zhaoping    2007-08-11
Grimes, David      2007-07-12
```

```
SELECT PATIENT.PAT_NAME, TO_CHAR(PAT_ADMIT_DT, 'fmMonth dd, yyyy HH:
MI AM') "Date of Admission"
FROM PATIENT;
```

```
Pat_name           Date of Admission
------------------ --------------------------
Davis, Bill        July 7, 2007 12:00 AM
Li, Sue            August 25, 2007 12:00 AM
Zhang, Zhaoping    August 11, 2007 10:15 AM
Grimes, David      July 12, 2007 12:00 AM
```

Suppose patient Zhaoping Zhang is discharged on August 12 at 3 : 3 5 pm (i.e. the value of the CURRENTJDATE function is 3 : 3 5 pm). The following SELECT statement records her length of stay.

```
SELECT PATIENT.PAT_NAME, CURRENT_DATE - PATIENT.PAT_ADMIT_DT "Length of Stay"
FROM PATIENT
WHERE PATIENT.PAT P#A = 'ZZ' AND PATIENT.PAT_P#N = '06912'

Pat_name              Length of Stay

Zhang, Zhaoping            1.2222338
```

Note that the 5 hours and 20 minutes (the dit'ference between the time of day when she was discharged on August 12 and the time of day when she was admitted on August 11) is 22.22 percent of one day.

12.4 A Potpourri of Other SQL Queries

Date (1995) has described SQL as an *extremely redundant* language in the sense that there are almost always a variety of ways to formulate the same query in SQL. The advantage of this flexibility is that users have the option of choosing the approach with which they are most comfortable. For example, a join can be used as an alternative way of expressing many subqueries. In addition, as we have seen, subqueries and join conditions may appear in the FROM clause or even in the column list of the SELECT clause.

The declarative, nonprocedural nature of SQL, which allows the user to specify what the intended results of the query are, rather than specifying the details of how the result should be obtained, can also be a disadvantage. Ideally, the user should be concerned only with specifying the query correctly. The DBMS, on the other hand, should then take the query and execute it efficiently. While DBMS products all make use of a variety of techniques to process, optimize, and execute queries written in SQL, in practice it helps if the user is aware of which types of constructs in a query are more expensive to process in terms of performance than others. See Chapter 15 of Elmasri and Navathc (2004) for an interesting discussion of query processing and optimization.

The remainder of this section contains six examples of SQL queries that combine a number of the features introduced in Chapter 11 as well as in this chapter. Their purpose is not so much to specify the most efficient way of expressing the emery as to illustrate the application of several SQL features and functions. Since each example makes use of SQL running on an Oracle9i platform, readers working with some other database platform may wish to refer to their SQL reference material to resolve any syntactical differences created by syntax unique to Oracle9i, such as differences in function names.

12.4.1 Concluding Example 1

Suppose Madeira College is interested in finding out how many professors have been hired each month. A review of the data in the PROFESSOR table identifies two interesting problems. First there have been months during which no professors have been hired (i.e., February, March, April, and July). Second, there are two professors (John B. Smith and Jeanine Troy) whose hire date is unavailable (i.e., unknown). The following SQL SELECT statement represents one way to count the number of professors hired during each month and also includes (a) one row for each month during which no professor has been hired, and (b) a row that records the number of professors for which a date hired is unavailable. The individual lines have been numbered to facilitate the discussion.