Chapter 4
SQL

As a standard language to work with relational database, SQL includes both data definition and manipulation operations. Below defines the sample database:

CREATE TABLE S (  
S# CHAR(5),  
SNAME CHAR(20),  
STATUS NUMERIC(5),  
CITY CHAR(15)  
PRIMARY KEY (S#));

CREATE TABLE P (  
P# CHAR(6),  
PNAME CHAR(20),  
COLOR CHAR(6),  
WEIGHT NUMERIC(5,1),  
CITY CHAR(15)  
PRIMARY KEY (P#));
CREATE TABLE SP (  
  S# CHAR(5),  
  P# CHAR(6),  
  QTY NUMERIC(9),  
  PRIMARY KEY (S#, P#),  
  FOREIGN KEY (S#) REFERENCES S,  
  FOREIGN KEY (P#) REFERENCES P);

Hence, to create a base table, we have to use the CREATE TABLE statement to specify the name of the table, the name and data type of the columns, the primary and any foreign keys, in the table.

Comparing the SQL definition for the sample database and the one at the beginning of this chapter, the type definitions are missing, since SQL 8 does not allow users to define their own types. Only built-in types can be used.

**Homework:** Exercises 4.1–4.3. For 4.2 and 4.3, you can check out such products as DB2, MYSQL, etc.
Built-in types

SQL supports the following built-in types: CHARACTER[VARYING](n), BIT[VARYING](n), NUMERIC(p,q), DECIMAL(p, q), INTEGER, SMALLINT, FLOAT(p), DATE, TIME, TIMESTAMP, etc.

A number of defaults, abbreviations and alternative spellings, e.g., CHAR for CHARACTER, are also supported. The “VARYING” word sandwiched in between ‘[’ and ‘]’ is optional.
A variable of type Char contains only one character, the one with the type of Varchar2 holds up to 2,000, and the one with the type of Long holds up to 2 GB.

A value of type date has to fall into the range of 1/1/4712 B.C. and 12/31/4712 AD.

The type of NUMERIC requires a length, or precision, to be clearly stated, which is the maximum number of digits. We can also specify its scale, namely, the maximum number of digits before (-), or after (+), the decimal point.
Data manipulation

SQL provides manipulative operations such as SELECT, INSERT, UPDATE, and DELETE. In particular, we can perform relational restriction, projection, and join operations on the data, by using the SQL statement of SELECT. Below are some examples for restriction, projection, and join:

SELECT S#, P#, QTY FROM SP WHERE QTY<150;

SELECT S#, CITY FROM S;

SELECT S.S#, SNAME, STATUS, CITY, P#, QTY FROM S, SP WHERE S.S#=SP.S#;

Questions: If we apply the above queries on the sample database, what will be the results?
More examples

```
INSERT INTO TEMP(P#, WEIGHT)
    SELECT P#, WEIGHT FROM P WHERE COLOR='RED';

UPDATE S SET STATUS=STATUS*2 WHERE CITY='Paris';

DELETE FROM SP WHERE P#='P2';
```

**Questions:**
1) What do the above queries do?
2) How could we assign one table to another?

**Homework:** Complete Exercise 4.4(a, c) and think about the b part, turn them into SQL statements and carry them out with the system.
The catalog

The SQL standard does include specifications for a standard catalog called the *Information Schema*. More specifically, an SQL catalog consists of the descriptors for an individual database, and an SQL schema contains descriptors for that portion of that database that belongs to some individual user.

Thus, there can be a bunch of catalogs (one per database), each can then be divided into any number of schemas, (one per user). However, each catalog includes exactly one schema called *INFORMATION_SCHEMA*, which plays the role of a catalog, as described in the last chapter.
Views

Recall a view does not really exist, it only provides a certain way to “look at” a table. Below is an SQL example,

CREATE VIEW GOOD_SUPPLIER
    AS SELECT S#, STATUS, CITY FROM S
    WHERE STATUS > 15;

together with an SQL query against this view:

SELECT S#, STATUS FROM GOOD_SUPPLIER
    WHERE CITY='London';

Again, the real story is the follows:

SELECT GOOD_SUPPLIER.S#, GOOD_SUPPLIER.STATUS
    FROM ( SELECT S#, STATUS, CITY FROM S
       WHERE STATUS > 15 ) AS GOOD_SUPPLIER
    WHERE GOOD_SUPPLIER.CITY='London';
The above expression can then be simplified to the following:

```sql
SELECT S#, STATUS
FROM S
WHERE STATUS > 15 AND CITY='London';
```

Consider another example of DELETE:

```sql
DELETE
FROM GOOD_SUPPLIER
WHERE CITY='London';
```

It is actually simplified as the follows, and then get executed.

```sql
DELETE
FROM S
WHERE STATUS > 15 AND CITY='London';
```
Show the views

We can check the definition of a view through another user data dictionary, user_views, as follows:

```
select * from user_views where view_name;
```

There are also other such dictionaries existing, such as user_tables, i.e., tabs, user_source, which keeps all the procedures, functions, packages, etc., that are defined by the user.
Embedded SQL

Most SQL products allow SQL statements to be executed both directly, and as part of an application program, in which the SQL statements can be embedded. Below is an example of such embedding.

```sql
EXEC SQL BEGIN DECLARE SECTION;
DCL SQLSTATE CHAR(5);
DCL P# CHAR(6);
DCL WEIGHT FIXED DECIMAL(5,1);
EXEC SQL END DECLARE SECTION;

P#='P2';
EXEC SQL SELECT P.WEIGHT
INTO :WEIGHT
FROM P
WHERE P.P#:P#
IF SQLSTATE='00000'
THEN ...;
ELSE ...;
```
A couple of points

1. An embedded SQL statement begins with EXEC SQL, and ends with a special terminator symbol.

2. An executable SQL statement can appear wherever an executable host language statement can appear.

3. SQL statements can refer to host variables, which must be prefixed with a ‘:’ prefix. Such variables can also appear in an INTO statement, where values are to be retrieved.
4. All host variables referenced in SQL statements must be declared within an *embedded SQL declare* section.

5. Every program containing embedded SQL statements must include a host variable, SQLSTATE. After any SQL statement is executed, a status code is returned to the program in that variable. A code of 00000 means that the statement executed successfully, and a value of 02000 means the statement did execute, but no data was found to satisfy the request.
Thus, every SQL statement should be followed by a test of the returned SQLSTATE value. The WHENEVER statement is provided to simplify this process, which takes the following form:

```
EXEC SQL WHENEVER <condition><action>;
```

Here `<condition>` is either SQLERROR or NOT FOUND, and the `<action>` is either CONTINUE or a GO TO statement. Technically speaking, it is really a compiler directive. For example, "WHENEVER `<condition>` GO TO `<label>`" causes the compiler to insert a statement of the form "IF `<condition>` GO TO `<label>` END IF" after each executable SQL statement it encounters.
Why a Cursor?

Most of the SQL statements can be easily handled in an embedded environment. Retrieval operations, however, have to be handled differently, since they retrieve many rows, not just one, and host language is usually not be able to support that. Thus, we need to provide some mechanism to fill the gap between the set-level retrieval capabilities of SQL and the row-level retrieval capabilities of the host.

A cursor is a special SQL object that applies to embedded SQL only. It is basically a (logic) pointer that can be used to run through a collection of rows, pointing to each of them at one time.
Operations w/o cursors

The following manipulation statements do not need cursors: Singleton SELECT, INSERT, UPDATE, AND DELETE. Below are some examples:

EXEC SQL SELECT STATUS, CITY
INTO :RANK, :CITY
FROM S
WHERE S# = :GIVENS#;

In the above, if there is exactly one row satisfying the condition, a code of 00000 will be returned; if no row satisfies, 02000 will be returned; otherwise, an error code will be sent back.
The following statement inserts a new part into table P.

EXEC SQL INSERT
INTO P (P#, PNAME, WEIGHT)
VALUES (:p#, :PNAME, :PWT);

The following updates the STATUS information.

EXEC SQL UPDATE S
SET STATUS=STATUS+:RAISE
WHERE CITY='London';

The following deletes certain shipments.

EXEC SQL DELETE FROM SP
WHERE :CITY= (SELECT CITY
    FROM S WHERE S.S#=SP.S#);
Operations with cursors

Now, we consider the question of retrieving a set containing an arbitrary number of rows, instead of at most one row in the previous case.

```
EXEC SQL DECLARE X CURSOR FOR
    SELECT S.S#, S.SNAME, S.STATUS
FROM S WHERE S.CITY=:Y
ORDER BY S# ASC;
EXEC SQL OPEN X;
DO for all S rows accessible via X;
EXEC SQL FETCH X INTO :S#:SNAME,:STATUS;
...
END;
EXEC SQL CLOSE X;
```

The idea is that once that set is defined, all the rows it may contain will be fetched out one at a time.

**Homework:** Think about Exercises 4.5.
Dynamic SQL

This form of SQL consists of a set of embedded SQL facilities that support the construction of generalized, online, and possibly interactive applications. For an online application, it will go through the following steps:
1. Accept a command from the terminal.
2. Analyze that command.
3. Execute appropriate SQL statements on that database.
4. Return a message and/or results to the terminal.

If the set of commands the program can accept is fairly smaller, then steps 2 and 3 will consist of simple logic to check the input command and then branch off to certain parts that issue the predefined SQL commands.
On the other hand, if there can be large variety in the input, then it might not be practical to hardwire SQL statement for every possible command. Instead, it might be more convenient to construct the needed SQL statements dynamically, and then compile and execute those statements just constructed. Below is an example.

The two main dynamic statements are PREPARE and EXECUTE.

```
DCL SQLSOURCE CHAR VARYING(65000)
SQLSOURCE='DELETE FROM SP WHERE QTY<300';
EXEC SQL PREPARE SQLPREPPED FROM :SQLSOURCE;
EXEC SQL EXECUTE SQLPREPPED;
```
SQL/CLI vs. Dynamic SQL

SQL Call-level Interface (CLI) is a more recent addition to the standard, which is heavily based on Microsoft’s ODBC interface. It also addresses the same problem as the dynamic SQL does, i.e., allow application to be written for which the exact SQL statements to be executed are not known until run time.

Dynamic SQL is a source code standard. Any application using it requires the services of some specific SQL compiler to support PREPARE and EXECUTE, etc, as required by that standard. CLI, by contrast, standardizes the routine invocation formats, thus only needs the standard service of certain host language, but not a specific sql processor. Thus, the latter is DBMS independent.
In contrast

The previous Dynamic SQL command can be represented, in SQL/CLI, as follows:

```c
char sqlsource [65000];

strcpy (sqlsource, "DELETE FROM SP WHERE QTY<300’’);
rc = SQLExecDirect (hstmt, (SQLCHAR *)sqlsource, SQL_NTS);
```

In the above, the SQL/CLI command, SQLExecDirect is similar to EXECUTE IMMEDIATELY of Dynamic SQL standard.