Chapter 5. Structured Query Language

CHAPTER 5

STRUCTURED QUERY LANGUAGE

After reading this chapter, the reader will understand:

- *The basic features of SQL*
- *The concept of schema and catalog in SQL*
- *Different types of data types available in SQL*
- *DDL commands to create relations in SQL*
- *Various constraints that can be applied on the attributes of a relation*
- *DDL commands to alter the structure of a relation*
- *DML commands to manipulate the data stored in the database*
- *Various clauses of* `SELECT` *command that can be used to extract data from database*
- *Various set operations like union, intersect, and minus*
- *The use of* `INSERT`, `UPDATE`, *and* `DELETE` *commands to insert, update, and delete tuples in a relation*
- *Searching of null values in a relation*
- *The use of various aggregate function in select query*
- *The use of* `GROUP BY` *and* `HAVING` *clause for the grouping of tuples in a relation*
- *Combining tuples from two relations using join queries*
- *The concept of nested queries*
- *Specifying complex constraints using the concept of assertions*
- *The concept views, whose contents are derived from already existing relations and does not exist in physical form*
- *The concept of stored procedures and functions in SQL*
- *The use of trigger for automatic execution of the stored procedures*
- *The difference between embedded and dynamic SQL*
- *The use of cursor for extracting multiple tuples from a relation*

- *The role of Open Database Connectivity (ODBC) for establishing connection with databases*
- *The use of Java Database Connectivity (JDBC) for establishing connectivity with databases in Java*
- *The use of SQL-Java standard for embedding SQL statements in Java program*

The structured query language (SQL) pronounced as "ess-que-el", is a language which can be used for retrieval and management of data stored in relational database. It is a non-procedural language as it specifies what is to be retrieved rather than how to retrieve it. It can be used for defining the structure of data, modifying data in the database, and specifying the security constraints. SQL is an interactive query language that helps users to execute complicated queries in minutes, sometimes even in seconds as compared to the number of days taken by a programmer to write a program for those complicated queries. In addition, commands in SQL resemble simple English statements, making it easier to learn and understand. These features make SQL as one of the major reasons of the success of relational databases. Some common relational database management systems that use SQL are Oracle, Sybase, Microsoft SQL Server, Access, Ingres, etc.

Donald D. Chamberlin and Raymond F. Boyce developed the first version of SQL, also known as ***Sequel*** *as* part of the System R project in the early 1970s at IBM. Later on, SQL was adopted as a standard by the ANSI in 1986, and was subsequently adopted as an ISO standard in 1987. This version of SQL was called SQL-86. In 1989, a revised standard commonly known as SQL-89 was published. The demand for more advanced SQL version led to the development of new version of SQL called SQL-92. This version addressed several weaknesses in SQL-89 standard. In 1999, the standard SQL-99 was released by ANSI/ISO. This version addresses some of the advanced areas of modern SQL systems, such as object-relational database concepts, integrity management, etc.

The complete coverage of SQL cannot be provided in a single chapter

hence; only the fundamental concepts have been discussed in this chapter. Moreover, the implementation of various SQL commands may differ from version to version.

5.1 BASIC FEATURES OF SQL

SQL has proved to be the standard language as same set of commands are used to create, retrieve, and modify data regardless of the system they are working on. SQL provides different types of commands, which can be used for different purposes. These commands are divided into two major categories.

- **Data definition language (DDL):** DDL provides commands that can be used to create, modify, and delete database objects. The database objects include relation schemas, relations, views, sequences, catalogs, indexes, etc.
- **Data manipulation language (DML):** DML provides commands that can be used to access and manipulate the data, that is, to retrieve, insert, delete, and update data in a database.

Learn More

The first commercially available implementation of SQL was released in 1979 by Relational Software Inc., which is known as *Oracle Corporation* today. Thus, Oracle is the pioneer RDBMS that started using SQL.

In addition, there are various techniques like embedded SQL, cursors, dynamic SQL, ODBC, JDBC, SQLJ, which can be used for accessing databases from other applications. These techniques will be discussed later in the chapter.

5.2 DATA DEFINITION

Data definition language (DDL) commands are used for defining relation schemas, deleting relations, creating indexes, and modifying relation schemas. In this section, DDL commands are discussed and an overview of the basic data types in SQL is provided. In addition, it discusses how

integrity constraints can be specified for a relation.

5.2.1 Concept of Schema and Catalog in SQL

In earlier versions of SQL, there was no concept of relational database schema, thus, all the database objects were considered to be a part of same database schema. It implies that no two database objects can share same name, since they all are part of the same name space. Due to this, all users of the database have to coordinate with each other to ensure that they use different names for database objects. However, the present day database systems provide a three level hierarchy for naming different objects of relational database. This hierarchy comprises **catalogs**, which in turn consists of **schemas**, and various database objects are incorporated within the schema. For example, any relation can be uniquely identified as:

```
catalog_name.schema_name.relation_name
```

NOTE Schema is identified by a schema name, and its elements include relations, constraints, views, domains, etc.

Database system can have multiple catalogs, and users working with different catalogs can use the same name for database objects without any clashes. Each user of database system is associated with a default catalog and schema, and whenever the user connects with the database (by providing the unique combination of user name and password), he gets connected with the default catalog and schema.

In SQL, one can create schema by using the `CREATE SCHEMA` command, including definitions of all the elements of schema. Alternatively, the schema can be given a name and authorization identifier to indicate the user who is owner of schema, and the elements of schema can be specified later. For example, to create *Online Book* schema owned by the user identified by authorization identifier *Smith*, the command can be specified as

```
CREATE SCHEMA OnlineBook AUTHORIZATION Smith;
```

The schema can be deleted by using the `DROP SCHEMA` command. The privilege to create or drop schemas, relations, and other components of database is granted to relevant user by the system administrator.

Integrity constraints like referential integrity can be defined between relations only if they belong to schemas within the same catalog.

5.2.2 Data Types

Data type identifies the type of data to be stored in an attribute of a relation and also specifies associated operations for handling the data. Data type defined for an attribute associates a set of properties with that attribute. These properties cause attributes of different data types to have different set of operations that can be performed on these attributes. The common data types supported by standard SQL are

- `NUMERIC(p, s)`: used to represent data as floating-point number like 17.312, 27.1204, etc. The number can have `p` significant digits (including sign) and `s` number of the `p` digits can be present on the right of decimal point. For example, data type specified as `NUMERIC(5, 2)` indicates that value of an attribute can be of form 332.32, where as number of the forms 32.332 or 0.332 are not allowed.
- `INT` or `INTEGER`: used to represent data as a number without a decimal point. The size of the data is machine dependent.
- `SMALLINT`: used to represent data as a number without a decimal point. It is a subset of the `INTEGER` so the default size is usually smaller than `INT`.
- `CHAR(n)` or `CHARACTER(n)`: used to represent data as fixed-length string of characters of size `n`. In case of fixed-length strings, a shorter string is padded with blank characters to the right. For example, if the value *ABC* is to be stored for an attribute with data

type `CHAR(8)`, the string is padded with five blanks to the right. Padded blanks are ignored during comparison operation.

- `VARCHAR(n)` or `CHARACTER VARYING`: used to represent data as variable length string of characters of maximum size `n`. In case of variable length string, a shorter string is not padded with blank characters.
- `DATE` and `TIME`: used to represent data as date or time. The `DATE` data type has three components, namely, *year, month,* and *day* in the form `YYYY-MM-DD`. The `TIME` data type also have three components, namely, *hours, minutes*, and *seconds* in the form `HH:MM:SS`.
- `BOOLEAN`: used to represent the third value *unknown*, in addition to *true* and *false* values, because of the presence of *null* values in SQL.
- `TIMESTAMP`: used to represent data consisting of both date and time. The `TIMESTAMP` data type has six components, *year, month, day, hour, minute*, and *second* in the form `YYYY-MM-DDHH:MM:SS[.sF]`, where `F` is the fractional part of the *second* value.

In addition to these data types, there are many more data types supported by SQL like `CLOB` (CHARACTER LARGE OBJECT), `BLOB` (BINARY LARGE OBJECT), etc. These data types are rarely used and hence, are not discussed here.

In addition to built-in data types, user-defined data types can also be created. The user-defined data type can be created using the `CREATE DOMAIN` command. For example, to create user-defined data type `Vchar`, the command can be specified as

```
CREATE DOMAIN Vchar AS VARCHAR(15);
```

Now, `Vchar` can be used as data type for any attribute for which data type `VARCHAR(15)` is to be defined. After declaring such data type, it becomes easier to make changes in the domain that is being used by numerous attributes in a relation schema.

5.2.3 CREATE TABLE Command

The `CREATE TABLE` command is used to define a new relation, its attributes and its data types. In addition, various constraints like key, entity integrity, and referential integrity constraints can also be specified. The syntax for `CREATE TABLE` command is shown here.

```
CREATE TABLE <table_name> (<attribute_1> <data_type_1> [constr
                           <attribute_2> <data_type_2> [constr
                           :
                           <attribute_n> <data_type_n> [constr
                           [table_constraint_1]
                           :
                           [table_constraint_n]
                      );
```

where,

`table_name` is the name of new relation

$attribute_i$ is the attribute of relation

$data\_type_i$ is the data type of values of the attribute

$constraint_i$ is any of the column-level constraints defined on the corresponding attribute

$table\_constraint_i$ is any of the table-level constraints

For example, the command to create `BOOK` relation whose schema is `BOOK(ISBN, Book_title, Category, Price, Copyright_date, Year, Page_count, P_ID)` can be specified as

CREATE TABLE BOOK (	ISBN	VARCHAR(15),
	Book_title	VARCHAR(50),
	Category	VARCHAR(20),
	Price	NUMERIC(6,2),

	Copyright_ date	NUMERIC(4),
	Year	NUMERIC(4),
	Page_count	NUMERIC(4),
	P_ID	VARCHAR(4)
);		

Once the relation is created, its structure can be viewed by using the DESCRIBE (or DESC) command. For example, the command to view the structure of BOOK relation can be specified as

```
DESCRIBE BOOK
```

OR

```
DESC BOOK
```

As a result of this command, the structure of the relation BOOK is displayed as shown here.

Name	Null?	Type
----------	------	----------
ISBN		VARCHAR2(15)
BOOK_TITLE		VARCHAR2(50)
CATEGORY		VARCHAR2(20)
PRICE		NUMBER(6,2)
COPYRIGHT_DATE		NUMBER(4)
YEAR		NUMBER(4)
PAGE_COUNT		NUMBER(4)
P_ID		VARCHAR2(4)

In this example, the relation BOOK is created without specifying any constraints. Since no constraints are defined, invalid values can be entered in the relation. To avoid such a situation, it is necessary to define constraints within the definition of the relation.

5.2.4 Specifying Constraints

As discussed earlier, constraints are required to maintain the integrity of the data, which ensures that the data in database is consistent, correct, and valid. These constraints can be specified within the definition of a relation. These include key, integrity, and referential constraints along with the restrictions on attribute domains and *null* values. This section discusses how different types of constraints can be specified at the time of creation of relation.

PRIMARY KEY Constraint

This constraint ensures that attribute declared as primary key cannot have *null* value and no two tuples can have same value for primary key attribute. In other words, the values in the primary key attribute are not *null* and are unique. If a primary key has a single attribute, the `PRIMARY KEY` can be applied as a column-level as well as table-level constraint. The syntax of `PRIMARY KEY` constraint when applied as a column-level constraint is given here.

```
<attribute> <data_type> PRIMARY KEY
```

The syntax of `PRIMARY KEY` constraint when applied as a table-level constraint is

```
PRIMARY KEY (<attribute>)
```

For example, the attribute `ISBN` of `BOOK` relation can be declared as a primary key as shown here.

```
ISBN VARCHAR(15) PRIMARY KEY
```

OR

```
PRIMARY KEY (ISBN)
```

If a primary key has more than one attribute, the PRIMARY KEY constraint is specified as table-level constraint. The syntax of PRIMARY KEY constraint when applied on more than one attribute is

```
PRIMARY KEY (<attribute1 >, <attribute2>, …, <attributen>)
```

For example, attributes ISBN and R_ID of REVIEW relation can be declared as composite primary key as shown here.

```
PRIMARY KEY (ISBN, R_ID)
```

UNIQUE Constraint

The UNIQUE constraint ensures that the set of attributes have unique values, that is, no two tuples can have same value in the specified attributes. Like PRIMARY KEY constraint, UNIQUE constraint can be applied as a column-level as well as table-level constraint. The syntax of UNIQUE constraint when applied as a column-level constraint is given here.

```
<attribute> <data_type> UNIQUE
```

The syntax of UNIQUE constraint when applied as a table-level constraint is

```
UNIQUE (<attribute>)
```

For example, the UNIQUE constraint on attribute Pname of the relation PUBLISHER can be specified as shown here.

```
Pname VARCHAR(50) UNIQUE
```

OR

```
UNIQUE (Pname)
```

When UNIQUE constraint is applied on more than one attribute it is specified as table-level constraint. The syntax of UNIQUE constraint when applied on more than one attribute is

```
UNIQUE (<attribute1>, <attribute2>;, …, <attributen >)
```

For example, the UNIQUE constraint on attributes Pname and Address of relation PUBLISHER can be specified as shown here.

```
UNIQUE (Pname, Address)
```

CHECK Constraint

This constraint ascertains that the value inserted in an attribute must satisfy a given expression. In other words, it is used to specify the valid values for a certain attribute. The syntax of CHECK constraint when applied as a column-level constraint is given here.

```
<attribute> <data_type> CHECK (<expression>)
```

Learn More

SQL allows setting default value for an attribute by adding the DEFAULT <default_value> clause to the definition of an attribute in CREATE TABLE command.

For example, the CHECK constraint for ensuring that the value of attribute Price of the relation BOOK is greater than *$20* can be specified as

```
Price NUMERIC(6,2) CHECK (Price>20)
```

The CHECK constraint when specified as table-level constraint can be given a separate name that allows referring to the constraint whenever needed. The syntax of CHECK constraint when applied as a table-level constraint is

```
CONSTRAINT <constraint_name> CHECK (<expression>)
```

For example, the constraint on the attribute Price of the BOOK relation can be given a name as shown here.

```
CONSTRAINT Chk_price CHECK (Price > 20)
```

Constraints can also be applied on more than one attribute simultaneously. For example, the constraint that the Copyright_date must be either less than or equal to the Year (publishing year) can be specified as

```
CONSTRAINT Chk_date CHECK (Copyright_date <= Year)
```

When CHECK constraint is applied to a domain, it allows specifying a condition that must be satisfied by any value assigned to an attribute whose type is the domain. For example, the CHECK constraint can ensure that the domain, say dom, allows values between *20* and *200* as shown here.

```
CREATE DOMAIN dom AS NUMERIC(5,2)
CONSTRAINT chk_dom CHECK (VALUE BETWEEN 20 and 200)
```

If the domain dom is assigned to the attribute Price of a BOOK relation, it

ensures that the attribute `Price` must have values between *20* and *200*. As a result, if a value that is not between *20* and *200* is inserted into the `Price` attribute, an error occurs.

A domain can also be restricted to contain a specified set of values using `IN` clause with `CHECK` constraint. For example, the `CHECK` constraint ensuring that the domain, say `dom1`, allows values within a list of certain values can be specified as

```
CREATE DOMAIN dom1 CHAR(20)
CONSTRAINT chk_dom1 CHECK (VALUE IN ('Textbook', 'Language E
```

NOT NULL Constraint

The `NOT NULL` constraint is used to specify that an attribute will not accept *null* values. For example, consider a tuple in the `BOOK` relation where the attribute `Page_count` contains a *null* value. Such a tuple provides incomplete information of that particular book. In such a case, using the `NOT NULL` constraint does not permit *null* value.

The syntax of `NOT NULL` constraint is given here.

```
<attribute> <data_type> NOT NULL
```

For example, the `NOT NULL` constraint for the attribute `Page_count` of `BOOK` relation can be specified as shown here.

```
Page_count Numeric(4) NOT NULL
```

The `NOT NULL` constraint can be specified using `CHECK` constraint. For example, the `NOT NULL` constraint on the attribute `Book_title` can be specified using the `CHECK` constraint as shown here.

```
Book_title VARCHAR(50) CHECK (Book_title IS NOT NULL)
```

The `NOT NULL` constraint ensures that the attribute of a particular domain is not permitted to take *null* values. For example, a domain, say `dom2`, can be restricted to take non-null values as shown here.

```
CREATE DOMAIN dom2 VARCHAR(20) NOT NULL
```

Keypoint: *The `NOT NULL` constraint can be specified only as column-level constraint and not as table-level constraint.*

FOREIGN KEY Constraint

This constraint ensures that the foreign key value in the referencing relation must exist in the primary key attribute of the referenced relation, that is, foreign key references the primary key attribute of referenced relation.

A `FOREIGN KEY` constraint can be applied as a column-level as well as table-level constraint. The syntax of `FOREIGN KEY` constraint when applied as a column-level constraint is given here.

```
<attribute> <data_type> REFERENCES <referenced_relation>
(<key_attribute>)
```

where, `key_attribute` is the primary key of the referenced relation

For example, the attribute `P_ID` of relation `BOOK` can be specified as foreign key, which refers to the primary key `P_ID` of relation `PUBLISHER` as shown here.

```
P_ID VARCHAR(4) REFERENCES PUBLISHER(P_ID)
```

When FOREIGN KEY constraint is applied on more than one attribute it is specified as table-level constraint. The syntax of FOREIGN KEY constraint defined on more than one attribute is given here.

```
FOREIGN KEY (<attribute_1 >) REFERENCES <referenced_relation>
(<key_attribute_1>)
:
FOREIGN KEY (<attribute_n >) REFERENCES <referenced_relation>
(<key_attribute_n>)
```

For example, attributes ISBN and R_ID of relation REVIEW can be declared as foreign keys as shown here.

```
FOREIGN KEY (ISBN) REFERENCES BOOK(ISBN)

FOREIGN KEY (R_ID) REFERENCES AUTHOR(A_ID)
```

All the constraints discussed here can be defined together for a relation. The CREATE TABLE command for the relations of *Online Book* database along with the required constraints is specified here.

```
CREATE TABLE PUBLISHER
   (    P_ID            VARCHAR(4),
        Pname           VARCHAR(50) NOT NULL,
        Address         VARCHAR(50),
        State           VARCHAR(15),
        Phone           VARCHAR(20),
        Email_id        VARCHAR(30),
        PRIMARY KEY(P_ID)
    );
CREATE TABLE BOOK
   (    ISBN            VARCHAR(15),
        Book_title      VARCHAR(50) NOT NULL,
        Category        VARCHAR(20),
```

```
        Price                   NUMERIC(6,2),
        Copyright_date          NUMERIC(4),
        Year                    NUMERIC(4),
        Page_count              NUMERIC(4),
        P_ID                    VARCHAR(4)   NOT NULL,
        CONSTRAINT Chk_price CHECK (Price BETWEEN 20 AND 200),
        PRIMARY KEY(ISBN),
        FOREIGN KEY(P_ID) REFERENCES PUBLISHER(P_ID)
 );
CREATE TABLE AUTHOR
   (    A_ID                    VARCHAR(4),
        Aname                   VARCHAR(30) NOT NULL,
        State                   VARCHAR(15),
        City                    VARCHAR(15),
        Zip                     VARCHAR(10),
        Phone                   VARCHAR(20),
        URL                     VARCHAR(30),
        PRIMARY KEY (A_ID)
   );
CREATE TABLE AUTHOR_BOOK
  (     A_ID                    VARCHAR(4)   NOT NULL,
        ISBN                    VARCHAR(15)  NOT NULL,
        FOREIGN KEY(A_ID) REFERENCES AUTHOR(A_ID),
        FOREIGN KEY(ISBN) REFERENCES BOOK(ISBN)
   );
CREATE TABLE REVIEW
   (    R_ID                    VARCHAR(4)   NOT NULL,
        ISBN                    VARCHAR(15) NOT NULL,
        Rating                  NUMERIC(2),
        PRIMARY KEY(R_ID, ISBN),
        CONSTRAINT Chk_rating CHECK (Rating BETWEEN 1 AND 10),
        FOREIGN KEY(R_ID) REFERENCES AUTHOR(A_ID),
        FOREIGN KEY(ISBN) REFERENCES BOOK(ISBN)
   );
```

5.2.5 ALTER TABLE Command

Once the relation is created, it might be required to make changes in the structure of a relation as per the needs of a user. The changes can be in the form of adding a new attribute, redefining attribute, or dropping attribute from a relation. To meet such requirements, `ALTER TABLE` command is used. In general, `ALTER TABLE` command is used for the following requirements.

- to add an attribute
- to modify the definition of an attribute
- to drop an attribute
- to add a constraint
- to drop a constraint
- to rename attribute and relation

Adding an Attribute

The new attribute can be added to the relation by using the `ADD` clause of `ALTER TABLE` command. The syntax to add new attribute in an existing relation is given here.

```
ALTER TABLE <table_name> ADD <attribute> <data_type>;
```

For example, the command to add a new attribute `Pname` in the relation `BOOK` can be specified as

```
ALTER TABLE BOOK ADD Pname VARCHAR(10);
```

If no default clause is specified, this attribute will have *null* values in all the tuples, hence `NOT NULL` constraint is not allowed in this case.

Modifying an Attribute

Any attribute of a relation can be modified by using the `MODIFY` clause of `ALTER TABLE` command. It can be used to change either its data type or size or both. The attribute to be modified must be empty before

modification. The syntax to modify an attribute of a relation is given here.

```
ALTER TABLE <table_name> MODIFY <attribute> <data_type>;
```

For example, the command to change the data type and size of the attribute `Pname` can be specified as

```
ALTER TABLE BOOK MODIFY Pname VARCHAR(50);
```

Dropping an Attribute

Sometimes, it is required to remove attribute, which is no longer required from a relation. The `DROP COLUMN` clause of `ALTER TABLE` command can be used to remove undesirable attributes from a relation. The syntax to remove an attribute from an existing relation is given here.

```
ALTER TABLE <table_name> DROP COLUMN <attribute>;
```

For example, the command to remove the attribute `Pname` from the relation `BOOK` can be specified as

```
ALTER TABLE BOOK DROP COLUMN Pname;
```

Whenever a particular attribute is dropped, the data stored in that attribute and its associated constraints are dropped.

Adding a Constraint

Different types of constraints can be added to the definition of an already existing relation. The syntax to add a constraint using `ADD` clause is given here.

```
ALTER TABLE <table_name> ADD [CONSTRAINT <constraint_name>]
```

```
<Cons>;
```

where,

`CONSTRAINT` is a keyword

`constraint_name` is a name given to constraint

`Cons` can be any of the constraints discussed earlier

Consider the following examples, in which different types of constraints are added for the different attributes of `BOOK` relation.

```
ALTER TABLE BOOK ADD CHECK (Book_title <> '');

ALTER TABLE BOOK ADD CONSTRAINT Cons_1 UNIQUE (ISBN);

ALTER TABLE BOOK ADD FOREIGN KEY (P_ID) REFERENCES PUBLISHER
```

The `NOT NULL` constraint is added differently as it cannot be specified as table constraint. For example, the command to apply this constraint on the attribute `Page_count` can be specified as

```
ALTER TABLE BOOK ALTER COLUMN Page_count SET NOT NULL ;
```

It is also possible to modify the definition of attribute by defining a new default value for an attribute, as shown here.

```
ALTER TABLE BOOK ALTER COLUMN Category SET DEFAULT 'Novel';
```

Dropping a Constraint

Any of the constraint defined can be dropped, provided it is given a name when specified. For example, the constraint `Cons_1` specified on the

attribute ISBN, can be dropped as shown here.

```
ALTER TABLE BOOK DROP CONSTRAINT Cons_1;
```

In addition, the DEFAULT and NOT NULL constraint can be dropped, as shown here.

```
ALTER TABLE BOOK ALTER COLUMN Category DROP DEFAULT ;

ALTER TABLE BOOK ALTER COLUMN Page_count DROP NOT NULL ;
```

Renaming an Attribute

The name of an attribute of a relation can be modified by using the RENAME COLUMN ... TO clause. The syntax to modify the name of an attribute is given here.

```
ALTER TABLE <table_name> RENAME COLUMN <old_name> TO <new_na
```

For example, the command to modify the name of an attribute Page_count can be specified as

```
ALTER TABLE BOOK RENAME COLUMN Page_count TO P_count;
```

Renaming a Relation

In addition to renaming an attribute, the name of a relation can also be modified using the RENAME TO clause. The syntax to rename a relation is given here.

```
ALTER TABLE <old_table_name> RENAME TO <new_table_name>;
```

For example, the command to rename the relation BOOK to new name can be specified as

```
ALTER TABLE BOOK RENAME TO Book_detail;
```

5.2.6 DROP TABLE Command

The DROP TABLE command is used to remove an already existing relation, which is no more required as a part of a database. The syntax to remove a relation is given here.

```
DROP TABLE <table_name>;
```

For example, the command to remove the relation Book_detail can be specified as

```
DROP TABLE Book_detail;
```

As a result of this command, all the tuples stored in Book_detail as well as relation itself is permanently removed and cannot be recovered. Hence, the relation Book_detail cannot be referred to for any purpose.

The two clauses that can be used with DROP TABLE command are CASCADE and RESTRICT. If CASCADE clause is used, all constraints and views that reference the relation to be removed are also dropped automatically. On the other hand, the RESTRICT clause, prevents a relation to be dropped if it is referenced by any of the constraints or views. These clauses can be used with DROP TABLE command as shown here.

```
DROP TABLE Book_detail CASCADE ;
```

```
DROP TABLE Book_detail RESTRICT ;
```

5.3 DATA MANIPULATION LANGUAGE

Data manipulation language (DML) commands are used for retrieving and manipulating data stored in the database. Basically it comprises of different actions, which are given here.

- retrieval of data stored in the database
- insertion of data into the database
- modification of data stored in the database
- deletion of data stored in the database

The basic retrieval and manipulating commands are `SELECT`, `INSERT`, `UPDATE`, and `DELETE`. For these commands, the relation must already exist.

5.3.1 SELECT Command

The `SELECT` command is the only data retrieval command in SQL. It is used to retrieve the subset of tuples or attributes from one or more relations. There are different ways and combinations in which `SELECT` command can be used. The syntax of SQL command is given here.

```
SELECT <attribute1>, <attribute2>;, …, <attributen >
FROM <table_name>;
```

Here, $\texttt{attribute}_i$ is an attribute of relation `table_name`. For example, the command to retrieve the attributes `ISBN`, `Book_title`, and `Category` from relation `BOOK` can be specified as

```
SELECT ISBN, Book_title, Category
FROM BOOK;
```

The order of the attributes appearing in the command can be different from the order of attributes in the relation. For example, the command to retrieve the attributes `ISBN`, `Category`, `Book_title`, `Page_count`, and `Price` can be specified as

```
SELECT ISBN, Category, Book_title, Page_count, Price
FROM BOOK;
```

In this example, the order of attributes is different from the order in which they are appearing in the relation. When all the attributes of a relation has to be retrieved, then instead of specifying list of all attributes in the SELECT command, the symbol asterisk (*) denoting "all attributes" can be used as shown here.

```
SELECT *
FROM BOOK;
```

The result of the SQL command may consist of duplicate tuples, as SQL does not automatically eliminate duplicate tuples from the resultant relations. For example, the command to retrieve Category of books from the relation BOOK can be specified as

```
SELECT Category
FROM BOOK;
```

The result of this command consists of tuples having duplicate values for the attribute Category. To eliminate duplicate tuples from the resultant relation, the keyword DISTINCT is used in SELECT command. Hence, the command to display only the unique values for the attribute Category can be specified as

```
SELECT DISTINCT Category
FROM BOOK;
```

Instead of DISTINCT keyword, if the keyword ALL is specified, the result retains the duplicate tuples. The result is same as when neither DISTINCT nor ALL keywords are specified. The ALL keyword can be used as shown

here.

```
SELECT ALL Category
FROM BOOK;
```

The SELECT command can be used to perform simple numeric computations on the data stored in a relation. SQL allows using scalar expressions and constants along with the attribute list. For example, the command to display the incremented value of price of books by 10% along with the attributes Book_title, Category, and Price can be specified as

```
SELECT Book_title, Category, Price, Price+Price*.10
FROM BOOK;
```

WHERE clause

The real life database might consist of a large number of tuples and it is not required to view all the tuples every time. Moreover, most of the time only subset of tuples is required to be viewed or processed. The criteria to determine the tuples to be retrieved is specified by using the WHERE clause. The syntax of SELECT command with WHERE clause is given here.

```
SELECT <attribute1>, <attribute2>;, …, <attributen >
FROM <table_name>
WHERE <condition>;
```

When a WHERE clause is present, the entire relation is scanned, one tuple at a time to determine whether it satisfies the condition or not. A particular tuple is retrieved only if it satisfies the condition specified in WHERE clause. For example, the command to retrieve the attributes Book_title, Category, and Price of those books from BOOK relation whose Category is *Novel* can be specified as

```
SELECT Book_title, Category, Price
FROM BOOK
WHERE Category = 'Novel';
```

The relational operators (=, <>, <, <=, >, >=) can be used to specify the conditions in the WHERE clause for selecting tuples from a relation. Moreover, more than one condition can be concatenated to give a more specific condition by using any of the logical operators AND, OR, and NOT. For example, the command to retrieve Book_title and Price of those books whose

Category is *Language Book* and Page_count is greater than *400* from BOOK relation can be specified as

```
SELECT Book_title, Price
FROM BOOK
WHERE Category = 'Language Book' AND Page_count>400;
```

Consider the queries covered in Chapter 04. These queries can be expressed here using the SELECT command along with the WHERE clause as shown here.

Query 1: Retrieve city, phone, and url of the author whose name is *Lewis Ian*.

```
SELECT City, Phone, URL
FROM AUTHOR
WHERE Aname = 'Lewis Ian';
```

Query 2: Retrieve name, address, and phone of all the publishers located in *New York* state.

```
SELECT Pname, Address, Phone
FROM PUBLISHER
WHERE State = 'New York';
```

Query 3: Retrieve title and price of all the textbooks with page count greater than *600*.

```
SELECT Book_title, Price
FROM BOOK
WHERE Category = 'Textbook' AND Page_count>600;
```

Query 4: Retrieve ISBN, title, and price of the books belonging to either novel or language book category.

```
SELECT ISBN, Book_title, Price
FROM BOOK
WHERE Category = 'Novel' OR Category = 'Language Book';
```

BETWEEN Operator

The `BETWEEN` comparison operator can be used to simplify the condition in `WHERE` clause that specifies numeric values based on ranges. For example, the command to retrieve details of all the books with price between *20* and *30* can be specified as

```
SELECT *
FROM BOOK
WHERE Price BETWEEN 20 AND 30;
```

Similarly, the `NOT BETWEEN` operator can also be used to retrieve tuples that do not belong to the specified range.

IN operator

The `IN` operator is used to specify a list of values. The `IN` operator selects values that match any value in a given list of values. For example, the command to retrieve the book details belonging to the categories *Textbook* or *Novel* can be specified as

```
SELECT *
FROM BOOK
WHERE Category IN ('Textbook', 'Novel');
```

Similarly, NOT IN operator can also be used to retrieve tuples that do not belong to the specified list.

Aliasing and Tuple Variables

The attributes of a relation can be given alternate name using AS clause in SELECT command. Note that the alternate name is provided within the query only. For example, consider the command given here.

```
SELECT Book_title AS Title, P_ID AS Publisher_ID
FROM BOOK;
```

In this command, attribute Book_title is renamed as Title and P_ID as Publisher_ID. An AS clause can also be used to define tuple variables. A tuple variable is associated with a particular relation and is defined in the FROM clause. For example, to retrieve details of publishers publishing books of language book category, the command can be specified as

```
SELECT P.P_ID, Pname, Address, Phone
FROM BOOK AS B, PUBLISHER AS P
WHERE P.P_ID = B.P_ID AND Category = 'Language Book';
```

In this command, tuple variables B and P are defined which are associated with the relations BOOK and PUBLISHER, respectively. The attribute publisher ID is present in both the relations with same name P_ID. Thus, to prevent ambiguity, the attribute name is qualified with the corresponding tuple variable separated by the dot (.) operator.

Tuple variable is especially useful for comparing two tuples from the same relation. For example, the command to retrieve title and price of

books belonging to the same category as the book titled *C++* can be specified as

```
SELECT B1.Book_title, B1.Price
FROM BOOK AS B1, BOOK AS B2
WHERE B1.Category = B2.Category AND B2.Book_title = 'C++';
```

NOTE The relation name itself is the implicitly defined tuple variable, if no other tuple variable is defined for that relation.

String Operations

In SQL, strings are specified by enclosing them in single quotes. Pattern matching is the most commonly used operation on strings. SQL provides `LIKE` operator, which allows to specify comparison conditions on only parts of the strings. Different patterns are specified by using two special wildcard character, namely, *percent* (`%`) and *underscore* (_). The `%` character is used to match substring with any number of characters, whereas, _ is used to match substring with only one character. Pattern matching is case sensitive as uppercase characters are considered different from lowercase characters. That is, the character *s* is different from the character *S*. To understand, how pattern matching is handled, consider these examples.

- `'A%'` matches any string beginning with character *A*
- `'%A'` matches any string ending with character *A*
- `'A%A'` matches any string beginning and ending with character *A*
- `'%AO%'` matches any string with character *AO* appearing anywhere in a string as substring
- `'A_ _'` matches any string beginning with character *A* and followed by exactly two characters
- `'_ _A'` matches any string ending with character *A* and preceded by exactly two characters
- `'A_ _ _D'` matches any string beginning with character *A*, ending with the character *D* and with exactly three characters in between

Consider the following queries to illustrate the use of `LIKE` operator.

The command to retrieve details of all the authors, whose name begins with the characters *Jo* can be specified as

```
SELECT *
FROM AUTHOR
WHERE Aname LIKE 'Jo%';
```

The command to retrieve details of all the authors, whose name ends with the characters *in* can be specified as

```
SELECT *
FROM AUTHOR
WHERE Aname LIKE '%in';
```

The command to retrieve details of all the authors, whose name begins with the characters *Jo* and ends with the characters *in* can be specified as

```
SELECT *
FROM AUTHOR
WHERE Aname LIKE 'Jo%in';
```

The command to retrieve details of all the authors, whose name begins with the characters *James* followed by exactly five characters, can be specified as

```
SELECT *
FROM AUTHOR
WHERE Aname LIKE 'James_ _ _ _ _';
```

The special pattern characters (`%` and _) can be included as a literal in the string by preceding the character by an escape character. The escape

character is specified after the string using the `ESCAPE` keyword. Any character not appearing in a string to be matched can be defined as an escape character. For example, consider the following patterns, in which backslash (\) is used as an escape character.

- `LIKE 'A\%b%' ESCAPE '\'`, searches the strings beginning with the characters *A%b*
- `LIKE 'A\_b%' ESCAPE '\'`, searches the strings beginning with the characters *A_b*
- `LIKE 'A\\b%' ESCAPE '\'`, searches the strings beginning with the characters *A\b*

Things to Remember

SQL allows to perform various functions on character strings like concatenation of strings (using '||' operator), extracting substrings, computing length of a string, converting case of strings uppercase (using `UPPER()` and `LOWER()` function), etc.

In order to search the strings not matching the patterns defined, `NOT LIKE` operator can be used. For example, the command to retrieve details of all the authors, whose name does not begins with the characters *Jo* can be specified as

```
SELECT *
FROM AUTHOR
WHERE Aname NOT LIKE 'Jo%';
```

Set Operations

As discussed in Chapter 04, there are various set operations that can be performed on relations, like, union, intersection, and difference. In SQL, these operations are implemented using `UNION, INTERSECT`, and `MINUS` operations, respectively.

The `UNION` operation is used to retrieve tuples from more than one relation and it also eliminates duplicate records. For example, the command to find the union of all the tuples with `Price` less than *40* and all the tuples with `Price` greater than *30* from the `BOOK` relation can be specified as

```
(SELECT *
FROM BOOK
WHERE Price<40)
UNION
(SELECT *
FROM BOOK
WHERE Price>30);
```

The `INTERSECT` operation is used to retrieve common tuples from more than one relation. For example, the command to find the intersection of all the tuples with `Price` less than *40* and all the tuples with `Price` greater than *30* from the `BOOK` relation can be specified as

```
(SELECT *
FROM BOOK
WHERE Price<40)
INTERSECT
(SELECT *
FROM BOOK
WHERE Price>30);
```

The `MINUS` operation is used to retrieve those tuples present in one relation, which are not present in other relation. For example, the command to find the difference (using `MINUS` operation) of all the tuples with `Price` less than *40* and all the tuples with `Price` greater than *30* from the `BOOK` relation can be specified as

```
(SELECT *
FROM BOOK
```

```
WHERE Price<40)
MINUS
(SELECT *
FROM BOOK
WHERE Price>30);
```

The UNION, INTERSECT, and MINUS operations automatically eliminate the duplicate tuples from the result. However, if all the duplicate tuples are to be retained, the ALL keyword can be used with these operations. For example, the command to retain duplicate tuples during UNION operation can be specified as

```
(SELECT *
FROM BOOK
WHERE Price<40)
UNION ALL
(SELECT *
FROM BOOK
WHERE Price>30);
```

Similarly, ALL keyword can be used along with the INTERSECT and MINUS operations.

NOTE During all the set operations, the resultant relations of both the participating SELECT commands must be union compatible. That is, both the resultant relations must have same number of attributes having similar data types and order.

Ordering the Tuples

Sometimes it may be required to display tuples arranged in a sorted order. SQL provides the ORDER BY clause to arrange the tuples of relation in some particular order. The tuples can be arranged on the basis of the values of one or more attributes. For example, the command to display the tuples of the relation BOOK, on the basis of Price attribute can be specified as

```
SELECT *
FROM BOOK
ORDER BY Price;
```

By default, the `ORDER BY` clause arranges the tuples in ascending order on the basis of values of specified attribute. In addition, the `ASC` keyword can also be used to explicitly mention the order to be ascending. However, the tuples can be sorted in descending order by using the `DESC` keyword. For example, the command to sort tuples of the relation `BOOK` on the basis of `Price` attribute in the descending order can be specified as

```
SELECT *
FROM BOOK
ORDER BY Price DESC ;
```

Tuples can also be sorted on the basis of more than one attribute. This can be done by specifying more than one attribute in `ORDER BY` clause. In `ORDER BY` clause, more than one attributes can also be specified on the basis of which tuples are to be sorted. For example, to arrange the sort of the relation `BOOK`, on the basis of two attributes `Category` and `Price`, the command can be specified as

```
SELECT *
FROM BOOK
ORDER BY Category, Price;
```

This command first arranges the tuples in ascending order on the basis of the `Category` attribute and then the tuples within a particular category are arranged in the ascending order of the `Price` attribute. The order of two attributes need not be the same. They can be different. For example, to arrange the tuples of relation `BOOK` in the descending order of `Category` and then in the ascending order of `Price`, the command can be specified as

```
SELECT *
FROM BOOK
ORDER BY Category DESC, Price ASC ;
```

5.3.2 INSERT Command

As discussed earlier, the CREATE TABLE command only defines the structure of a relation and tuples can be inserted in the relation using the INSERT command. This command adds a single tuple at a time in a relation. The syntax to add a tuple in a relation is given here.

```
INSERT INTO <table_name> [(<attribute_list>)]
VALUES (<value_list>);
```

The value_list is the list of values to be inserted in the corresponding attributes listed in attribute_list. The values should be specified in the same order in which the attributes are listed in attribute_list. In addition, the data type of the corresponding values and the attributes must be same. For example, the command to add a tuple in the relation BOOK can be specified as

```
INSERT INTO BOOK (ISBN, Book_title, Category, Price,
Copyright_date, Year, Page_count, P_ID)

VALUES ('001-987-760-9', 'C++', 'Textbook', 40, 2004, 2005,
800, 'P001');
```

In this example, the order and the data type of attributes and corresponding values are same. Alternatively, tuple can be inserted into a relation by omitting the attribute list from the command. That is, the tuple can also be inserted as shown here.

```
INSERT INTO BOOK

VALUES ('001-987-760-9', 'C++', 'Textbook', 40, 2004, 2005,
```

```
800, 'P001');
```

The INSERT command also allows user to insert data only for the selected attributes. That is, in INSERT command, values for some of the attributes can be skipped. The attributes skipped in the command are provided with the default values defined for them, otherwise *null* values are inserted. In this case, the attribute list must be provided. For example, the command to add values for the selected attributes of the relation PUBLISHER can be specified as

```
INSERT INTO PUBLISHER (P_ID, Pname, Address, Phone)

VALUES ('P002', 'Sunshine Publishers Ltd.', '45, Second
street, Newark', '6548909');
```

In this example, the values for the attributes State and Email_id are skipped. Hence, these attributes are provided either with the default values or *null* values.

5.3.3 UPDATE Command

Sometimes, there may be a situation to make changes in the values of attributes of the relations. SQL provides UPDATE command for this type of requirement. The SET clause in the UPDATE command specifies the attributes to be modified and the new values to be assigned to them. More than one attribute can be specified in the SET clause separated by comma. The WHERE clause is also required to specify the tuples for which the attributes are to be modified, otherwise value for all the tuples in the relation will be modified. While modifying the values of attributes all the constraints must be satisfied, otherwise the update is not done. For example, the command to modify the City to *New York* for the author whose name is *Lewis Ian* can be specified as

```
UPDATE AUTHOR
SET City = 'New York'
WHERE Aname = 'Lewis Ian';
```

The command to modify `State` and `Phone` for the publisher *Bright Publications* can be specified as

```
UPDATE PUBLISHER
SET State = 'Georgia', Phone = '27643676'
WHERE Pname = 'Bright Publications';
```

The command to increment the price of all the books by 10 per cent can be specified as

```
UPDATE BOOK
SET Price = Price + 0.10 * Price;
```

5.3.4 DELETE Command

The tuples, which are no more required as a part of a relation, can be removed from the relation using the `DELETE` command. Tuples can be deleted from only one relation at a time. The condition in the `WHERE` clause of `DELETE` commands is used to specify the tuples to be deleted. If `WHERE` clause is omitted, all the tuples of a relation are deleted; however, the relation remains in the database as an empty relation. For example, to delete tuples of the `BOOK` relation, whose `Page_Count` is less than *50*, the command can be specified as

```
DELETE FROM BOOK
WHERE Page_count <50;
```

Consider another example, to delete those tuples from the `BOOK` relation, whose category is *Novel*, the command can be specified as

```
DELETE FROM BOOK
WHERE Category = 'Novel';
```

5.4 COMPLEX QUERIES IN SQL

The queries discussed so far are some of the simple queries in SQL. In addition to these, complex queries can be specified using additional features of SQL. Some of these additional features are discussed in this section.

5.4.1 Null Values

SQL allows attribute to have *null* values. The *null* value usually represents missing value having one of three interpretations—value is unknown, value is not available, or attribute is not applicable for particular tuple. However, SQL does not distinguish between the different meanings of *null*. The *null* value in an attribute for a relation can be searched using the `IS NULL` predicate in the `WHERE` clause. In all cases, *null* value for an attribute is checked with same syntax. For example, the command to retrieve the details of publishers not having email ID can be specified as

```
SELECT *
FROM PUBLISHER
WHERE Email_id IS NULL ;
```

The predicate `IS NOT NULL` can be used to check whether an attribute contains non-null value. For example, to retrieve the details of publishers having email ID (`Email_id` is not *null*), the command can be specified as

```
SELECT *
FROM PUBLISHER
WHERE Email_id IS NOT NULL ;
```

In addition, the conditions in `WHERE` clause may involve logical operators `AND`, `OR`, and `NOT`. Hence, when *null* value is involved, the definition of the logical operators is extended with new value *unknown*. In case of `NOT` operator, the condition, say `A`, whose value is *unknown*, results in *unknown* value. Whereas, the `AND` and `OR` operator involves two (or more)

conditions. For this, consider a WHERE clause involving two conditions A and B, and the result of any one of them, say B, is *unknown*, then the condition evaluates as follows:

- Conditions involving AND operator:
  - If the value of A is *true* or *unknown*, the result is *unknown*.
  - If the value of A is *false*, the result is *false*.
- Conditions involving OR operator:
  - If the value of A is *false* or *unknown*, the result is *unknown*.
  - If the value of A is *true*, the result is *true*.

5.4.2 Aggregate Functions

Aggregate functions process set of values taken as input and return a single value as a result. SQL provides five built-in aggregate functions, namely, AVG, MIN, MAX, SUM and COUNT.

The SUM and AVG functions works for numeric values only, whereas other functions can work for numeric as well as non-numeric values, like strings, date, time, etc. For example, the command to find the average price of books in BOOK relation can be specified as

```
SELECT AVG (Price)
FROM BOOK;
```

This command calculates the average value of price of all the books. To calculate the average price of only textbooks, the command can be specified as

```
SELECT AVG (Price)
FROM BOOK
WHERE Category='Textbook';
```

Consider another example to find the maximum price of the book

belonging to *Novel* category, the command can be specified as

```
SELECT MAX (Price)
FROM BOOK
WHERE Category='Novel';
```

The `COUNT(*)` function is used to count the total number of tuples in the resultant relation, whereas, `COUNT()` is used to count the number of non-null values in a particular attribute. For example, the command to find the total number of tuples in the relation `PUBLISHER` can be specified as

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

To find the number of non-null values in the attribute `Email_id`, the command can be specified as

```
SELECT COUNT (Email_id)
FROM PUBLISHER;
```

Consider another example, to find the number of non-null values in the attribute `Category` of `BOOK` relation, the command can be specified as

```
SELECT COUNT (Category)
FROM BOOK;
```

This command counts the total number of non-null values in the attribute `Category`, including duplicate values also. However, if duplicate values are to be eliminated, the `DISTINCT` keyword can be used and the command can be specified as

```
SELECT COUNT (DISTINCT Category)
FROM BOOK;
```

5.4.3 GROUP BY and HAVING Clause

In many situations, it is required to apply the aggregate function on a group of tuples from a relation rather on the whole relation. The tuples in the relation can be divided on the basis of the values of one or more attribute. The tuples belonging to a particular group have same value for the attribute on the basis of which grouping is done. The GROUP BY clause can be used in SELECT command to divide the relation into groups on the basis of values of one or more attributes. After dividing the relation into groups, the aggregate functions can be applied on the individual group independently. The aggregate functions are performed separately for each group and return the corresponding result value separately. For example, the command to calculate average price for each category of book in the BOOK relation can be specified as

Learn More

Using GROUP BY clause, one can create groups within groups known as nested grouping. An Up to 10 level of nesting is allowed in a GROUP BY expression.

```
SELECT AVG (Price)
FROM BOOK
GROUP BY Category;
```

To calculate maximum, minimum, and average price of the books published by each publisher, the command can be specified as

```
SELECT MAX (Price), MIN (Price), AVG (Price)
FROM BOOK
GROUP BY P_ID;
```

Conditions can be placed on the groups using HAVING clause. Note that the HAVING clause places conditions on groups, whereas, WHERE clause is

used to place conditions on the individual tuples. Another difference between WHERE and HAVING clause is that, conditions specified in the WHERE clause cannot include aggregate functions, whereas, HAVING clause can. For example, the command to retrieve the book categories for which number of books published is less than *5* can be specified as

```
SELECT Category, COUNT (*)
FROM BOOK
GROUP BY Category
HAVING COUNT (*) <5;
```

More than one condition can be specified in the HAVING clause by using logical operators. For example, the command to retrieve average price and average page count for each category of books with average price greater than *30* and average page count less than *900* can be specified as

```
SELECT Category, AVG (Price), AVG (Page_count)
FROM BOOK
GROUP BY Category
HAVING AVG (Price)>30 AND AVG (Page_count)<900;
```

Consider another example, the command to retrieve average price for each category of book with minimum price greater than *30* can be specified as

```
SELECT Category, AVG (Price)
FROM BOOK
GROUP BY Category
HAVING MIN (Price)>30;
```

The IN and the BETWEEN operators can also be used in the HAVING clause. To understand the use of these operators, consider the commands given here.

```
SELECT Category, AVG (Price), MIN (Page_count)
FROM BOOK
GROUP BY Category
HAVING Category IN ('Textbook', 'Novel');

SELECT Category, MIN (Price), MAX (Price)
FROM BOOK
GROUP BY Category
HAVING AVG (Page_count) BETWEEN 300 AND 1000;
```

While using `GROUP BY` and `HAVING` clause, some errors may occur. These errors result from the illegal use of group queries. Some of the possible illegal use of group queries is discussed here.

- Combination of non-group and group expressions without using `GROUP BY` clause.

  For example, consider the command given here.

  ```
  SELECT Book_title, AVG (Price)
  FROM BOOK;
  ```

This command includes non-group attribute `Book_title` and a group function `AVG(Price)`, which is illegal and results in an error.

- Combination of non-group and group expressions with `GROUP BY` clause.

  For example, consider the command given here.

  ```
  SELECT Book_title, Category, AVG (Price)
  FROM BOOK
  GROUP BY Category;
  ```

This command includes non-group attribute `Book_title` along with group

attribute `Category` and group expression `AVG(Price)`. This command generates an error, as `Book_title` cannot have same value for all books in a particular `Category`. This query can be made error free by omitting the attribute `Book_title`.

- Using group function with `WHERE` clause instead of `HAVING` clause.

  For example, consider the command given here.

```
SELECT Category, AVG (Price)
FROM BOOK
WHERE MIN (Price)>30
GROUP BY Category;
```

The `WHERE` clause operates on individual tuples, whereas, group function operates on multiple tuples or on the group of tuples. Hence, group functions cannot be used with `WHERE` clause.

5.4.4 Joins

A query that combines the tuples from two or more relations is known as **join query**. In such type of queries, more than one relation is listed in the FROM clause. The process of combining data from multiple relations is called **joining**. For example, consider the command given here.

```
SELECT *
FROM BOOK, PUBLISHER;
```

This command returns the cartesian product of the relations `BOOK` and `PUBLISHER`, that is, the resultant relation consists of all possible combinations of all tuples from both the relations. It returns relation with cardinality $c_1 * c_2$, where c_1 is cardinality of first relation and c_2 is cardinality of second relation. Only selected tuples can be included in the resultant relation by specifying condition on the basis of common attribute of both the relations. The condition on the basis of which

relations are joined is known as **join condition**. For example, the command to retrieve details of both book and publishers, where `P_ID` attribute in both the relations have identical values can be specified as

```
SELECT *
FROM BOOK, PUBLISHER
WHERE BOOK.P_ID=PUBLISHER.P_ID;
```

Using alias names for the relations can make this command simple, as shown here.

```
SELECT *
FROM BOOK AS B, PUBLISHER AS P
WHERE B.P_ID=P.P_ID;
```

This type of join query in which tuples are concatenated on the basis of equality condition is known as **equi-join query**. The resultant relation of this query consists of two columns for the attribute `P_ID` having identical values, one from `BOOK` relation and other from `PUBLISHER` relation. This can be avoided by explicitly specifying the name of attributes to be included in the resultant relation. Such type of command can be specified as

```
SELECT ISBN, Book_title, Price, B.P_ID, Pname
FROM BOOK AS B, PUBLISHER AS P
WHERE B.P_ID=P.P_ID;
```

As a result of this command only one column for the attribute `P_ID` from relation `BOOK` is displayed in the result. This type of query is known as **natural join query**. In addition, some additional conditions can also be given to make the selection of tuples more specific apart from the join condition. For example, consider Query 5 (Chapter 04) in which the command to retrieve `P_ID`, `Pname`, `Address`, and `Phone` of publishers publishing novels can be specified as

```
SELECT P.P_ID, Pname, Address, Phone
FROM BOOK AS B, PUBLISHER AS P
WHERE B.P_ID=P.P_ID AND Category = 'Novel';
```

Sometimes, it is required to extract information from more than two relations. In such a case, more than two relations are required to be joined through join query. This can be achieved by specifying names of all the required relations in `FROM` clause and specifying join conditions in `WHERE` clause using `AND` logical operator. For example, consider Query 8 (Chapter 04) in which the command to retrieve title, category and price of all the books written by author *Charles Smith* can be specified as

```
SELECT Book_title, Category, Price
FROM BOOK AS B, AUTHOR AS A, AUTHOR_BOOK AS AB
WHERE B.ISBN=AB.ISBN AND AB.A_ID=A.A_ID AND Aname = 'Charles
```

Consider some other examples of join queries given here.

Query 6: Retrieve title and price of all the books published by *Hills Publications*.

```
SELECT Book_title, Price
FROM BOOK AS B, PUBLISHER AS P
WHERE B.P_ID=P.P_ID AND Pname='Hills Publications';
```

Query 7: Retrieve book title, reviewers ID, and rating of all the textbooks.

```
SELECT Book_title, R_ID, Rating
FROM BOOK S B, REVIEW S R
WHERE B.ISBN=R.ISBN AND
Category = 'Textbook';
```

Query 9: Retrieve ID, name, url of author, and category of the book *C++*.

```
SELECT AB.A_ID, Aname, URL, Category
FROM BOOK S B, AUTHOR S A, AUTHOR_BOOK S AB
WHERE A.A_ID=AB.A_ID AND AB.ISBN=B.ISBN AND Book_title= 'C++
```

Query 10: Retrieve book title, price, author name, and url for the publishers *Bright Publications*.

```
SELECT Book_title, Price, Aname, URL
FROM BOOK S B, AUTHOR_BOOK S AB, AUTHOR S A, PUBLISHER S P
WHERE B.ISBN=AB.ISBN AND AB.A_ID=A.A_ID ND B.P_ID=P.P_ID AND
```

5.4.5 Nested Queries

The query defined in the `WHERE` clause of another query is known as **nested query** or **subquery**. The query in which another query is nested is known as enclosing query. The result returned by the subquery is used by the enclosing query for specifying the conditions. In general, several levels of nested queries can be defined. That is, query can be defined inside another query number of times.

Things to Remember

The two queries are said to be correlated when condition specified in the `WHERE` clause of the inner query refers to an attribute of a relation specified in enclosing query.

For example, the command to retrieve `ISBN`, `Book_title`, and `Category` of book with minimum price can be specified as

```
SELECT ISBN, Book_title, Category
FROM BOOK
WHERE PRICE = (SELECT MIN (Price)
               FROM BOOK);
```

In this command, first the nested query returns a minimum `Price` from

the BOOK relation, which is used by the enclosing query to retrieve the required tuples. Consider another example to retrieve ISBN, Book_title, and Category of book published by the publisher residing in the *New York* state.

```
SELECT ISBN, Book_title, Category
FROM BOOK
WHERE P_ID = (SELECT P_ID
              FROM PUBLISHER
              WHERE State = 'New York');
```

SQL provides four useful operators that are generally used with subqueries. These are ANY, ALL, and EXISTS.

- ANY operator compares a value with any of the values in a list or returned by the subquery. This operator returns *false* value if the subquery returns no tuple. For example, the command to retrieve details of books with price equal to any of the books belonging to *Novel* category can be specified as

  ```
  SELECT ISBN, Book_title, Price
  FROM BOOK
  WHERE Price = ANY (SELECT Price
                     FROM BOOK
                     WHERE Category = 'Novel');
  ```

In addition to ANY operator, the IN operator can also be used to compare a single value to the set of multiple values. For example, the command to retrieve the details of books belonging to category with page count greater than *300* can be specified as

```
SELECT *
FROM BOOK
WHERE Category IN (SELECT Category
                   FROM BOOK
                   WHERE Page_count >300);
```

In case the nested query returns a single attribute and a single tuple, the query result will be a single value. In such a situation, the = can be used instead of `IN` operator for the comparison.

- `ALL` operator compares a value to every value in a list returned by the subquery. For example, the command to retrieve details of books with price greater than the price of all the books belonging to *Novel* category can be specified as

```
SELECT ISBN, Book_title, Price
FROM BOOK
WHERE Price > ALL (SELECT Price
                   FROM BOOK
                   WHERE Category = 'Novel');
```

- `EXISTS` operator evaluates to *true* if a subquery returns at least one tuple as a result otherwise, it returns *false* value. For example, the command to retrieve the details of publishers having at least one book published can be specified as

```
SELECT P_ID, Pname, Address
FROM PUBLISHER
WHERE EXISTS (SELECT *
              FROM BOOK
              WHERE PUBLISHER.P_ID = BOOK.P_ID);
```

On the other hand, `NOT EXISTS` operator evaluates to *true* if subquery returns no tuple as a result. For example, the command to retrieve the details of publishers having not publishing any book can be specified as

```
SELECT P_ID, Pname, Address
FROM PUBLISHER
WHERE NOT EXISTS (SELECT *
                  FROM BOOK
                  WHERE PUBLISHER.P_ID = BOOK.P_ID);
```

5.5 ADDITIONAL FEATURES OF SQL

Some of the advanced features of SQL that help in specifying complex constraints and queries efficiently are assertions and views. Assertion is a condition that must always be satisfied by the database. Assertions do not fall into any of the categories of constraints discussed earlier. But, view is a virtual relation which derives its data from one or more existing relations. The result of view is not physically stored.

The two standard features of SQL that are related to programming in database are stored procedures and triggers. The stored procedures are program modules that are stored by the DBMS at the database server. These procedures can be invoked and executed whenever required using the single SQL statement from various applications that have access to the database. Whereas, triggers are type of stored procedures which are automatically invoked and are needed to perform complex data verification operations. This section discusses assertions, views, stored procedure, and triggers.

5.5.1 Assertions

Some of the simple data integrities can be specified while defining the structure of the relation with the help of various constraints like `PRIMARY KEY`, `NOT NULL`, `UNIQUE`, `CHECK`, etc. In addition, the referential integrity can be specified using the `FOREIGN KEY` constraint. However, there are some data integrities, which cannot be specified using any of the constraints discussed so far. Such type of constraints can be specified using the assertions. The DBMS enforces the assertion on the database and the constraints specified in assertion must not be violated. Assertions are always checked whenever modifications are done in corresponding relation.

The syntax to create an assertion can be specified as

```
CREATE ASSERTION <assertion_name>
CHECK (<condition>);
```

The assertion name is used to identify the constraints specified by the assertion and can be used for modification and deletion of assertion, whenever required. An assertion is implemented by writing a query that retrieves any tuples that violates the specified condition. Then this query is placed inside a `NOT EXISTS` clause, which indicates that the result of this query must be empty. Hence, the assertion is violated whenever the result of this query is not empty. For example, the price of textbook must not be less than the minimum price of novel, the assertion for this requirement can be specified as

```
CREATE ASSERTION price_constraint
CHECK (NOT EXISTS (
                    SELECT *
                    FROM BOOK
                    WHERE Category = 'Textbook' AND
                    (Price<(SELECT MIN (Price)
                    FROM BOOK
                    WHERE Category='Novel')
                )
    );
```

In this command, query retrieves tuples with category as *Textbook* and price less than the minimum price of book with *Novel* category. The result of this query must be empty; otherwise it will violate the assertion. DBMS tests the assertion for its validity when it is created. After its creation, future modification to the database is allowed only if it does not violate the assertion. Note that, the assertions should be used to specify complex constraints only that cannot be specified by any of the other constraints, as assertions introduce a significant amount of overhead in terms of time and cost.

5.5.2 Views

A view is a virtual relation, whose contents are derived from already existing relations and it does not exist in physical form. The contents of

view are determined by executing a query based on any relation and it does not form the part of database schema. Each time a view is referred to, its contents are derived from the relations on which it is based. A view can be used like any other relation, which is, it can be queried, inserted into, deleted from, and joined with other relations or views, though with some limitations on update operations. These are very useful in the situations where only parts of relations are to be accessed frequently instead of complete data.

The `CREATE VIEW` command of SQL can be used for creating views. This command provides the name to the view and specifies the list of attributes and tuples to be included using a subquery. The syntax to create a view is given here.

```
CREATE VIEW <view_name>
AS <subquery>;
```

For example, the command to create a view containing details of books which belong to `Textbook` and `Language Book` categories can be specified as

```
CREATE VIEW BOOK_1
AS SELECT *
   FROM BOOK
   WHERE Category IN ('Textbook', 'Language Book');
```

This command creates a view, named as `BOOK_1`, having details of books satisfying the condition specified in `WHERE` clause. The view created like this consists of all the attributes of `BOOK` relation; however, only selected attributes can also be included in the view and they can be given another name also. For example, consider the command given here.

```
CREATE VIEW BOOK_2(B_Code, B_Title, B_Category, B_Price)
AS SELECT ISBN, Book_title, Category, Price
```

```
    FROM BOOK
    WHERE Category IN ('Textbook', 'Language Book');
```

This command creates a view `BOOK_2`, which consists of the attributes, `ISBN`, `Book_title`, `Category`, and `Price` from the relation `BOOK` with new names, namely, `B_Code`, `B_Title`, `B_Category`, and `B_Price`, respectively. Now queries can be performed on these views as they are performed on the other relations. For example, consider the commands given here.

```
SELECT *
FROM BOOK_1;

SELECT *
FROM BOOK_2
WHERE B_Price > 30;

SELECT B_Title, B_Category
FROM BOOK_2
WHERE B_Price BETWEEN 30 AND 50;
```

Views can be based on more than one relation. For example, the command to create a view consisting of attributes `Book_title`, `Category`, `Price` and `P_ID` of `BOOK` relation, `Pname` and `State` of `PUBLISHER` relation can be specified as

```
CREATE VIEW BOOK_3
AS SELECT Book_title, Category, Price, BOOK.P_ID, Pname, Sta
   FROM BOOK, PUBLISHER
   WHERE BOOK.P_ID = PUBLISHER.P_ID;
```

The views that are based on more than one relation are said to be **complex views**. These types of views are inefficient as they are time consuming to execute, especially if multiple queries are involved in the view definition. Since their contents are not physically stored, they are

executed each and every time they are referred to. As an alternative to this, certain database systems allow views to be physically stored, also known as **materialized views**. These types of views save the time spent to execute the subqueries each and every time they are referred to. Such views are updated whenever tuples are inserted, deleted, or modified in the underlying relations. The view is stored temporarily, that is, if view is not queried for a certain period of time, it is physically removed and is recomputed when it is again referred in future. While implementing the concept of materialized views, their advantages must be compared with the cost incurred for storing them and their updations.

A view is updateable if it is based on single relation and the update query can be mapped on to the already existing relation successfully under certain conditions. Any view can be made non-updateable, by adding the `WITH READ ONLY` clause. That is, no `INSERT`, `UPDATE`, or `DELETE` command can be carried over that view.

Views can also be created using the queries based on other views. For example, the command to create view based on the view `BOOK_3`, having details, where publishers belong to the state *New York* can be specified as

```
CREATE VIEW BOOK_4
AS SELECT *
   FROM BOOK_3
   WHERE State = 'New York';
```

Like relation, view can also be deleted from the database by using `DROP VIEW` command. For example, to delete the view `BOOK_1` from the database, the command can be specified as

```
DROP VIEW BOOK_1;
```

5.5.3 Stored Procedures

Stored procedures are procedures or functions that are stored and executed by the DBMS at the database server machine. In SQL standard, stored procedures are termed as **persistent stored modules** (PSM), as these procedures, like data, are persistently stored by the DBMS. Stored procedures improve performance, as these procedures are compiled only at the time of their creation or modifications. Hence, whenever these procedures are called for the execution the time for compilation is saved. Stored procedures are beneficial when a procedure is required by different applications located at remote sites, as it is stored at server site and can be invoked by any of the applications. This eliminates duplication of efforts in writing SQL application logic and makes code maintenance easy.

Most of the DBMSs allow stored procedures to be written in any general-purpose programming language. As an alternative, stored procedures may consist of simple SQL commands like `INSERT`, `SELECT`, `UPDATE`, etc. Stored procedures can be compiled and executed with different parameters and results. They can accept input parameters and pass values to output parameters. A stored procedure can be created as shown here.

```
CREATE PROCEDURE <name> (<parameters1, …, parametersn >)
<local_declarations>
<body of procedure>;
```

Parameters and local declarations are optional and if declared must be of valid SQL data types. Parameters declared must have one of the three modes, namely, `IN`, `OUT`, or `INOUT` specified for it. Parameters specified with `IN` mode are arguments passed to the stored procedure and act like a constant, whereas, `OUT` parameters act like an un-initialized variables and they cannot appear on the right hand side of = symbol. Parameters with `INOUT` mode have combined properties of both `IN` and `OUT` mode. They contain values to be passed to the stored procedure and also can be assigned values to be returned from the stored procedure. For

example, the procedure to update the value of price of a book with a given ISBN of relation BOOK can be specified as

```
CREATE PROCEDURE Update_price (IN B_ISBN VARCHAR(15), IN New
NUMERIC(6,2))
UPDATE BOOK
SET Price = New_Price
WHERE ISBN = B_ISBN;
```

Functions are required when value is required to be returned to the calling program since procedures cannot return a value. The function can be created as shown here.

```
CREATE FUNCTION <name> (<parameters1, …, parameters n >)
RETURNS <return_type>
<local_declarations>
<body of function> ;
```

For example, the function returning a rating of book with a given ISBN and author ID can be specified as

```
CREATE FUNCTION Book_rating (IN B_ISBN VARCHAR(15),
IN Au_ID VARCHAR(4))
RETURNS NUMERIC(2)
DECLARE B_rating NUMERIC(2)
SET B_rating=(SELECT Rating
              FROM REVIEW
              WHERE ISBN=B_ISBN AND R_ID = Au_ID);
RETURN B_rating;
```

SQL/PSM

SQL/PSM is a part of SQL standard that includes programming constructs for writing the coding part of persistent stored modules. It also includes constructs for specifying conditional statements and

looping statements. The conditional statements in SQL/PSM can be specified as

```
IF <condition> THEN <statements>
     ELSEIF <condition> THEN <statements>
     :
     ELSEIF <condition> THEN <statements>
     ELSE <statements>
END IF ;
```

The while looping construct can be specified as

```
WHILE <condition> DO
     <statements>;
END WHILE ;
```

The repeat looping construct can be specified as

```
REPEAT
     <statements>;
UNTIL <condition>
END REPEAT ;
```

For example, a function that searches a book with a given `ISBN` and declares it to be of *High, Medium*, or *Low* price can be written as

```
CREATE FUNCTION B_price (IN B_ISBN VARCHAR(15))
RETURNS VARCHAR(7)
DECLARE Book_price NUMERIC(6,2)
SET Book_price=(SELECT Price
                FROM BOOK
                WHERE ISBN=B_ISBN);
IF Book_price>100 THEN RETURN 'High'
     ELSEIF Book_price>50 THEN RETURN 'Medium'
     ELSE RETURN 'Low'
END IF ;
```

Similarly, loops can also be used in the code of stored procedure. Moreover, loops can be named in the code. The name of loop can be used for breaking out of the loop based on some condition by using the statement `LEAVE <loop_name>`.

If the procedure or function is written in some general-purpose programming language, it is necessary to specify the language and the file name where program code is stored. In such a case, the procedure can be created as shown here.

```
CREATE PROCEDURE <procedure_name>(<parameters>)
LANGUAGE <name of programming language>
EXTERNAL NAME <file path name>;
```

Once the stored procedures or functions are created, they can be called in any application for execution. The calling statement can be specified as

```
CALL <procedure or function name> (<arguments>);
```

For example, the statements for calling procedures and functions can be specified as

```
CALL Update_price('003-456-433-6', 28);

CALL Book_rating('003-456-533-8', 'A004');

CALL B_price('001-987-760-9');
```

5.5.4 Triggers

A trigger is a type of stored procedure that is executed automatically when some database related events like, insert, update, delete, etc.,

occur. In addition, unlike procedures, triggers do not accept any arguments. The main aim of triggers is to maintain data integrity and also one can design a trigger for recording information, which can be used for auditing purposes. The triggers are mainly needed for following purposes.

- Implementing and maintaining complex integrity constraints.
- Recording the changes for auditing purposes.
- Automatically passing signal to other programs that action needs to be taken whenever specific changes are made to a relation.

Triggers are usually defined on relations. However, it can be defined on views and can also be used to execute other triggers, procedures, and functions. Although triggers like constraints are defined to maintain the database integrity, yet they are different from constraints in the following ways.

- Triggers affect only those tuples that are inserted after the trigger is enabled, whereas, constraints affect all tuples in the relation.
- Triggers allow enforcing the user-defined constraints, which are not possible through standard constraints supported by database.

A database having triggers associated with it is known as **active database**. A trigger consists of three parts.

- **Event:** any change in the database that activates the trigger.
- **Condition:** when the trigger is activated, the condition is checked.
- **Action:** a procedure that is to be executed, whenever the trigger is activated and the corresponding condition is satisfied.

In other words, a trigger is an event-condition-action rule that states that whenever a specified event occurs and the condition is satisfied, the corresponding action must be executed. The trigger can be created as shown here.

```
CREATE TRIGGER <trigger_name>
```

```
[BEFORE or AFTER] [INSERT or UPDATE or DELETE] ON <relation_
[FOR EACH ROW]
WHEN <condition>
<statements>;
```

For example, when value in `Phone` attribute of new inserted tuple of a relation `AUTHOR` is empty, indicating absence of phone number, the trigger to insert a *null* value in this attribute can be specified as

```
CREATE TRIGGER Setnull_phone BEFORE INSERT ON AUTHOR
REFERENCING NEW ROW AS nr
FOR EACH ROW
WHEN nr.Phone = ' '
SET nr.Phone = NULL;
```

The `FOR EACH ROW` clause makes sure that trigger is executed for every single tuple processed. Such type of trigger is known as **row level trigger**. Whereas, the trigger, which is executed only once for specified statement, regardless of the number of tuples being effected as a result of that statement, is known as **statement level trigger**. The `FOR EACH STATEMENT` clause specifies the trigger as statement level trigger. For example, trigger defined for a `INSERT` command, will be executed only once, regardless of the number of tuples inserted through single `INSERT` statement. The statement level triggers are the default type of triggers, which are identified by omitting the `FOR EACH ROW` clause.

Learn More

Triggers are provided with a separate namespace from procedures, package, relations (sharing the same namespace), which means that triggers can have same name as of a relation or procedure.

The `REFERENCING NEW ROW AS` clause can be used to create a variable for storing the new values of an updated tuple. Similarly, `REFERENCING OLD ROW AS` clause can be used to create variables for storing old values of an

updated tuple. In addition, the clauses `REFERENCING NEW TABLE AS` or `REFERENCING OLD TABLE AS` can be used to refer to temporary relation consisting of all the affected tuples. Such relations can be used with only `AFTER` triggers and not `BEFORE` triggers.

Consider another example, to create the trigger for changing value of `Price` attribute to a default value *$60*, if the value entered by the user exceeds the upper limit of *$200* in case of insertion in relation `BOOK`. The trigger can be specified as

```
CREATE TRIGGER New_price BEFORE INSERT ON BOOK
FOR EACH ROW
WHEN (new.Price>200)
BEGIN
     new.Price = 60;
END
```

The keyword `new` and `old` can be used to refer to the values after and before the modifications are made. Triggers can be enabled and disabled by using the `ALTER TRIGGER` command. The triggers which are not required can be removed by using the `DROP TRIGGER` command. For example, consider the following statements.

```
ALTER TRIGGER New_price DISABLE ;

ALTER TRIGGER New_price ENABLE ;

DROP TRIGGER New_price;
```

The triggers come with lot of advantages. However, triggers must be used with great care as sometimes action of one trigger can lead to the activation of another trigger. Such triggers are known as **recursive triggers**. In the worst situation, it can lead to an infinite chain of triggering. A database system typically limits the length of such chains of

triggers to 16 or 32 and considers longer chains of triggering an error.

5.6 ACCESSING DATABASES FROM APPLICATIONS

So far, we have discussed how queries are executed in SQL. In most of the situations, databases are accessed through software programs implementing database applications. These types of software are usually developed in general-purpose programming languages, such as C, Java, COBOL, Pascal, and FORTRAN. And these general-purpose programming languages are known as **host languages**. Various techniques have been developed for accessing databases from other programs. Some of the techniques for using SQL statements in host language are embedded SQL, cursors, dynamic SQL, ODBC, JDBC, and SQLJ. These techniques are discussed in this section.

5.6.1 Embedded SQL

For accessing a database from any application program, the SQL statements are embedded within an application program written in host language. The use of SQL statements within a host language program is known as **embedded SQL**. These statements are also known as **static**, that is, they do not change at runtime. SQL statements included in a host language program must be marked so that the pre-processor can handle them before invoking the compiler for a host language. The embedded SQL statement is prefixed with the keywords `EXEC SQL` to distinguish it from the other statements of host language. Prior to compilation, an embedded SQL program is processed by a special pre-processor or pre-compiler, which identifies the SQL statements during program scan, extracts and pass it to the DBMS for processing. The end of SQL statements is identified by encountering a semicolon (`;`) or a matching `END-EXEC`. For the discussion of embedded SQL, consider C as a host language.

Variables of host language can be referred in SQL statements. Such variables are also known as **host variables** and are prefixed by a colon (`:`) when they appear in SQL statements and are declared between the

commands `EXEC SQL BEGIN DECLARE SECTION` and `EXEC SQL END DECLARE SECTION`. The declaration of these variables is similar to the declaration of variables in C language and is separated by semicolons. These variables are also known as **shared variables** as they are shared by both C language and SQL statements. For example, the declaration of variables corresponding to the attributes of relation `BOOK` will be like as shown here.

```
EXEC SQL BEGIN DECLARE SECTION;
varchar c_ISBN[15], c_book_title[50], c_category[20];
char c_p_id[4];
int c_copyright_date[10], c_year, c_page_count;
float c_price;
int SQLCODE;
char SQLSTATE[6];
EXEC SQL END DECLARE SECTION;
```

Program variables may or may not have same names as that of attributes in a corresponding relation. The SQL data type `NUMERIC` can be mapped to C data type `int` or `float` (if includes decimal portion). The SQL fixed length and variable length strings can be mapped to arrays of characters of type `char` and `varchar`, respectively. The variables `SQLCODE` and `SQLSTATE` are used to communicate errors and exception conditions between the database system and the host language program. The `SQLCODE` is an integer variable, which returns a positive value (like `SQLCODE=100`) when there is no more data in the resultant relation. The `SQLCODE` returns negative value when an error occurs. `SQLSTATE` is a five character code the 6th character is for *null* character in C language. The value *00000* of the variable `SQLSTATE` indicates error free condition. It associates predefined values with several common error conditions.

SQL statements can appear anywhere in the host language program, where other statements of host language can appear. Before executing any SQL statements, the host program must establish a connection with the database as shown here.

```
EXEC SQL CONNECT TO <server_name> AS <connection_name>
AUTHORIZATION <user_name and password>;
```

In this command, `server_name` identifies the server to which a connection is to be established and the `connection_name` is the name provided to the connection. In addition, `user_name` and `password` identifies the authorized user. While programming, more than one connection can be established with only one connection active at a time. When a connection is no longer needed, it can be ended by using the command as shown here.

```
EXEC SQL DISCONNECT <connection_name>;
```

Note that the commands are terminated by semicolon. Assuming that the required connection is already established, the command to insert a tuple in relation using host variables can be specified as

```
EXEC SQL

INSERT INTO BOOK

VALUES (:c_ISBN, :c_book_title, :c_category, :c_price,
:c_copyright_date, :c_year, :c_page_count, :c_p_id);
```

Consider another example to retrieve the details of book with `ISBN` value stored in the variable `c_ISBN`, the embedded SQL statement can be specified as

```
EXEC SQL
SELECT Book_title, Category, Price, Year, Page_count
INTO :c_book_title, :c_category, :c_price, :c_year, :c_page_
FROM BOOK
WHERE ISBN=:c_ISBN;
```

In this command, the details of book satisfying the condition is stored in host variables. As a result, only single tuple is retrieved and hence, can

be stored in the host variables. However, while programming, more than one tuple satisfying the condition can also be retrieved. To deal with such a situation, the concept of cursors can be used, which is discussed in the following section.

5.6.2 Cursors

Cursor is a mechanism that provides a way to retrieve multiple tuples from a relation and then process each tuple individually in a host program. The cursor is first opened, then processed, and then closed. Cursor can be declared on any relation or any SQL statement that returns a set of tuples. Once the cursor is declared, it can be opened, moved to n^{th} tuple and closed. When the cursor is opened, it fetches the query result from the database and it is positioned just before the first tuple. In the query result, any individual tuple of a relation can be fetched by pointing cursor to the desired tuple. Sometimes the cursor is opened implicitly (automatically) by the RDBMS especially for the queries returning single tuple. However, for the queries returning more than one tuple, explicit cursors (defined by user) are declared.

In general, to use the explicit cursor, the following steps are performed.

1. **Declare the cursor:** The cursor is declared using the `DECLARE` command.
2. **Open the cursor:** Before using the cursor, it must be opened after it has been declared.
3. **Fetch tuple from the cursor:** After opening the cursor, the tuples associated with the cursor can be processed individually with the help of `FETCH` command.
4. **Close the cursor:** When cursor is not required, it must be closed by using the `CLOSE` command.

The declaration of cursor takes the following form.

```
DECLARE <cursor_name> [INSENSITIVE][SCROLL] CURSOR
[WITH HOLD] FOR <query>
```

```
[ORDER BY <attribute_list>]
[FOR READ ONLY | FOR UPDATE [OF <attribute_list>]];
```

The clause FOR UPDATE OF is added in the declaration of cursor, indicating that the tuples associated with cursor will be retrieved for updating and the attribute to be updated is specified with it. The ORDER BY clause is used to order the tuples in the resultant relation. A cursor can be declared as read only cursor using the clause FOR READ ONLY, indicating the tuples retrieved as a result of SQL query cannot be modified. Other keywords that can be used in the declaration of cursor are SCROLL, INSENSITIVE, and WITH HOLD. When the keyword SCROLL is specified, then that cursor is scrollable, this means that the FETCH command can be used to position the cursor in a flexible manner instead of default sequential access. The other two keywords INSENSITIVE and WITH HOLD are used to refer the transaction characteristics of database programs.

For example, the program segment to increment the price of books belonging to a given category from relation BOOK by the value entered by the user can be written as shown in Figure 5.1.

When the cursor is opened, it is positioned at the beginning of the first tuple. When FETCH command is executed, the attribute values in the corresponding tuple are read and stored in respective host variables in the specified order. To read consecutive tuples, the FETCH command is executed repeatedly. If the execution of FETCH command results in the moving of cursor to the end of last tuple, a positive value (SQLCODE>0) is returned in SQLCODE. Positive value in SQLCODE indicates presence of no more data in the resultant relation. This state of SQLCODE is used to terminate the loop. The WHERE CURRENT OF Cur1 clause in the embedded UPDATE command is used to specify that the tuple referred currently by the cursor is the tuple to be updated. After the complete processing of resultant relation, the cursor is closed using CLOSE command.

Things to Remember

Once the cursor is opened it cannot be reopened. That is, cursor must be closed before reopening it. Also, the cursor always moves forward and not backwards.

5.6.3 Dynamic SQL

Embedded SQL allows the integration of SQL statements within the host language program. Such statements are of static nature, that is, once these statements are written in the program then they cannot be modified at any time. Hence, to specify a new query, new program must be written to accommodate the new query. Dynamic SQL, unlike embedded SQL statements, are generated at the run-time and it is placed as a string in the host variable. The SQL statements created like this are passed to the DBMS for further processing. Dynamic SQL is slower than embedded SQL as it involves time to generate a query also during the runtime. However, it is more powerful than embedded SQL, as queries can be generated at runtime as per varying user requirements.

```
printf("Enter the category of book : ");
scanf("%s", c_category);
EXEC SQL DECLARE Cur1 CURSOR FOR
SELECT Book_title, Price, Year
FROM BOOK
WHERE Category=:c_category
ORDER BY Book_title
FOR UPDATE OF Price;
EXEC SQL OPEN Cur1;
EXEC SQL FETCH FROM Cur1 INTO :c_Book_title, :c_Price
WHILE(SQLCODE ==0)
{
 printf("Enter the increment amount : ");
 scanf("%f", &inc);
 EXEC SQL
       UPDATE BOOK
       SET Price = Price + :inc
       WHERE CURRENT OF Cur1;
 EXEC SQL FETCH FROM Cur1 INTO :c_Book_title, :c_Pric
}
```

```
EXEC SQL CLOSE Cur1;
```

Fig. 5.1 *Retrieving tuples using cursor*

For example, consider program segment given in Figure 5.2. This sample program allows user to enter the update SQL query to be executed.

```
EXEC SQL BEGIN DECLARE SECTION;
       char sqlquerystring[200];
EXEC SQL END DECLARE SECTION;
printf("Enter the required update query : \n");
scanf("%s", sqlquerystring);
EXEC SQL PREPARE sqlcommand FROM :sqlquerystring;
EXEC SQL EXECUTE sqlcommand;
```

Fig. 5.2 *Program segment using dynamic SQL*

In this code segment, first the string `sqlquerstring` is declared, which is used to hold the SQL query statement entered by the user. After prompting the required query from the user, it is converted into corresponding SQL command by using the `PREPARE` command. This query is executed by using the `EXECUTE` command. The query entered by the user can be in the form of a single string or it can be created by using concatenation of strings.

5.6.4 Open Database Connectivity (ODBC)

To access the database from general-purpose programming language, the application programs needs to set up a connection with the database server. A standard called **Open Database Connectivity (ODBC)** provides an **application programming interface (API)** that the application programs can use to establish a connection with the database. Once the connection is established, the application program can communicate with the database through queries. Most DBMS vendors provide ODBC drivers for their systems.

An application program from the client-site can access several DBMSs by making ODBC API call. The requests from the client program are then processed at the server-sites and the results are sent back to the client program. Consider an example of C code using ODBC API given in Figure 5.3.

```
void Example()
{
        HENV En1;      //environment
        HDBC Cn;   //to establish connection
        RETCODE Err;
        SQLAllocEnv(&En1);
 SQLAllocConnect(En1,&Cn); SQLConnect(Cn, "db.onlinebook.edu
        int c_Price1, c_Price2;
        int L1, L2;
        HSTMT st;
 char *Query = "SELECT MAX(Price), MIN(Price) FROM BOOK
   GROUP BY Category";
        SQLAllocStmt(Cn, &st);
        Err = SQLExecDirect(st, Query, SQL_NTS);
        if (Err == SQL_SUCCESS)
        {
                SQLBindCol(st, 1, SQL_C_INT, &c_Price1, 0, &L
                SQLBindCol(st, 2, SQL_C_INT, &c_Price2, 0, &L
                while(SQLFetch(st) == SQL_SUCCESS)
                {
                        printf("Maximum Price is %d", c_Price
                        printf("Minimum Price is %d", c_Price
                }
        }
        SQLFreeStmt(st, SQL_DROP);
        SQLDisconnect(Cn);
        SQLFreeConnect(Cn);
        SQLFreeEnv(En1);
}
```

Fig. 5.3 *An example of ODBC code*

In the beginning of the program, the variables `En1`, `Cn`, and `Err` of types `HENV`, `HDBC`, and `RETCODE`, respectively are declared. The variable `En1` and `Cn` are used to allocate an SQL environment and database connection handle, respectively. The variable `Err` is used for error detection. Further, the program establishes a connection with the database by using `SQLConnect()` function, which accepts parameters, database connection handle (`Cn`), server (`db.onlinebook.edu`), user identifier (`John`), and the password (`Passwd`). Notice the use of constant `SQL_NTS`, which denotes that the previous argument is a null-terminated string.

After establishing a connection, the program communicates with the database by sending a query to `SQLExecDirect()` function. The attributes of the query result are bounded to corresponding C variables by using `SQLBindCol()` function. The parameters passed to the `SQLBindCol()` are:

- **First parameter (`st`):** stores the result of the query
- **Second parameter (`1` or `2`):** determines the location of an attribute in the result of a query
- **Third parameter (`SQL_C_INT`):** specifies the required data type conversion of an attribute from SQL to C.
- **Fourth parameter (`&c_Price1` or `&c_Price2`):** specifies the address of the C variable where the attribute value is to be stored. Note that the values of last two parameters passed to `SQLBindCol()` function, depends on the data type of an attribute. For example, in case of fixed-length types, such as float or integer the fifth parameter is ignored and negative value in last parameter indicates *null* value in an attribute.

When the resultant tuple is fetched using `SQLFetch()` function, the attribute values of the query are stored in corresponding C variables. The `SQLFetch()` function executes the statement `st` as long as it returns the value `SQL_SUCCESS`. In each iteration of `while` loop, the attribute values for each category are stored in corresponding C variables and are displayed through `printf` statements. The connection must be closed when it is no

more required, using `SQLDisconnect()` function. Also, all the resources that are allocated must be freed.

5.6.5 Java Database Connectivity (JDBC)

Accessing a database in Java requires Java Database Connectivity (JDBC). JDBC provides a standard API that is used to access databases, through Java, regardless of the DBMS. All the direct interactions with specific DBMSs are accomplished by DBMS specific driver. The four main components required for the implementation of JDBC are: application, driver manager, data source specific drivers, and corresponding data sources.

An application establishes and terminates the connection with a data source. The main goal of driver manager is to load JDBC drivers and pass JDBC function calls from the application to the corresponding driver. The driver establishes the connection with data source. The driver performs various basic functions like submitting requests and returning results. In addition, the driver translates data, error formats, and error codes from a form that is specific to data source into the JDBC standard. The data source processes commands from the driver and returns the results.

For understanding the connectivity of Java program with JDBC, consider sample program segment given in Figure 5.4.

In this program segment, following steps are taken when writing a Java application program accessing database through JDBC function calls.

1. Appropriate drivers for the database are loaded by using `Class.forName`.
2. The `getConnection()` function of DriverManager class of JDBC is used to create the connection object. The first parameter (`URL`) specifies the machine name (`db.onlinebook.edu`)

```
public static void Sample(String DB_id, String U_id
{
  String URL = "jdbc:oracle:oci8:@db.onlinebook.edu
```

```
  Class.forName("oracle.jdbc.driver.OracleDriver");
  Connection Cn=DriverManager.getConnection(URL,U_i
  Statement st = Cn.createStatement();
  try
  {
      st.executeUpdate("INSERT INTO AUTHOR VALUES('
     Jones', 'Texas')");
  }
  catch(SQLException se)
  {
      System.out.println("Tuple cannot be inserted.
  }
  ResultSet rs = st.executeQuery("SELECT Aname, Sta
  WHERE City = 'Seatle'");
  while(rs.next())
  {
System.out.println(rs.getString(1) + " "+ rs.getStr
  }
  st.close();
  Cn.close();
}
```

Fig. 5.4 *An example of JDBC code*

where the server runs, the port number (`100`) used for communication, schema (`onbook_db`) to be used and the protocol (`jdbc:oracle:oci8`) used to communicate with the database.

To connect to the database, username and password are also required which are specified by the strings `U_id` and `Pword`, respectively, the other two parameters of `getConnection()` function.

3. A statement handle is created for the connection established which is used to execute an SQL statement. In this example, `st.executeUpdate` is used to execute an `INSERT` statement of SQL. The `try {..}` catch`{..}` is used to catch any exceptions that may arise as a result of executing this query statement.
4. Another query is executed using the statement `st.executeQuery`.

The result of this may consist of set of tuples, which is assigned to `rs` of type `ResultSet`.

5. The `next()` function is used to fetch one tuple at a time from the result set `rs`. The value of attributes of a tuple is retrieved by using the position of the attribute. The attribute `Aname` is at first (1) position and `State` is at second (2) position. The values for the attributes can also be retrieved by using its name as shown here.

   ```
   System.out.println(rs.getString("Aname")+
   " " +rs.getString("State"));
   ```

6. The connection must be closed after it is no more required at the end of procedure.

The special type of statement also known as **prepared statement,** can be used to specify SQL query, in which unknown values are replaced by '`?`'. The user provides the unknown values later at the run-time. Some database systems compile the query when it is prepared and whenever the query is executed with new values, database uses this compiled form of this query. The `setString()` function is used to specify the values for the parameters. Consider the following statements, to understand the concept of prepared statement.

```
PreparedStatement ps = Cn.prepareStatement("INSERT INTO BOOK
ps.setString(1, "A010");   //assigns value to first attribut
ps.setString(2, "Smith Jones"); //assigns value to second
                //attribute
ps.setString(3, "Texas"); //assigns value to third attribute
ps.executeUpdate();
```

The values can be assigned to the corresponding attributes through variables also instead of using literal values. Prepared statements are preferred in the situations when query uses the values provided by the user. JDBC also provides a `CallableStatement` interface, which can be used for invoking SQL stored procedures and functions.

`CallableStatement` is a subclass of `PreparedStatement`. For example, the command to call the procedure, say `Number_Of_Books`, can be specified as

```
CallableStatement cs1 = Cn.prepareCall("{call Number_Of_Book
ResultSet rs = cs1.executeQuery();
```

A stored procedure may contain multiple SQL statements or a series of SQL statements, thus, resulting into many different `ResultSet` objects. In this example, it is assumed that there is only single `ResultSet`.

5.6.6 SQL-Java (SQLJ)

SQLJ is a standard that has been used for embedding SQL statements in Java programming language, which is an object-oriented language like Java. In SQLJ, a pre-processor called "SQLJ translator" converts SQL statements into Java, which can be executed through JDBC interface. Hence, it is essential to install a JDBC driver while using SQLJ. When writing SQLJ applications, regular Java code is written and SQL statements are embedded into it using a set of rules. SQLJ applications are pre-processed using SQLJ translation program that replaces the embedded SQLJ code with calls to an SQLJ library. Any Java compiler can then compile this modified program code. The SQLJ library calls the corresponding driver, which establishes the connection with the database system.

The use of SQLJ improves the productivity and manageability of JAVA code as code becomes compact and no runtime syntax errors occur as SQL statements are checked at compile time. Moreover, it allows sharing of Java variables with SQL statements.

For understanding the coding in SQLJ, consider a sample SQLJ program segment given in Figure 5.5. This program retrieves the book details belonging to a given category.

```
String Book_title; float Price; String Category;
#sql iterator Bk(String Book_title, float Price);
Bk book = {SELECT Book_title, Price INTO :Book_title, :
BOOK WHERE Category = :Category};

while(book.next())
{
   System.out.println(book.Book_title() + “, ” + book.P
}
book.close();
```

Fig. 5.5 *An example of SQLJ code*

SQLJ statements always begin with the `#sql` keyword. The result of SQL queries is retrieved through the objects of **iterator**, which are basically a type of cursor. The iterator is associated with the tuples and attributes appearing in the query result. An iterator is declared as an instance of iterator class. The usage of an iterator in SQLJ basically goes through following four steps.

1. **Declaration of iterator class:** In the Figure 5.5, the iterator class `Bk` is declared by using the statement.

   ```
   #sql iterator Bk(String Book_title, float Price);
   ```

2. **Creation and initialization of iterator object:** An object of iterator class is created and initialized with a set of tuples returned as a result of SQL query statement.
3. **Accessing the tuples with the help of iterator object:** Iterator is set to the position just before the first row in the result of query, which becomes the current tuple for the iterator. The `next()` function is then applied on the iterator repeatedly to retrieve the subsequent tuples from the result.
4. **Closing the iterator object:** An object of iterator is closed after all the tuples associated with the result of SQL query are processed.

There are two types of iterator classes, namely, *named* and *positional* iterators. In case of **named iterators** both the variable types and the name of each column of the iterator is specified. This helps in retrieving the individual columns by their name. Like, in the Figure 5.5, the sample program to retrieve `Book_title` from the iterator book, the expression `book.Book_title()` is used. While, in case of **positional iterators**, only the variable type for each column of iterator is specified. The individual columns of the iterator are accessed using the `FETCH. .INTO` statement like embedded SQL. Both types of iterators have same performance and can be used according to the user requirement.

NOTE SQLJ is much easier to read than JDBC code. Thus, SQLJ reduces software development and maintenance cost.

1. The structured query language (SQL) pronounced as "ess-que-el", is a language which can be used for retrieval and management of data stored in relational database. It is a non-procedural language as it specifies what is to be retrieved rather than how to retrieve it.
2. Some common relational database management systems that use SQL are: Oracle, Sybase, Microsoft SQL Server, Access, Ingres, etc.
3. SQL provides different types of commands, which can be used for different purposes. These commands are divided into two major categories, namely, data definition language (DDL) and data manipulation language (DML).
4. Data definition language (DDL) commands are used for defining relation schemas, deleting relations, creating indexes, and modifying relation schemas.
5. Present day database systems provide a three level hierarchy for naming different objects of relational database. This hierarchy comprises catalogs, which in turn consists of schemas, and various database objects are incorporated within the schema.
6. SQL supports various data types such as numeric, integer, character, character varying, date and time, Boolean, and timestamp.
7. In addition to built-in data types, user-defined data types can also

be created. The user-defined data type can be created using the `CREATE DOMAIN` command.

8. The `CREATE TABLE` command is used to define a new relation, its attributes, and its data types. In addition, various constraints such as key, integrity, and referential constraints along with the restrictions on attribute domains and null values can also be specified within the definition of a relation.
9. `ALTER TABLE` command is used to change the structure of a relation. The user can add, modify, delete, or rename an attribute. The user can also add or delete a constraint.
10. The `DROP TABLE` command is used to remove an already existing relation, which is no more required as a part of a database.
11. DML commands are used for retrieving and manipulating data stored in the database. The basic retrieval and manipulating commands are `SELECT`, `INSERT`, `UPDATE`, and `DELETE`.
12. The criteria to determine the tuples to be retrieved is specified by using the `WHERE` clause.
13. A tuple variable is associated with a particular relation and is defined in the `FROM` clause.
14. SQL provides `LIKE` operator, which allows to specify comparison conditions on only parts of the strings. Different patterns are specified by using two special wildcard character, namely, per cent (`%`) and underscore (_).
15. There are various set operations that can be performed on relations, like, union, intersection, and difference. In SQL, these operations are implemented using `UNION`, `INTERSECT`, and `MINUS` operations, respectively.
16. SQL provides the `ORDER BY` clause to arrange the tuples of relation in some particular order. The tuples can be arranged on the basis of the values of one or more attributes.
17. The `INSERT` command is used to add a single tuple at a time in a relation.
18. The `UPDATE` command is used to make changes in the values of the

attributes of the relations. The DELETE command is used to remove the tuples from the relation, which are no more required as a part of a relation. Tuples can be deleted from only one relation at a time.

19. SQL provides five built-in aggregate functions, namely, SUM, AVG, MIN, MAX, and COUNT.
20. The GROUP BY clause can be used in SELECT command to divide the relation into groups on the basis of values of one or more attributes. Conditions can be placed on the groups using HAVING clause.
21. A query that combines the tuples from two or more relations is known as join query. In such type of queries, more than one relation is listed in the FROM clause.
22. The query defined in the WHERE clause of another query is known as nested query or subquery. The query in which another query is nested is known as enclosing query.
23. Some of the advanced features of SQL that help in specifying complex constraints and queries efficiently are assertions and views.
24. Assertion is a condition that must always be satisfied by the database.
25. View is a virtual relation, whose contents are derived from already existing relations and it does not exist in physical form. Certain database systems allow views to be physically stored, also known as materialized views.
26. The two standard features of SQL that are related to programming in database are stored procedures and triggers.
27. Stored procedures are procedures or functions that are stored and executed by the DBMS at the database server machine.
28. A trigger is a type of stored procedure that is executed automatically when some database related events like, insert, update, delete, etc., occur.
29. Various techniques have been developed for accessing databases from other programs. Some of the techniques for using SQL statements in host language are embedded SQL, cursors, dynamic SQL, ODBC, JDBC, and SQLJ.
30. The use of SQL statements within a host language is known as

embedded SQL. The embedded SQL statement is prefixed with the keywords `EXEC SQL` to distinguish it from the other statements of host language.

31. Cursor is a mechanism that provides a way to retrieve multiple tuples from a relation and then process each tuple individually in a host program. The cursor is first opened, then processed, and then closed.
32. Dynamic SQL, unlike embedded SQL statements, are generated at the run-time and it is placed as a string in the host variable.
33. A standard called Open Database Connectivity (ODBC) provides an application programming interface (API) that the application programs can use to establish a connection with the database.
34. Accessing a database in Java requires Java Database Connectivity (JDBC). JDBC provides a standard API that is used to access databases, through Java, regardless of the DBMS.
35. SQLJ is a standard that has been used for embedding SQL statements in Java programming language, which is object-oriented language like Java. In SQLJ, a pre-processor called SQLJ translator converts SQL statements into Java, which can be executed through JDBC interface.

KEY TERMS

- Data definition language (DDL)
- Data manipulation language (DML)
- `CREATE SCHEMA`
- `VARCHAR`
- `NUMERIC`
- `CREATE DOMAIN`
- `CREATE TABLE`
- `DESCRIBE`
- `PRIMARY KEY` constraint
- `UNIQUE` constraint
- `CHECK` constraint

- `CONSTRAINT`
- `NOT NULL` constraint
- `FOREIGN KEY` constraint
- `ALTER TABLE`
- `ADD`
- `MODIFY`
- `DROP COLUMN`
- `DROP CONSTRAINT`
- `RENAME COLUMN`
- `DROP TABLE`
- `SELECT`
- `DISTINCT` keyword
- `WHERE` clause
- `BETWEEN` operator
- `IN` operator
- `AS` clause
- `LIKE` operator
- `UNION` operation
- `INTERSECT` operation
- `MINUS` operation
- `ORDER BY` clause
- `INSERT`
- `UPDATE`
- `DELETE`
- `IS NULL`
- `AVG`
- `MIN`
- `MAX`
- `SUM`
- `COUNT`
- `GROUP BY` clause
- `HAVING` clause

- Join query
- Equi-join query
- Natural join query
- Nested query
- `ANY` operator
- `ALL` operator
- `EXISTS` operator
- Assertion
- `CREATE ASSERTION`
- View
- `CREATE VIEW`
- Materialized views
- Stored procedures
- `CREATE PROCEDURE`
- `CREATE FUNCTION`
- SQL/PSM
- Trigger
- `CREATE TRIGGER`
- Row level trigger
- Statement level trigger
- `REFERENCING NEW ROW AS` clause
- `REFERENCING OLD ROW AS` clause
- `ALTER TRIGGER`
- `DROP TRIGGER`
- Embedded SQL
- Cursor
- Dynamic SQL
- Open Database Connectivity (ODBC)
- Application programming interface (API)
- Java Database Connectivity (JDBC)
- Prepared statement
- SQL-Java (SQLJ)
- Iterator
- Named iterators

- Positional iterators

EXERCISES

A. Multiple Choice Questions

1. DML provides commands that can be used to
    1. Create
    2. Modify
    3. Delete database objects
    4. Delete data from a database
2. Which of these constraints ensures that the foreign key value in the referencing relation must exist in the primary key attribute of the referenced relation?
    1. Primary key
    2. Reference Key
    3. Foreign key
    4. Unique Key
3. Which of these keywords is used to eliminate duplicate tuples from the resultant relation?
    1. `DELETE`
    2. `DISTINCT`
    3. `EXISTS`
    4. `NON DUPLICATE`
4. Which of these operators selects values that match any value in a given list of values?
    1. `BETWEEN`
    2. `LIKE`
    3. `IN`
    4. None of these
5. Which of these characters is used to match substring with any number of characters?
    1. `%`
    2. `*`
    3. `?`

    4. _
6. Which of these operations is used to retrieve those tuples present in one relation, and not present in other relation?
    1. `UNION`
    2. `INTERSECT`
    3. `MINUS`
    4. `DIFFERENCE`
7. Which of these commands can be used to make changes in the name of publishers in a relation `PUBLISHER`?
    1. `UPDATE`
    2. `ALTER`
    3. `MODIFY`
    4. None of these
8. Which of the following is true statement?
    1. If the value of `A` is *true* or *unknown*, the result is *unknown*
    2. If the value of `A` is *false*, the result is *false*
    3. If the value of `A` is *false* or *unknown*, the result is *unknown*
    4. If the value of `A` is *true*, the result is *unknown*
9. Which of these clauses is used to restrict groups returned by the `Group By` clause?
    1. `DISTINCT`
    2. `WHERE`
    3. `EXISTS`
    4. `HAVING`
10. Which of these mechanisms provides a way to retrieve multiple tuples from a relation and then process each tuple individually in a host program?
    1. Assertions
    2. Cursors
    3. Triggers
    4. None of these

B. Fill in the Blanks

1. The _____ command is used to define a new relation, its attributes and its data types.
2. The _____ constraint ensures that the set of attributes have unique values, that is, no two tuples can have same value in the specified attributes.
3. The _____ clause, prevents a relation to be dropped if it is referenced by any of the constraints or views.
4. A query that combines the tuples from two or more relations is known as _____.
5. _____ operator evaluates to *true* if a subquery returns at least one tuple as a result otherwise, it returns *false* value.
6. _____ are type of stored procedures which are automatically invoked and are needed to perform complex data verification operations.
7. Certain database systems allow views to be physically stored, which are known as _____.
8. _____, unlike embedded SQL statements, are generated at the run-time and it is placed as a string in the host variable.
9. _____ provides a standard API that is used to access databases, through Java, regardless of the DBMS.
10. In case of SQLJ, the result of SQL queries is retrieved through the objects of _____, which are basically a type of cursor.

C. Answer the Questions

1. What is SQL? What are the two major categories of SQL commands? Explain them.
2. What type of information can be specified about a relation in DDL?
3. Discuss the concept of schema and catalog in SQL.
4. Discuss common data types supported by standard SQL.
5. How user-defined data types can be created in SQL? Give example.
6. Give purpose and syntax of `CREATE TABLE` and `DESCRIBE` command.
7. Discuss different types of constraints along with syntax that can be specified while creating a relation in SQL.
8. Give purpose, syntax, and example of the following commands along

with their various clauses:

    1. `ALTER TABLE`
    2. `DROP TABLE`
    3. `SELECT`
    4. `INSERT`
    5. `UPDATE`
    6. `DELETE`

9. Which operator of SQL is used to specify string patterns in the queries? Explain in detail with examples.
10. Discuss how set operations can be implemented in SQL.
11. Which clause of `SELECT` command can be used to change the order of tuples in a relation? Explain with syntax and example.
12. How *null* values are represented and handled in SQL queries?
13. What is the role of aggregate functions in SQL queries?
14. Explain how the `GROUP BY` clause works. What is the difference between `WHERE` and `HAVING` clauses? Explain them with the help of an example.
15. How queries based on joins are expressed in SQL?
16. What do you understand by nested queries? Explain with example.
17. What are assertions? What is the utility of assertions?
18. What is view in SQL? How are assertions different from views? Compare assertions and views with the help of example.
19. What are the conditions under which views can be updated?
20. What are stored procedures? When are they beneficial? Give syntax and example of creating a procedure and function in SQL.
21. Discuss about triggers. How they are created? How do triggers offer a powerful mechanism for dealing with the changes to database? Explain with example.
22. What do you mean by embedded SQL? How variables are declared in embedded SQL? Explain with examples.
23. What problem is faced while using embedded SQL and how it is addressed by cursors? Explain with example.
24. What is dynamic SQL and how is it different from embedded SQL?
25. What is the purpose of ODBC in SQL? How it is implemented?

Explain with example.

26. What is the purpose of JDBC in SQL? How it is implemented? Explain with example.
27. Differentiate between ODBC and JDBC.
28. What is SQLJ? What are the requirements of SQLJ? Briefly describe the working of SQLJ.
29. Give steps involved in the implementation of iterators in SQLJ. Discuss two types of iterators available in SQLJ.

D. Practical Questions

1. Consider the relation schemas given in question 1–4 (Practical questions) of Chapter 04. Write appropriate DDL commands to define these relation schemas along with the required constraints defined. Use `INSERT` command to add five tuples in each relation.
2. Express queries given in questions 1–4 (Practical questions) of Chapter 04 in SQL using various DML commands.
3. Consider the following relational scheme:

   ```
   BOOKS(Book_Id, B_name, Author, Purchase_date, Cost)
   MEMBERS(Member_Id, M_name, Address, Phone, Birth_date)
   ISSUE_RETURN(Book_Id, Member_Id, Issue_date, Return_date)
   ```

   Specify the following queries in SQL.

   1. Find the names of all those books that have not been issued.
   2. Display the `Member_Id` and the number of books issued to that member. (Assume that it a book in `ISSUE_RETURN` relation does not have a `Return_date`, then it is issued.)
   3. Find the book that has been issued the maximum number of times.
   4. Display the names and authors of books that have been issued at any time to a member `Member_Id` is *ab*.
   5. List the `Book_Id` of those books that have been issued to any member whose date of birth is less than *01-01-1985*, but have

not been issued to any member having the birth date equal to or greater than *01-01-1985*.

4. Specify the following queries on *Online Book* database in SQL.
    1. In BOOK relation, increase the price of all the books belonging to novel category by 10%.
    2. In BOOK relation, increase the price of all the books published by *Hills Publication* by 5%.
    3. In PUBLISHER relation, change the phone of *Wesley Publications* to *9256774*.
    4. In REVIEW relation, increase the rating of all the books written by the authors *A003* by one.
    5. Calculate and display average, maximum, and minimum price of book from each category.
    6. Calculate and display average, maximum, and minimum price of book from each publisher.
    7. Delete details of all the books having page count less than *100*.
    8. Retrieve details of all the books whose name begins with character *D*.
    9. Retrieve details of all the publishers located in state *Georgia*.
    10. Retrieve details of all the authors residing in the city *New York* and whose name begins with character *J*.
    11. Retrieve book title, name of publisher, and author name of all language books.
    12. Retrieve book details having price equals to average price of all the books.
    13. Retrieve author details residing in the same city as *Lewis Ian*.
5. Give commands to create views based on the following queries of *Online Book* database and state whether they are updateable.
    1. A view containing details of all the books belonging to textbook category.
    2. A view containing details of all the books published by the publisher *Bright Publications* and price greater than *$30*.
    3. A view containing details of all the textbooks written by author *Charles Smith*.

    4. A view containing details of all the books having page count *600* and published in the year *2004*.
6. Create a procedure to update the value of page count of a book with a given `ISBN`.
7. Create a procedure to update the value of phone of a publisher with a given publisher ID.
8. Create a function that returns the price of a book with a given `ISBN`.
9. Create a function that searches a book with a given `ISBN` and returns the value *Old* if it is published before year *2000* and *New* if it is published in or after year *2000*.
10. Create a trigger for changing the value of `Page_count` attribute to a default value *100*, if the value entered by the user exceeds the upper limit of *1500* in case of insertion in relation `BOOK`.
11. Specify embedded SQL statement to retrieve the detail of author with `A_ID` value stored in the variable `c_AID`.
12. Write a program segment which uses cursor to update the page count of books belonging to a given publisher ID from relation `BOOK` by the value entered by the user.
13. Write an ODBC code for displaying publisher along with maximum price of books published by them from `BOOK` relation.
14. Write JDBC code for displaying title, price, and page count of books belonging to language book category.
15. Write SQLJ code for retrieving the book details belonging to a given publisher.
16. Write SQLJ code for retrieving the publisher details publishing novels.