CHAPTER 1

INTRODUCTION TO RDBMS

1.0 Objectives
1.1 Introduction
1.2 What is RDBMS ?
1.3 Difference between DBMS & RDBMS
1.4 Summary
1.5 Check your Progress – *Answers*
1.6 Questions for Self – Study
1.7 Suggested Readings

1.0 OBJECTIVES

After reading this chapter you will be able to,

- Describe what RDBMS is
- State the difference between DBMS & RDBMS

1.1 INTRODUCTION

Most of the problems faced at the time of implementation of any system are outcome of a poor database design. In many cases it happens that system has to be continuously modified in multiple respects due to changing requirements of users. It is very important that a proper planning has to be done.
A relation in a relational database is based on a relational schema, which consists of number of attributes.
A relational database is made up of a number of relations and corresponding relational database schema.
The goal of a relational database design is to generate a set of relation schema that allows us to store information without unnecessary redundancy and also to retrieve information easily.
One approach to design schemas that are in an appropriate normal form. The normal forms are used to ensure that various types of anomalies and inconsistencies are not introduced into the database.

1.2 WHAT IS RDBMS?

RDBMS stands for Relational Database Management System. RDBMS data is structured in database tables, fields and records. Each RDBMS table consists of database table rows. Each database table row consists of one or more database table fields.
RDBMS store the data into collection of tables, which might be related by common fields (database table columns). RDBMS also provide relational operators to manipulate the data stored into the database tables. Most RDBMS use SQL as database querylanguage.
The most popular RDBMS are MS SQL Server, DB2, Oracle and MySQL.
The relational model is an example of record-based model. Record based models are so named because the database is structured in fixed format records of several types. Each table contains records of a particular type. Each record type defines a fixed number of fields, or attributes. The columns of the table correspond to the attributes of the record types. The relational data model is the most widely used data model, and a vast majority of current database systems are based on the relational model.
The relational model was designed by the IBM research scientist and mathematician, Dr. E.F.Codd. Many modern DBMS do not conform to the Codd's definition of a RDBMS, but nonetheless they are still considered to be RDBMS.
Two of Dr.Codd's main focal points when designing the relational model were to further reduce data redundancy and to improve data integrity within database systems.

The relational model originated from a paper authored by Dr.codd entitled “A Relational Model of Data for Large Shared Data Banks”, written in 1970. This paper included the following concepts that apply to database management systems for relational databases.
The relation is the only data structure used in the relational data model to represent both entities and relationships between them.
Rows of the relation are referred to as **tuples** of the relation and columns are its **attributes**. Each attribute of the column are drawn from the set of values known as **domain**. The domain of an attribute contains the set of values that the attribute may assume.
From the historical perspective, the relational data model is relatively new .The first database systems were based on either network or hierarchical models .The relational data model has established itself as the primary data model for commercial data processing applications. Its success in this domain has led to its applications outside data processing in systems for computer aided design and other environments.

1.3 DIFFERENCE BETWEEN DBMS & RDBMS

A DBMS has to be persistent, that is it should be accessible when the program created the data ceases to exist or even the application that created the data restarted. A DBMS also has to provide some uniform methods independent of a specific application for accessing the information that is stored.
RDBMS is a Relational Data Base Management System Relational DBMS. This adds the additional condition that the system supports a tabular structure for the data, with enforced relationships between the tables. This excludes the databases that don't support a tabular structure or don't enforce relationships between tables.
You can say DBMS does not impose any constraints or security with regard to data manipulation it is user or the programmer responsibility to ensure the ACID PROPERTY of the database whereas the RDBMS is more with this regard because RDBMS define the integrity constraint for the purpose of holding ACID PROPERTY.

1.1,1.2, and 1.3 Check your progress

Fill in the blanks

1) A relation in a relational database is based on a relational schema, which consists of number of ..... .
2) .....is a Relational Data Base Management System.
3) Rows of the relation are referred to as ..... of the relation
4) The relational model was designed by the IBM research scientist and mathematician, Dr. .....
5) The ..... is the only data structure used in the relational data model to represent both entities and relationships between them.

State true or false

1) The normal forms never removes anomalies.
2) Each attribute of the column are drawn from the set of values known as domain.
3) The first database systems were based on either network or hierarchical models .
4) Most RDBMS use SQL as database query language.
5) Relational database design makes data retrieval difficult.

1.4 SUMMARY

The goal of a relational database design is to generate a set of relation schema that allows us to store information without unnecessary redundancy and also to retrieve information easily.
A database system is an integrated collection of related files, along with details of interpretation of the data contained therein. DBMS is a s/w system that allows access to data contained in a database. The objective of the DBMS is to provide a convenient and effective method of defining, storing and retrieving the information contained in the database.

The DBMS interfaces with application programs so that the data contained in the database can be used by multiple applications and users. The DBMS allows these users to access and manipulate the data contained in the database in a convenient and effective manner. In addition the DBMS exerts centralized control of the database, prevents unauthorized users from accessing the data and ensures privacy of data.

1.5 CHECK YOUR PROGRESS - *ANSWERS*

1.1, 1.2 & 1.3
Fill in the blanks

1) attributes
2) RDBMS
3) tuples
4) E.F.Codd
5) relation

True or false

1) False
2) True
3) True
4) True
5) False

1.6 QUESTIONS FOR SELF - STUDY

1) Explain the following terms
   i) Domain ii) Tuple iii) Relation iv) Attribute
2) Explain difference between DBMS and RDBMS.
3) Why relational data model is so popular ?
4) What are record based models ?
5) How RDBMS stores its data ?

1.7 SUGGESTED READINGS

Teach Yourself SQL in 21 Days **-** By Ryan K. Stephens Ronald R Plew

Using Oracle Application - By Jim Crum

NOTES

CHAPTER 2

DATA MANIPULATION & CONTROL

2.0 Objectives
2.1 Introduction
2.2 Subdivisions of SQL
2.3 Data Definition Language
2.4 Data Manipulation Language Commands
2.5 Data Control Language
2.6 Select Query and Clauses
2.7 Select Statement with Order by Clause
2.8 Group by Clause
2.9 Having Clause
2.10 String Operation
2.11 Distinct Rows
2.12 Rename Operation
2.13 Set Operations
2.14 Aggregate Functions
2.15 Nested Sub Queries
2.16 Embedded SQL
2.17 Dynamic SQL
2.18 Summary
2.19 Check Your Progress - *Answers*
2.20 Questions for Self – Study
2.21 Suggested Readings

2.0 OBJECTIVES

After reading this chapter you will able to

- state SQL, DDL, DML, DCL Statements
- explain Select,group by & having clause
- explain String & set operations
- describe Aggregate Functions
- describe Nested Sub Queries
- describe Embedded & Dynamic SQL

2.1 INTRODUCTION

In this chapter we study the query language : Structured Query Language (SQL) which uses a combination of Relational algebra and Relational calculus.

It is a data sub language used to organize, manage and retrieve data from relational database, which is managed by Relational Database Management System (RDBMS).

Vendors of DBMS like Oracle, IBM, DB2, Sybase, and Ingress use SQL as programming language for their database.

SQL originated with the system R project in 1974 at IBM's San Jose Research Centre.

Original version of SQL was SEQUEL which was an Application Program Interface (API) to the system R project.

The predecessor of SEQUEL was named SQUARE.

SQL-92 is the current standard and is the current version.

The SQL language can be used in two ways :

- Interactively or
- Embedded inside another program.

The SQL is used interactively to directly operate a database and produce the desired results. The user enters SQL command that is immediately executed. Most databases have a tool that allows interactive execution of the SQL language. These include SQL Base's SQL Talk, Oracle's SQL Plus, and Microsoft's SQL server 7 Query Analyzer.

The second way to execute a SQL command is by embedding it in another language such as Cobol, Pascal, BASIC, C, Visual Basic, Java, etc. The result of embedded SQL command is passed to the variables in the host program, which in turn will deal with them. The combination of SQL with a fourth-generation language brings together the best of two worlds and allows creation of user interfaces and database access in one application.

2.2 SUBDIVISIONS OF SQL

Regardless of whether SQL is embedded or used interactively, it can be divided into three groups of commands, depending on their purpose.

- Data Definition Language (DDL).
- Data Manipulation Language (DML).
- Data Control Language (DCL).

Data Definition Language :

Data Definition Language is a part of SQL that is responsible for the creation, updation and deletion of tables. It is responsible for creation of views and indexes also. The list of DDL commands is given below :

CREATE TABLE
ALTER TABLE
DROP TABLE
CREATE VIEW
CREATE INDEX

Data Manipulation Language :

Data manipulation commands manipulate (insert, delete, update and retrieve) data. The DML language includes commands that run queries and changes in data. It includes the following commands :

SELECT
UPDATE
DELETE
INSERT

Data Control Language :

The commands that form data control language are related to the security of the database performing tasks of assigning privileges so users can access certain objects in the database.

The DCL commands are :

GRANT
REVOKE
COMMIT
ROLLBACK

2.3 DATA DEFINITION LANGUAGE

The SQL DDL provides commands for defining relation schemas, deleting relations, creating indices, and modifying relation schemas.

The SQL DDL allows the specification of not only a set of relations but also information about each relation including :

- The schema for each relation.
- The domain of values associated with each attribute.
- The integrity constraints.
- The set of indices to be maintained for each relation.
- The security and authorization information for each relation.
- The physical storage structure of each relation on disk.

Domain/Data Types in SQL :

The SQL - 92 standard supports a variety of built-in domain types, including the following :

(1) Numeric data types include

- Integer numbers of various sizes
  INT or INTEGER
  SMALLINT
- Real numbers of various precision
  REAL
  DOUBLE PRECISION
  FLOAT (n)
- Formatted numbers can be represented by using
  DECIMAL (i, j) or
  DEC (i, j)
  NUMERIC (i, j) or NUMBER (i, j)

where, i - the precision, is the total number of decimal digits
and j - the scale, is the number of digits after the decimal point.

The default for scale is zero and the default for precision is implementation defined.

(2) Character string data types - are either fixed - length or varying - length.

CHAR (n) or CHARACTER (n) - is fixed length character string with user specified length n.

VARCHAR (n) - is a variable length character string, with user - specified maximum length n. The full form of CHARACTER VARYING (n), is equivalent.

(3) Date and Time data types :

There are new data types for date and time in SQL-92.

DATE - It is a calendar date containing year, month and day typically in the form
yyyy : mm : dd

TIME - It is the time of day, in hours, minutes and seconds, typically in the form
HH : MM : SS.

Varying length character strings, date and time were not part of the SQL - 89 standard.

In this section we will study the three Data Definition Language Commands :

CREATE TABLE
ALTER TABLE
DROP TABLE

1. CREATE TABLE Command :

The CREATE TABLE COMMAND is used to specify a new relation by giving it a name and specifying its attributes and constraints.

The attributes are specified first, and each attribute is given a name, a data type to specify its domain of values and any attribute constraints such as NOT NULL. The key, entity integrity and referential integrity constraints can be specified within the CREATE TABLE statement, after the attributes are declared.

Syntax of create table command :

```
CREATE TABLE table_name (
Column_name 1 data type [NOT NULL],
    :
    :
Column_name n data_type [NOT NULL]);
```

The variables are defined as follows :

If NOT NULL is not specified, the column can have NULL values.

table_name - is the name for the table.

column_name 1 to column_name n - are the valid column names or attributes.

NOT NULL – It specifies that column is mandatory. This feature allows you to prevent data from being entered into table without certain columns having data in them.

Examples of CREATE TABLE Command :

(1) Create Table Employee

```
(E_name          varchar2 (20)      NOT NULL,
B_Date           Date,
Salary           Decimal (10, 12)
Address          Varchar2 (50);
```

(2) Create Table Student

```
(Student_id      Varchar2 (20)      Not Null,
Last_Name        Varchar2 (20)      Not Null,
First_name       Varchar2 (20),
BDate            Date,
State            Varchar2 (20),
City  Varchar2 (20));
```

(3) Create Table Course

```
(Course_id       Varchar2 (5),
Department_id    Varchar2 (20),
Title            Varchar2 (20),
Description      Varchar2 (20));
```

Constraints in CREATE TABLE Command :

CREATE TABLE Command lets you enforce several kinds of constraints on a table : primary key, foreign key and check condition, unique condition.

A constraint clause can constrain a single column or group of columns in a table. There are two ways to specify constraints :

- As part of the column definition i.e. a *column constraint*.
- Or at the end of the create table command i.e. a *table constraint*.

Clauses that constrain several columns are the table constraints.

The Primary Key :

A table's primary key is the set of columns that uniquely identifies each row in the table. CREATE TABLE command specifies the primary key as follows :

```
create table table_name (
Column_name 1 data_type [not null],
    :
    :
Column_name n data type [NOT NULL],
[Constraint constraint_name]
```

[Primary key (Column_name A, Column_name B… Column_name X)]);

Variables are defined as follows :
table_name is the name for the table.
column_name 1 through column_name n are the valid column names
data_type is valid datatype
constraint which is optional
constraint_name identifies the primary key
column_name A through column_name X are the table's columns that compose the primary key.

Example :

```
Create table Employee
    (E_name             Varchar2 (20),
    B_Date              Date,
    Salary              Decimal (10, 2),
    Address             Varchar2 (80),
    Constraint          PK_Employee
    Primary key (Ename));
Create table_student
    (Student_id         Varchar2 (20),
    Last_name           Varchar2 (20)       NOT NULL,
    First_name          Varchar2 (20),
    B_Date              Date,
    State               Varchar2 (20),
    City                Varchar2 (20),
    Constraint          PK_Student
    Primary key         Student_id));
Create Table_Course
    (Course_id          Varchar2 (5),
    Department_id       Varchar2 (20),
    Title               Varchar2 (20),
    Description         Varchar2 (20),
    Constraint          PK_Course
    Primary key (Course_id, Department_id));
```

Note : We do not specify NOT NULL constraint for those columns which form the primary key, since those are the mandatory columns by default. Primary keys are subject to several restrictions.

(i) A column that is a part of the primary key cannot be NULL.
(ii) A column that is defined as LONG, or LONG RAW (ORACLE data types) cannot be a part of primary key.
(iii) The maximum number of columns in the primary key is 16.

Foreign Key : A foreign key is a combination of columns with values based on the primary key values from another table. A foreign key constraint also known as a referential integrity constraint, specifies that the values of the foreign key correspond to actual values of primary key in other table.

Create table command specifies the foreign key as follows :

```
Create Table table_name
(Column_name 1          data type [NOT NULL],
```

:

:

Column_name N data type [NOT NULL],

[constraint constraint_name

Foreign key (column_name F1 … Column_name FN) references referenced-table (column_name P1, … column_name P_N)]);

table_name - is the name for the table.
Column_name 1 through column_name N are the valid columns.
constraint_name is the name given to foreign key.
referenced_table - is the name of the table referenced by the foreign key declaration.
column_name F_1 through column_name F_N are the columns that compose the foreign key.
Column_name P_1 through column_name P_N are the columns that compose the primary key in referenced-table.

Examples :

```
Create table_department
    (Department_id          Varchar2 (20),
    Department_name         Varchar2 (20),
    Constraint              PK_Department
    Primary key             (Department_id));
Create table_course
    (Course_id              Varchar2 (20),
    Department_id           Varchar2 (20),
    Title                   Varchar2 (20),
    Description             Varchar2 (20),
    Constraint              PK_course
    Primary key (Course_id, Department_id),
    Constraint              FK - course
Foreign key (Department_id) references Department (Department_id));
```

Thus, primary key of course table is (Course_id, Department_id).

The primary key of Department table is (Department_id).

Foreign key of course table is (Department_id) which references the department table.

When you define a foreign key, the DBMS verifies the following :

(1) A primary key has been defined for table referenced by the foreign key.
(2) The number of columns composing the foreign key matches the number of primary key columns in the referenced table.
(3) The datatype and width of each foreign key columns matches the datatype and width of each primary key column in the referenced table.

Unique Constraint or Candidate key :

A candidate key is a combination of one or more columns, the values of which uniquely identify each row of the table. Create table command specifies the unique constraint as follows :

```
CREATE  TABLE  table_name
    (column_name 1      data_type           [NOT NULL],
        :
        :
    column_name n       data_type           [NOT NULL],
    [constraint         constraint_name
    Unique  (Column_name A,……. Column_nameX)]);
```

Example :

```
Create table student
    (Student_id         Varchar2 (20),
    Last_name           Varchar2 (20),          NOT NULL,
    First_name          Varchar2 (20),          NOT NULL,
    BDate               Date,
    State               Varchar2 (20),
    City                Varchar2 (20),
    Constraint          UK-student
    Unique              (last_name, first_name),
    Constraint          PK-student
    Primary key         (Student_id));
```

A unique constraint is not a substitute for a primary key. Two differences between primary key and unique constraints are :

(1) A table can have only one primary key, but it can have many unique constraints.

(2) When a primary key is defined, the columns that compose the primary key are automatically mandatory. When a unique constraint is declared, the columns that compose the unique constraint are not automatically defined to be mandatory, you must also specify that the column is NOT NULL.

Check Constraint :

Using CHECK constraint SQL can specify the data validation for column during table creation. CHECK clause is a Boolean condition that is either TRUE or FALSE. If the condition evaluates to TRUE, the column value is accepted by database, if the condition evaluates to FALSE, database will return an error code.

The check constraint is declared in CREATE TABLE statement using the syntax :

Column_name datatype [constraint constraint_name] [CHECK (Condition)]

The variables are defined as follows :

Column_name - is the column name

data_type - is the column's data type

constraint_name - is the name given to check constraint condition is the legal SQL

Condition that returns a Boolean value.

Examples :

```
Create table_worker
    (NameVarchar2 (25)                          NOT NULL,
    Age         Number      Constraint CK_worker
    CHECK (Age Between 18 AND 65) );
Create table_instructor
    (Instructor_id      Varchar2 (20),
    Department_id       Varchar2 (20)           NOT NULL,
    Name                Varchar2 (25),
    Position            Varchar2 (25)
    Constraint          CK_instructor
CHECK (Position in ('ASSISTANT PROFESSOR', 'ASSOCIATE PROFESSOR', 'PROFESSOR')),
    Address             Varchar2 (25),
    Constraint          PK_instructor
    Primary key         (Instructor_id));
```

If the position of the instructor is not one of the three legal values, DBMS will return an error code indicating that a check constraint has been violated.

More than one column can have check constraint.

```
Create table_Patient
```

```
(Patient_id          Varchar2 (25)          Primary key,
    Body_Temp        Number (4, 1)
    Constraint       Patient_BT
    CHECK (Body_Temp >= 60.0 and
    Body_Temp <= 110.0),
    Insurance_StatusChar(1)
    Constraint              Patient_IS
    CHECK  (Insurance-Status in ('Y', 'y', 'N', 'n')));
```

One column can have more than one CHECK constraint.

```
Create table_Loan - application
    (loan_app_no          number (6)        primary key,
    Name                  Varchar2 (20),
    Amount_requestednumber (9, 2)            NOT NULL,
    Amount_approvednumber (9, 2)
    Constraint            Amount_approved_limit
    Check (Amount_approved <= 10,00,000)
    Constraint Amount_Approved_Interval
    Check (Mod (Amount_Approved, 1000) = 0));
```

Establishing a Default value for a column :

By using DEFAULT clause when defining a column, you can establish a default value for that column. This default value is used for a column, whenever, row is inserted into the table without specifying the column in the INSERT statement.

Example :

```
Create table_student
    (Student_id         Varchar2 (20),
    Last_name           Varchar2 (20)        NOT NULL,
    First_name          Varchar2 (20)        NOT NULL,
    B_Date              Date,
    State               Varchar2 (20),
    City                Varchar2 (20),       DEFAULT  'PUNE'.
    Constraint          PK_student
    Primary key    (Student_id);
```

2. ALTER TABLE Command :

You can modify a table's definition using ALTER TABLE command. This statement changes the structure of a table, not its contents. Using ALTER TABLE command, you can make following changes to the table.

(1) Adding a new column to an existing table.

```
ALTER  TABLE  table_name
ADD (Column_name                          datatype
    :
    :
  Column_name n                           datatype);
```

Example :

```
SQL>    Describe Department;
        Name                    NULL?        Type
        Department_id                  Varachar2 (20)
        Department_name                Varachar2 (20)
SQL>    Alter table Department         ADD (University
        Varchar2 (20),
```

```
        No_of_student  Number (3));
SQL>    Describe  Department;
        Name              Null              Type
        Department_id                       Varachar2 (20)
        Department_Name                     Varachar2 (20)
        University                          Varachar2 (20)
        No_of_student                       Varachar2 (20)
```

(2) Modify an existing column in the existing table.

```
ALTER  TABLE  table_name
MODIFY  (Column_name        datatype  :  constraint,
…      Column_name          datatype  :  constraint,);
```

A column in the table can be modified in following ways -

(i) Changing a column definition from NOT NULL to NULL i.e. from mandatory to optional

Consider a table ex_table.

```
SQL>    describe ex_table;
        Name          NULL?            Type
        Record_no     NOTNULL          Numbers (38)
        Description   Varchar2 (40)
        Current_value NOT NULL           Number
SQL>    Alter Table ex_table;
        modify (current_value number     Null);
        Table altered
SQL>    Describe ex_table;
        Name          NULL?            Type
        Record_No     NOT NULL         Number (38)
        Description   Varchar2 (40)
        Current_value                  Number
```

(ii) Changing a column definition from NULL to NOT NULL.

If a table is empty, you can define a column to be NOT NULL. However, if table is not empty, you cannot change a column to NOT NULL unless every row in the table has a value for that particular column.

(iii) Increasing and Decreasing a Column's Width :

You can increase a character column's width and can increase the number of digits in a number column at any time.

Example :

```
SQL>    Describe ex_table;
        Name                    NULL ?          Type
        Record_No NOT NULL      Number (38)
        Description Varchar2 (40)
        Current_value           NOT NULL        Number
SQL>    Alter table ex_table
        modify (Description     Varchar2 (50));
        Table altered
SQL>    Describe ex_table;
        Name                    NULL ?          Type
        Record_No               NOT NULL        Number (38)
        Description             Varchar2 (50)
        Current_value           NOT NULL        Number
```

You can decrease a column's width only if the table is empty or if that column is NULL for every row of table.

(3) Adding a constraint to an existing table :

Any constraint i.e. a primary key, foreign key, unique key or check constraint can be added to an existing table using ALTER TABLE command.

```
ALTER  TABLE  table_name
```

ADD (constraint)

Example :

```
SQL>  Create Table ex_table
      (Record_No     Number (38),
      Description    Varchar2 (40),
      Current_value Number);
      Table created
SQL>  Alter Table ex_table  add
      (Constraint PK_ex_table  primary key (Record-No));
      Table Altered.
```

(4) Dropping the constraints

```
ALTER  TABLE  table_name
DROP  Primary key
Using this you can drop primary key of table.
ALTER  TABLE  Table_name
DROP  constraint  constraint_name
```

Using this you can drop any constraint of the table.

Rules for adding or modifying a column :

Following are the rules for adding column to a table :

(1) You may add a column at any time if NOT NULL is not specified.

(2) You may add a NOT NULL column in three steps :

(i) Add a column without NOT NULL specified,

(ii) Fill every row in that column with data,

(iii) Modify the column to be NOT NULL.

Following are the rules to modify a column.

(1) You can increase a character column's width at any time.

(2) You can increase the number of digits in a NUMBER column at any time.

(3) You can increase or decrease the number of places in a NUMBER column at any time.

If a column is NULL for every row of the table, you can make following changes.

(i) You can change its data type

(ii) You can decrease a character column's width

(iii) You can decrease the number of digits in a NUMBER column.

3. DROP TABLE Command :

Dropping a table means to remove the table's definition from the database. DROP TABLE command is used to drop the table as follows :

```
DROP  TABLE  table_name;
```

Example :

```
(1)SQL >    Drop table_student;
      Table dropped
(2)SQL >    Drop table instructor;
      Table dropped.
```

You drop a table only when you no longer need it.

Note : The truncate command in ORACLE can also be used to remove only the rows or data in the table and not the table definition.

Example :

```
Truncate student
Table truncated
```

Truncating cannot be rolled back.

2.4 DATA MANIPULATION LANGUAGE COMMANDS

The SQL DML includes commands to insert tuples into database, to delete tuples from database and to modify tuples in the database.

It includes a query language based on both relational algebra and tuple relational calculus.

In this section we'll study following SQL DML commands.

INSERT
DELETE
UPDATE
SELECT

1. INSERT Command :

The syntax of insert statement is :

```
INSERT INTO table_name
[(column_name [ , column_name] …… [ , column_name])]
VALUES
(column_value [ , column_value] …… [ , column_value]);
```

The variables are defined as follows :

Table_name - is the table in which to insert the row.

column_name - is a column belonging to table.

column_value - is a literal value or an expression whose type matches the corresponding column_name.

The number of columns in the list of column_names must match the number of literal values or expressions that appear in parenthesis after the keyword *values*.

Example :

```
SQL>  Insert into Employee
      (E_name, B_Date, Salary, Address)
      Values
      ('Sachin', '21-MAR-73', 50000.00, 'Mumbai');
      row created
SQL>  Insert into student
      (Student_id, Last_name, First_name)
      Values
      ('SE201', 'Tendulkar', 'Sachin');
      row created
```

If the column names specified in Insert statement are more than values, then it returns an error.

Column and value datatype must match.

According to the syntax of INSERT statement, column list is an optional element. Therefore, if you do not specify the column names to be assigned values, it (DBMS) by default uses all the columns. The column order that DBMS uses is the order in which the columns were specified, when the table was created. However, use of Insert statement without column list is dangerous.

For example,

```
SQL>  Describe ex_class;
      Name                  NULL ?      Type
      Class_building        NOT NULL    Varchar2 (25)
      Class_room            NOT NULL    Varchar2 (25)
      Seating_capacity      Number (38)
SQL>  Insert into ex_class
      Values
      ('250', 'Kothrud Pune', 500);
      1 row created.
```

The row is successfully inserted into the table, because, value and column data types were matching.

But the value 250 is not a correct value for column class_building.

The use of insert without column list may cause following problems.

1. The table definition might change, the number of columns might decrease or increase, and the INSERT fails as a result.
2. The INSERT statement might succeed but the wrong data could be entered in the table.

2. DELETE Command :

The syntax of delete statement is :

```
DELETE FROM table_name
[WHERE condition]
```

The variables are defined as follows :

table_name - is the table to be updated.

condition - is a valid SQL condition.

DELETE Command without WHERE clause will empty the table completely.

Example :

```
SQL> Delete from Student
     Where Student_id = 'SE 201';
     1 row deleted.
SQL> Detete from student
     Where first_name = 'Sachin' and
     Student_id ='SE 202';
     1 row deleted.
```

3. UPDATE Command :

If you want to modify existing data in the database, UPDATE command can be used to do that. With this statement you can update zero or more rows in a table.

The syntax of UPDATE command is :

```
UPDATE table_name
SET  column_name : : expression
     [, column_name : : expression]
     [, column_name : : expression]
     [where condition]
```

The variables are defined as follows :

table_name is the table to be updated

column_name is a column in the table being updated.

expression is a valid SQL expression.

condition is a valid SQL condition.

The UPDATE statement references a single table and assigns an expression to at least one column. The WHERE clause is optional; if an UPDATE statement does not contain a WHERE clause, the assignment of a value to a column will be applied to all rows in the table.

Example :

```
SQL> Update Student
     Set
     City = 'Pune',
     State = 'Maharashtra';
SQL> Update Instructor
     Set
     Position = 'Professor'
     where
     Instructor_id = 'P3021';
```

SQL Grammar :

Here, are some grammatical requirements to keep in mind when you are working with SQL.

1. Every SQL statement is terminated by a semicolon.
2. An SQL statement can be entered on one line or split across several lines for clarity.

3. SQL isn't case sensitive. You can mix uppercase and lowercase when referencing SQL keywords (Such as SELECT and INSERT), table names, and column names.

However, case does matter when referencing to the contents of a column.

For Example : If you ask for all customers whose last names begin with 'a' and all customer names are stored in uppercase, you won't receive any rows at all.

4. SELECT Command :

The basic structure of an SQL expression consists of three clauses :

select, from and where

- The select clause corresponds to the projection operation of the relational algebra.
  It is used to list the attributes desired in the result of a query.
- The from clause corresponds to the cartesian product operation of the relational algebra. It lists the relations to be scanned in the elevation of the expression.
- The where clause corresponds to the selection predicate of the relational algebra.
  It consists of predicate involving attributes of the relations that appear in the from clause.

  Simple SQL query i.e. select statement has the form :

  select $A_1, A_2, \ldots\ldots, A_n$

  from $r_1, r_2, \ldots\ldots, r_m$

  where P.

  The variables are defined as follows :

  $A_1, A_2, \ldots, A_n$ represent the attributes.

  $r_1, r_2, \ldots, r_m$ represent the relations from which the attributes are selected.

  P - is the predicate.

  This query is equivalent to the relational algebra expression

  $$\Pi_{A_1 A_2 \ldots A_n} (\sigma_p (r_1 \times r_2 \times r_3 \ldots \times r_m))$$

  where clause is optional. If the where clause is omitted, the predicate P is true.

Select clause forms the cartesian product of relations named in the *from* clause, performs a relational algebra selection using the *where* clause and then projects the results onto the attributes of the select clause.

A simple select statement :

At a minimum, select statement contains the following two elements.

- The select list, the list of columns to be retrieved.
- The from clause, the tables from which to retrieve the rows.

Example : Consider the student database table.

(1) A simple select statement - a query that retrieves only student_id from the student table is given

```
SQL> select student_id
     from student;
     student_id
     ----------
     S 10231
     S 10232
     S 10233
     S 10234
     S 10235
     S 10236
```

6 rows selected.

(2) To select student_id and students Last name, the select statement is :

```
SQL>  select student_id, First_name
      from student;
```

student_id	First_name
S 10231	Sachin
S 10232	Rahul
S 10233	Ajay
S 10234	Sunil
S 10235	Kapil
S 10236	Anil

6 rows selected.

To select all columns in the table you can use

```
select  *
from table_name;
```

Example :

```
SQL>  select  *
      from student;
```

student_id	Last_name	First_name	B Date	State	City
S 10231	Deshpande	Sachin	12/3/78	Maharashtra	Pune
S 10232	Gandhi	Rahul	9/2/58	Delhi	Delhi
S 10233	Kapur	Ajay	7/12/62	Maharashtra	Bombay
S 10234	Kulkarni	Sunil	6/9/75	Maharashtra	Pune
S 10235	Dev	Kapil	2/3/71	Tamilnadu	Madras
S 10236	Kumar	Anil	5/9/80	Maharashtra	Bombay

The results returned by every SELECT statement constitutes a temporary table. Each received record is a row in this temporary table, and each element of the select list is a column. If a query does not return any record, the temporary - table can be thought of as empty.

Expressions in the select list :

In addition to specifying columns, you also can specify expressions in the select list.

Following arithmetic operators can be used in select list :

Description	Operator
Addition	+
Subtraction	–
Multiplication	*
Division	/

For example, consider the following queries using operators in select list :

```
SQL>Select E_name, Salary * 1000
    from Employee;
```

E_name	Salary * 1000
Sachin	1,00,00,000
Rahul	2,00,00,000
Ajay	1,00,00,000
Anil	1,00,00,000

4 rows selected.

```
SQL>Select Ename, Salary + 10000
    from Employee;
```

E_name	Salary + 10000
Sachin	20,000
Rahul	30,000
Ajay	20,000
Anil	30,000

4 rows selected.

Select statement using where clause :

select and *from* clauses provide you with either some columns and all rows or all columns and all rows. But if you want only certain rows, you need to add another clause, the *where* clause.

where clause consists of one or more conditions that must be satisfied before a row is retrieved by the query.

It searches for a condition and narrows your selection of data.

For example, consider select statement with where clause given below :

```
SQL>Select Student_id, First_name
    from Student
    where Student_id = 'S10234';
```

Student_id	First_name
S10234	Sunil

1 row selected

```
SQL>Select E_name  Salary
    from Employee
    where Salary  > 10000;
```

E_name	Salary
Rahul	20000
Anil	20000

2 row selected

where uses the logical connectives : **and, or** and **not**.

where clause uses the comparison operators

Description	Operator
Less than	<
Less than or equal to	<=
Greater than	>
Greater than or equal to	>=
Equal to	=
Not equal to	!= or < >

```
SQL>Select E_name, Salary
    from  Employee
    where Salary  >  10000 and E_name  =  Anil
```

Ename	Salary
Anil	20000

1 row selected.

5. Views in SQL :

A view in SQL terminology is a single table that is derived from other tables. These other tables could be base tables or previously defined views. A view does not necessarily exist in physical form; it is considered a virtual table in contrast to base tables whose tuples are actually stored in the database. This limits the possible update operations that can be applied to views but does not provide any limitations on querying a view. We can think of view as a way specifying a table that we need not exist physically.

Specification of Views in SQL :

The command to specify a view is CREATE VIEW. 'We give the view a table name, a list of attribute names, and a query to specify the contents of view. If none of the view attributes result from applying functions or arithmetic operations, we do not have to specify attribute names for the view as they will be the same as the names of the attributes of the defining tables.

Example :

Consider the following relation scheme and corresponding relation.

employee_schema (emp_name, street, city)
works_schema (emp_name, comp_name, salary
company_schema (comp_name, city)

emp_name	street	city
Sachin	XYZ	Pune
Rahul	ABC	Bombay
Raj	ABC	Pune
Ajay	XYZ	Bombay
Anil	XYZ	Delhi
Sunil	ABC	Bombay

emp_name	Comp_name	salary
Sachin	TCS	10000
Rahul	MBT	12000
Raj	PCS	13000
Ajay	MBT	14000
Anil	PCS	15000
Sunil	TCS	11000

Comp_name	city
TCS	Delhi
MBT	Bombay
PCS	Pune

```
Create view emp_detail (emp, comp, street, city)
As  select  C.emp_name, C.comp_name, E.street, E.city
    from Employee E.company C
    where E.emp_name = C.emp_name;
```

A view is always up date; if we modify the base tables on which the view is defined, the view automatically reflects these changes. Hence, the view is not realized at the time of view definition but rather at the time we specify a query on the view. It is the responsibility of the DBMS and not the user to make sure that the view is up to date.

If we do not need a view any more, we can use the DROP VIEW command to dispose of it.

```
Drop View emp_detail;
```

Updating of views :

(1) A view with a single defining table is up datable if the view attributes contain the primary key or some other candidate key of the base relation, because this maps each view tuple to a single base tuple.

(2) Views defined on multiple tables using joins are generally not updatable.

(3) Views defined using grouping and aggregate functions are not updatable.

Example :

Consider the view consisting of branch names and names of customers who have either an account or a loan at that branch.

```
SQL>Create view all_customer as
        (select branch_name, customer_name
            from depositor, account
            where depositor·account_number =
        account·account·account_no)
        Union
        (select branch_name, customer_name
            from borrower
            where borrower·loan_number = loan·loan_number);
```

The attribute names of a view can be specified explicitly as follows :

```
SQL>  Create view branch_total_loan (branch_name,
      total_loan) as
      select branch_name, sum (amount)
      from loan
          group by branch_name;
```

6. Indexes in SQL : SQL has statements to create and drop indexes on attributes of base relation. These commands are generally considered to be part of the SQL data definition language (DDL).

An index is a physical access structure that is specified on one or more attributes of the relation. The attributes on which an index is created are termed indexing attributes. An index makes accusing tuples based on conditions that involve its indexing attributes more efficient. This means that in general executing a query will take less time if some attributes involved in the query conditions were indexed than if they were not. This improvement can be dramatic for queries where large relations are involved. In general, if attributes used in selection conditions and in join conditions of a query are indexed, the execution time of the query is greatly improved.

In SQL indexed can be created and dropped dynamically. The create Index command is used to specify an index. Each index is given a name, which is used to drop the index when we do not need it any more.

Example :

```
Create Index  Emp_Index
    ON Employee (Emp_name);
```

In general, the index is arranged in ascending order of the indexing attribute values. If we want the values in descending order we can add the keyword DESC after the attribute name. The default in ASC for ascending. We can also create an index on a combination of attributes.

Example :

```
Create  Index  Emp_Index1
    ON  Employee (Emp_name ASC,
            Comp_name  DESC);
```

There are two additional options on indexes in SQL. The first is to specify the key constraint on the indexing attribute or combination of attributes.

The keyword unique following the CREATE command is used to specify a key. The second option on index creation is to specify whether on index is clustering index. The keyword cluster is used in this case of the end of the create Index command. A base relation can have atmost one clustering index but any number of non_clustering indexes.

To drop an index, we issue the Drop Index command. The reason for dropping indexes is that they are expensive to maintain whenever the base relation is updated and they require additional storage. However, the indexes that specify a key constraint should not be dropped as long as we want the system to continue enforcing that constraint.

Example :

```
Drop Index Emp_Index;
```

7. Sequences

The quickest way to retrieve data from a table is to have a column in the table whose data uniquely identifies a row.By using this column and a specific value in the WHERE condition of a SELECT sentence the oracle engine will be able to identify and retrieve the row the fastest.

To achieve this , a constraint is attached to a specific column in the table that ensures that the column is never left empty and that the data values in the column are unique.Since human beings do data entry,it is quite likely that a duplicate value could be entered ,which violets this constraint and the entire row is rejected.

If the value entered into this column is computer generated it will always fulfill the unique constraint and the row will always be accepted for storage.

Oracle provides an object called a sequence that can generate numeric values. The value generated can have a maximum of 38 digits. A sequence can be defined to:

-Generate numbers in ascending or descending order

-Provide intervals between numbers

-Caching of sequence numbers in memory to speed up their availability

A sequence is an independent object and can be used with any table that requires its output.

Creating Sequences

Always give sequence a name so that it can be referenced later when required.

The minimum information required for generating numbers using a sequence is :

-The starting number

-The maximum number that can be generated by a sequence

- The increment value for generating the next number

This information is provided to oracle at the time of sequence creation

Syntax:

```
CREATE SEQUENCE <SequenceName>
[INCREMENT BY  <IntegerValue>
[START WITH <IntegerValue>
MAXVALUE <IntegerValue> / NOMAXVALUE
MINVALUE <IntegerValue> /NOMINVALUE
CYCLE/NOCYLCLE
CACHE <IntegerValue>/NOCACHE
ORDER/NOORDER ]
```

Keywords and Parameters

INCREMENT BY : -Specifies the interval between sequence numbers. It can be any positive or negative value but not zero.If this clause is omitted ,the default value is 1 .

MINVALUE :- Specifies the sequence minimum value.

NOMINVALUE : Specifies a minimum value of 1 for an ascending sequence and –(10)^26 for a descending sequence.

MAXVALUE: Specifies the maximum value that a sequence can generate.

NOMAXVALUE : Specifies a maximum of 10^27 for an ascending sequence or -1 for a descending sequence. This is the default clause.

START WITH : Speciifes the first sequence number to be generated. The default for an ascending sequence is the sequence minimum value(1) and for a descending sequence, it is the maximum value(-1)

CYCLE: Specifies that the sequence continues to generate repeat values after reaching either its maximum value.

NOCYCLE: Specifies that a sequence cannot generate more values after reaching the maximum value.

CACHE : Specifies how many values of a sequence oracle pre-allocates and keeps in memory for faster access.The minimum value for this parameter is two.

NOCHACHE : Specifies that values of a sequence are not pre-allocated.

ORDER : This guarantees that sequence numbers are generated in the order of request.This is only necessary if using parallel server in parallel mode option .In exclusive mode option ,a sequence always generates numbers in order.

NOORDER : This does not guarantee sequence numbers are generated in order of request.This is only necessary if you are using parallel server in parallel mode option. If the ORDER/NOORDER clause is omitted , a sequence takes the NOORDER clause by default.

Example

Create a sequence by the name ADDR_SEQ ,which will generate numbers from 1 uptp 9999 in ascending order with an interval of 1.The sequence must restart from the number 1 after generating number 999.

CREATE SEQUENCE ADDR_SEQ **INCREMENT BY 1 START WITH 1 MINVALUE 1 MAXVALUE 999 CYCLE ;**

Referencing a sequence

Once a sequence is created SQL can be used to view the values held in its cache.To simply view sequence value use a SELECT sentence as described below.

SELECT <SequenceName>.Nextval from DUAL ;

This will display the next value held in the cache on the VDU screen. Everytime nextval references a sequence its output is automatically incremented from the old value to the new value ready for use.

To reference the current value of a sequence:

SELECT <SequenceName>.CurrVal FROM DUAL;

Dropping a Sequence

The DROP SEQUENCE command is used to remove the sequence from the database.

Syntax:

DROP SEQUENCE <SequenceName> ;

2.5 DATA CONTROL LANGUAGE

The data control language commands are related to the security of database. They perform tasks of assigning privilages, so users can access certain objects in the database. This section deals with DCL commands.

1. GRANT Command :

The objects created by one user are not accessible by another user unless the owner of those objects gives such permissions to other users. These permissions can be given by using the **GRANT** statement. One user can grant permission to another user if he is the owner of the object or has the permission to grant access to other users.

The grant statement provides various types of access to database objects such as tables, views and sequences.

Syntax :

GRANT {object privilages}
ON object name
To user name
[with GRANT OPTION]

Object privilages :

Each object privilage that is granted authorizes the grantee to perform some operations on the object. The user can grant all the privilages or grant only specific object privilages.

The list of object privilages is as follows :

Alter - allows the grantee to change the table definition with the ALTER TABLE command.

Delete - allows the grantee to remove the records from the table with the DELETE command.
Index - allows the grantee to create an index on table with the CREATE INDEX command.
Insert - allows the grantee to add records to the table with the INSERT command.
Select - allows the grantee to query the tables with SELECT command.
Update - allows the grantee to modify the records in tables with UPDATE command.
With grant option : It allows the grantee to grant object privilages to other users.

Example 1 : Grant all privilages on student table to user Pradeep.

```
SQL >  GRANT ALL
       ON student
       To Pradeep;
```

Example 2 : Grant select and update privilages on student table to mita

```
SQL>   GRANT SELECT, UPDATE
       ON student
    To Mita;
```

Example 3 : Grant all privilages on student table to user Sachin with grant option.

```
SQL>   GRANT ALL
       ON student
    To Sachin
       WITH GRANT OPTION;
```

2. REVOKE Command :

The REVOKE statement is used to deny the grant given on an object.

Syntax :

```
REVOKE {object privilages}
        ON object name
        FROM user name;
```

The list of object privilages is :

Alter - allows the grantee to change the table definition with the ALTER TABLE command.
Delete - allows the grantee to remove the records from the table with the DELETE command.
Index - allows the grantee to create an index on table with the CREATE INDEX command.
Insert - allows the grantee to add records to the table with the INSERT command.
Select - allows the grantee to query the tables with SELECT command.
Update - allows the grantee to modify the records in tables with UPDATE command.

You cannot use REVOKE command to perform following operations :

1. Revoke the object privilages that you didn't grant to the revokee.
2. Revoke the object privilages granted through the operating system.

Example 1 : Revoke Delete privilege on student table from Pradeep.

```
REVOKE DELETE
ON student
From    Pradeep;
```

Example 2 : Revoke the remaining privilages on student that were granted to Pradeep.

```
Revoke ALL
ON  student
FROM   Pradeep
```

3. COMMIT Command :

Commit command is used to permanently record all changes that the user has made to the database since the last commit command was issued or since the beginning of the database session.

Syntax :

COMMIT;

Implicity COMMIT :

The actions that will force a commit to occur even without your instructing it to are :

quit, exit,
create table or create view
drop table or drop view
grant or revoke
connect or disconnect
alter
audit and non-audit

Using any of these commands is just like using commit. Until you commit, only you can see how your work affects the tables. Anyone else with access to these tables will continue to get the old information.

4. ROLLBACK command :

The ROLLBACK statement does the exact opposite of the commit statement. It ends the transaction but undoes any changes made during the transaction. Rollback is useful for two reasons :

(1) If you have made a mistake, such as deleting the wrong row for a table, you can use rollback to restore the original data. Rollback will take you back to intermediate statement in the current transaction, which means that you do not have to erase the entire transaction.

(2) ROLLBACK is useful if you have started a transaction that you cannot complete. This might occur if you have a logical problem or if there is an SQL statement that does not execute successfully. In such cases rollback allows you to return to the starting point to allow you to take corrective action and perhaps try again.

Syntax : ROLLBACK [WORK] [TO [SAVEPOINT] save point]

where

WORK - is optional and is provided for ANSI compatibility

SAVEPOINT - is optional and is used to rollback a partial transaction, as far as the specified save point.

Savepoint : is a savepoint created during the current transaction.

Using rollback without savepoint clause.

1. Ends the transaction.
2. Undoes all the changes in the current transaction.
3. Erases all savepoints in that transaction
4. Releases the transaction locks.

Using rollback with the to savepoint clause.

1. Rolls back just a portion of the transaction.
2. Retains the savepoint rolled back to, but losses those created after the named savepoint.
3. Releases all tables and row locks that were acquired since the savepoint was taken.

Example :

To rollback entire transaction : ROLLBACK,

To rollback to savepoint sps : ROLLBACK TO SAVEPOINT sps;

Savepoints :

Savepoints mark and save the current point in the current processing of a transaction. Used with the ROLLBACK statement, savepoints can undo part of a transaction.

By default the maximum number of savepoints per transaction is 5. An active savepoint is the one that is specified since the last commit or rollback.

Syntax : SAVEPOINT savepoint :

After a savepoint, is created, you can either continue processing, commit your work rollback the entire transaction, or rollback to the savepoint.

2.6 SELECT QUERY AND CLAUSES

The basic structure of an SQL expression consists of three clauses :

select, from and where,

- The select clause corresponds to the projection operation of the relational algebra.
  It is used to list the attributes desired in the result of a query.
- The from clause corresponds to the cartesian product operation of the relational algebra. It lists the relations to be scanned in the elevation of the expression.
- The where clause corresponds to the selection predicate of the relational algebra.
  It consists of predicate involving attributes of the relations that appear in the from clause.

Simple SQL query i.e. select statement has the form :

select $A_1, A_2, \ldots\ldots, A_n$

from $r_1, r_2, \ldots\ldots, r_m$

where P.

The variables are defined as follows :

$A_1, A_2, \ldots, A_n$ represent the attributes.

$r_1, r_2, \ldots, r_m$ represent the relations from which the attributes are selected.

P - is the predicate.

This query is equivalent to the relational algebra expression

$$\Pi_{A_1 A_2 \ldots A_n} (\sigma_p (r_1 \times r_2 \times \ldots \times r_m))$$

where clause is optional. If the where clause is omitted, the predicate P is true.

Select clause forms the cartesian product of relations named in the *from* clause, performs a relational algebra selection using the *where* clause and then projects the results onto the attributes of the select clause.

The purpose of select statement is to retrieve and display data from one or more database tables It is an extremely powerful statement capable of performing the equivalent relational algebra's Selection, Projection, and Join operations in a single statement. Select is the most frequently used SQL command and has the following general form :

```
SELECT      DISTINCT |ALL]
FROM        Table_Name [alias][,…]
[WHERE      condition]
[GROUP BY   column_List] [HAVING condition]
[ORDER BY   column_List]
```

The sequence of processing in a select statement is :

FROM
WHERE
GROUP BY
HAVING

SELECT
ORDER BY

The order of the clauses in the select command can not be changed. The only two mandatory columns are : SELECT and FROM, the remainder are optional.

1. Expressions in the select list :

In addition to specifying columns, you also can specify expressions in the select list.

Following arithmetic operators can be used in select list :

Description	Operator
Addition	+
Subtraction	–
Multiplication	*
Division	/

For example, consider the following queries using operators in select list :

```
SQL>Select E_name, Salary * 1000

from Employee;
        E_name                  Salary * 1000
        Sachin                  1,00,00,000
        Rahul                   2,00,00,000
        Ajay                    1,00,00,000
        Anil                    1,00,00,000
        4 rows selected.
SQL>    Select E_name, Salary + 10000
        from Employee;
        E_nam          Salary + 10000
        Sachin              20,000
        Rahul               30,000
        Ajay                20,000
        Anil                30,000
    4 rows selected.
```

2. Select statement using where clause :

select and *from* clauses provide you with either some columns and all rows or all columns and all rows. But if you want only certain rows, you need to add another clause, the *where* clause.

where clause consists of one or more conditions that must be satisfied before a row is retrieved by the query.

It searches for a condition and narrows your selection of data.

For example, consider select statement with where clause given below :

```
SQL>    Select Student_id, First_Name
        from Student
        where Student_id  =  'S10234';
        Student_id                          First_name
        S10234 Sunil
        1 row selected
SQL>    Select E_name  Salary
        from Employee
        where Salary  >  10000
        E_name                              Salary
        Rahul                               20000
        Anil                                20000
        2 row selected
```

where uses the logical connectives : **and, or** and **not**.

where clause uses the comparison operators

Description	Operator
Less than	<
Less than or equal to	<=
Greater than	>
Greater than or equal to	>=
Equal to	=
Not equal to	!= or < >

```
SQL>Select E_name, Salary
        from  Employee
        where Salary>10000 and Ename  =  Anil
    E_name                                  Salary
    ------------------------------------------------
    Anil                                    20000
    1 row selected.
```

Range Searching

In order to select data that is within a range of values ,the BETWEEN operator is used. The BETWEEN operator allows the selection of rows that contain values within a specified lower and upper limit. The range coded after the word BETWEEN is inclusive.

The lower value must be coded first.The two values in between the range must be linked with the keyword AND.The BETWEEN operator can be used with both character and numeric data types.However the datatypes can not be mixed.i.e the lower value of a range of values from a character column and the other from a numeric column.

Example 1 : List the transactions performed in months of January to March

Solution :

```
SELECT * FROM TRANS_MSTR WHERE TO_CHAR(DT,'MM') BETWEEN 01 AND
03 ;
```

Equivalent to

```
SELECT * FROM TRANS_MSTR WHERE TO_CHAR (DT,'MM')>=01 AND
TO_CHAR(DT,'MM')<=03;
```

Explanation

The above select will retrieve all those records from the ACCT_MSTR table where the value held in the DT field is between 01 and 03 (both values inclusive).This is done using TO_CHAR() function which extracts the month value from the DT field. This is then compared using the AND operator.

Example 2 : List all the accounts which have not been accessed in the fourth quarter of the financial year

Solution

```
SELECT DISTINCT  FROM TRANS_MSTR WHERE TO_CHAR(DT,'MM') NOT
BETWEEN 01 AND 04 ;
```

Explanation

The above select will retrieve all those records from the ACCT_MSTR table where the value held in the DT field is not between 01 and 04(both values inclusive).This is done using TO_CHAR() function which extracts the month value from the DT field and then compares them using the not and the between operator.

2.7 SELECT STATEMENT WITH ORDER BY CLAUSE

ORDER BY clause is similar to the GROUP BY clause. The ORDER BY clause enables you to sort your data in either ascending or descending order.

The ORDER BY clause consists of a list of column identifiers that the result is to be sorted on, separated by columns. A column identifier may be either a column name or a column number.

It is possible to include more than one element in the ORDER BY clause. The major sort key determines the overall order of the result table

If the values of the major sort key are unique, there is no need for additional keys to control the sort. However, if the values of the major sort key are not unique, there may be multiple rows in the result table with the same value for the major sort key. In this case it may be desirable to order rows with the same value for the major sort key by some additional sort key. If a second element appears in the ORDER BY clause, it is called a ***minor sort key.***

Example : Consider the worker database :

```
SQL>select  *
        from worker
    order By F_NAME  asc 0;
```

F_NAME	STATUS	GENDER	BIRTHDATE
Ajay	Regular	M	05 / 03 / 69
Ashwini	Regular	F	11 / 01 / 70
Rahul	Summer	M	01 / 12 / 72
Smita	Regular	F	23 / 09 / 67

2.8 GROUP BY CLAUSE

Another helpful clause is the group by clause. A group by clause arranges your data rows into a group according to the columns you specify.

A query that includes group by clause is called a ***grouped query*** because it groups that data from the SELECT tables and generates single summary row for each group.

The columns named in the group by clause are called the ***grouping columns.***

When GROUP BY clause is used, each item in the SELECT list must be single-valued per group.

The select clause may contain only :

- Column names
- Aggregate functions
- Constants
- An expression involving combinations of the above.

All column names in SELECT must appear in GROUP BY clause, unless the name is used only in an aggregate function. The contrary is not true; there may be column names in GROUP BY clause that do not appear in SELECT clause.

When the WHERE clause is used with GROUP BY the WHERE clause is applied first, then groups are formed from the remaining rows that satisfy the search condition.

Example :

Consider the worker table given below :

```
SQL>select  *
        from worker
                    STATUS          GENDER          BIRTHDATE
F_NAME
```

Ashwini	Regular	F	11 / 01 / 70
Rahul	Summer	M	01 / 12 / 72
Ajay	Regular	M	05 / 03 / 69
Smita	Regular	F	23 / 09 / 67

```
SQL>Select   *
      from worker
      Group By status;
```

F_NAME	STATUS	GENDER	BIRTHDATE
Ashwini	Regular	F	11 / 01 / 70
Ajay	Regular	M	05 / 03 / 69
Smita	Regular	F	23 / 09 / 67
Rahul	Summer	M	01 / 12 / 72

(2) To group by more than one column,

```
SQL>select   *
      from worker
      Group By status, Gender;
```

F_NAME	STATUS	GENDER	BIRTHDATE
Ashwini	Regular	F	11 / 01 / 70
Smita	Regular	F	23 / 09 / 67
Ajay	Regular	M	05 / 03 / 69
Rahul	Summer	M	01 / 12 / 72

2.2, 2.3,2.4,2.5,2.6, 2.7 Check Your Progress

Fill in the blanks

1) DCL contain ……&……commands.
2) Primry Key is the combination of……&…….
3) After table command operates on ……ends.
4) ……cmd is used to save data in database.
5) The condition in group by clause is given by ……clause.

2.9 HAVING CLAUSE

The Having clause is similar to the where clause. The Having clause does for aggregate data what where clause does for individual rows. The having clause is another search condition. In this case, however, the search is based on each group of grouped table.

The difference between where clause and having clause is in the way the query is processed.

In a where clause, the search condition on the row is performed before rows are grouped. In having clause, the groups are formed first and the search condition is applied to the group.

Syntax is :

```
select select_list
from table_list
[where condition [AND : OR]  …… condition]
[group by column 1, column 2, …… column N]
[Having condition]
```

Example :

```
SQL>select  *
        from worker
        Group By status, Gender
        Having Gender  =  'F';
```

F_NAME	STATUS	GENDER	BIRTHDATE
Ashwini	Regular	F	11 / 01 / 70
Smita	Regular	F	23 / 09 / 72

```
SQL>select   *
        from worker
        where Birthdate  <  11 / 01 / 70
        Group By status, Gender
        Having Gender  =  'M';
```

F_NAME	STATUS	GENDER	BIRTHDATE
Ajay	Regular	M	05 / 03 / 69

2.10 STRING OPERATION

(1) Searching for rows with the LIKE operator.

The most commonly used operation on strings is pattern matching using the operator like.

We describe patterns using two special characters.

- Percent (%) - The % character matches any substring
- Underscore (_) : The-character matches any character.

Patterns are case sensitive.

To illustrate consider the following examples :

1. "con%" matches with any string beginning with 'con'. For example : concurrent, conference.
2. "% nfi %" matches any string containing "nfi" as a substring.
   For example : confidence, confidential, confirm, confine.
3. "- - -" matches any three characters.
4. "- - - %" matches any string of at least three characters.
   Patterns are expressed in SQL using like operator.

Example Queries :

(1) Find the names of customers whose city name include "bad"

```
SQL>   select cust_name, cust_city
       from   customer
       where cust_city  like "%bad";
```

Cust_name	Cust_city
Sachin	Aurangabad
Rahul	Hyderabad
Ajay	Ahemadabad

(2) Find the student's last name and id if the last name begins with "Desh"

```
SQL>   select student_id, last_name
       from student
       where last_name like "Desh %";
```

student_id	last_name
101	Deshpande
102	Deshmukh

For patterns to include the special characters (i.e. % & –), SQL allows the specification of an escape character (\). The escape character is used immediately before a special character to indicate that the special pattern character is to be treated like a normal character. We define the escape character for a like comparison using the escape keyword. To illustrate, consider the following patterns, which use a backslash (\) as the escape character :

(1) like 'ab\%cd' escape '\'
matches all strings beginning with "ab%cd".

(2) like 'ab\\cd' escape '\'
matches all strings beginning with ab\cd.

(3) like 'ab_cd' escape '\'
matches all strings beginning with ab_cd.

SQL allows us to search for mismatches instead of matches by using the not like comparison operator.

2.11 DISTINCT ROWS

SELECT statement has an optional Keyword **distinct**. This keyword follows select and return only those rows which have distinct values for the specified columns. i.e. it eliminates duplicate values.

The keyword all allows to specify explicitly that the duplicates are not removed.

Example :

```
SQL>select distinct branch_name
        from loan;
```

which eliminates duplicate values in the result.

```
SQL>select all branch_name
        from loan;
```

it specifies that duplicates are not eliminated from result relation.

Since duplicate retention is by default, we will not use **all**.

2.12 RENAME OPERAITON

SQL provides a mechanism for renaming both relations and attributes. It uses **as** clause and the syntax is :

old_name **as** new_name

The **as** clause can appear in both the select and from clauses.

Example :

```
SQL>    select distinct customer_name, borrower_loan_no.
        from borrower, loan
      where borrower·loan_no = loan·loan_no and
      branch name =  'ICICI';
```

This query can be rewritten using as clause as follows :

```
SQL>    select customer_name, borrower_loan no as loan_id
        from   borrower, loan
        where  borrower loan_no = loan·loan_no and
        branch name  = 'ICICI';
        where borrower_loan_no attribute is renamed as
        loan_id.;
```

2.8 - 2.12 Check Your Progress

Fill in the blanks

1) A query that include group by clause is called……query.

2) Duplication of data avoid by ……Keyword.

2.13 SET OPERATIONS

The SQL-92 operations UNION, INTERSECT and MINUS operate on relations and correspond to the relational algebra operations $\cup$, $\cap$, $-$.

Like the union, intersect and set difference in relational algebra, the relations participating in the operations must be compatible, i.e. they must have the same set of attributes.

There are restrictions on the tables that can be combined using the set operations, the most important one being that the two tables have to be union-compatible; that is they have the same structure. This implies that the two tables must contain the same number of columns, and that their corresponding columns have the same data types and lengths. It is the user's responsibility to ensure that data values in corresponding columns come from the same domain.

Union operator :

The syntax for this set operator is :

```
select_statement 1
Union
select_statement 2
[ order_by_clause]
```

The variables are defined as :

select_statement 1 and select_statement 2 are valid select statements

order_by_clause is optional ORDER By clause that references the columns by number rather than by name.

The UNION operator combines the rows returned by the first SELECT statement with rows returned by the second SELECT statement.

Keep following things in mind when you use the UNION operator.

1. The two SELECT statement may not contain an ORDER By clause; however, you can order the results of the union operation.
2. The number of columns retrieved by select_statement 1 must be equal to the number of columns retrieved by select_statement 2.
3. The data types of the columns retrieved by select_statement 1 must match with the data types of the columns retrieved by select_statement 2.
4. Here the optional order_by_clause differs from the usual ORDER By clause in a select statement, because the columns used for ordering must be referenced by number rather than by name. The reason that columns must be referenced by number is that SQL does not require that the column names retrieved by select_statement-1 be identical to the column names retrieved by select statement - 2.

Example :

Find all customers having a loan, an account or both at the bank.

```
SQL> select customer_name
     from depositor
     union
     select customer_name
         from borrower.
```

Union operation finds all customer having an account, loan or both at bank.

Union operation eliminates duplicates.

Intersect Operator :

The Intersect operator returns the rows that are common between two sets of rows.

The syntax for using the INTERSECT operator is :

```
select_statement-1
Intersect
select_statement-2
[Order_By_clause]
```

The variables are defined as follows :

Select_statement 1 and select_statement 2 are valid SELECT statements.

Order_By clause is an optional Order By clause that references the columns by number rather than by name.

Here are some requirements and considerations for using the INTERSECT operator.

1. The two select statement may not contain Order_By clause; however, you can order the results of the entire **Intersect** operation.
2. The number of columns retrieved by select_statement 1 must be equal to the number of columns retrieved by select_statement 2.
3. The data types of columns retrieved by select_statement 1 must match the data types of the columns retrieved by select_statement 2.
4. The optional Order_By_clause differs from the usual Order By clause in the SELECT statement because the columns used for ordering must be referenced by number rather than by name. The reason that the columns in the Order_By_clause must be referenced by number rather than by name is that SQL does not require that the column names retrieved by select_statement 1 be identical to column names retrieved by select-statement 2. Therefore, you must indicate the columns to be used in ordering results by their position in select list.

Example :

Find all customers who have both an account and loan at the bank.

```
SQL>  (select customer_name
       from depositor)
       INTERSECT
       (select customer_name
       from borrower)
```

The intersect operator automatically eliminates duplicates. If we want to retain all duplicates, we must write INTERSECT all in place of INTERSECT.

The Minus Operator (Except operator) :

The syntax for using Minus operator is :

```
select_statement 1
Minus
select_statement 2
[order by clause]
```

The variables defined are :

select_statement 1 and select_statement 2 are valid SELECT statements.

Order_By_clause is an ORDER By Clause that references columns by numbers rather than by name.

The requirements and considerations for using the MINUS operator are essentially the same as those for the INTERSECT and UNION operator.

Example : Find all customers who have an account but no loan at the bank.

```
SQL>  Select customer_name
      from depositor
      MINUS
      Select customer_name
      from borrower
```

2.14 AGGREGATE FUNCTIONS

Aggregate functions are the functions that take a collection of values as input and return a single value.

SQL offers five built-in aggregate functions.

1. Average : AVG
2. Minimum : MIN
3. Maximum : MAX
4. Total : SUM
5. Count : COUNT

These functions operate on a single column of a table and return a single value.

COUNT, MIN and MAX apply to both numeric and non-numeric fields, but SUM and AVG may be used on numeric fields only.

Apart from COUNT(*), each function eliminates nulls first and operates only on the remaining non-null values.

If we want to eliminate duplicates before the function is applied, we use the keyword DISTINCT before the column name in the function.

The keyword ALL can be used if we do not want to eliminate the duplicates. ALL is assumed if nothing is specified.

DISTINCT has no effect on MIN and MAX functions. It may effect on the result of SUM or AVG.

It is important to note that an aggregate function can be used only in SELECT list and in the HAVING clause. It is incorrect to use it elsewhere.

avg function :

avg function computes the column's average value.

The input to *avg* must be a collection of numbers.

Example : Find the average balance

```
SQL>  select avg (balance)
      from account;
```

This aggregate function can also be applied to a group of set of tuples using group by clause.

Example : Find the average balance at each branch

```
SQL>  select branch_name, avg (balance)
      from account
      group by branch_name;
```

min and max functions :

min and *max* return the minimum and maximum values for the specified column.

Example :

Find the minimum and maximum values of balance.

Select *max* (balance) *min* (balance) from account.

sum function :

sum function computes the column's total value. Input to this function must be a collection of numbers.

Count function :

count function counts the number of rows. There are two forms of count.

count (*) - which counts all the rows in a table that satisfy any specified criteria.

count (column_name) - which counts all rows in a table that have a non-null value for column_name and satisfy the specified criteria.

NULL Values :

SQL allows the use of null values to indicate absence of information about the value of an attribute.

We can use the special keyword NULL in a predicate to test for a null value.

Example :

```
SQL>  select loan_no
      from loan
      where amount is NULL;
```

The predicate NOT NULL tests for the absence of null values.

The use of a NULL value in arithmetic and comparison operations causes several complications. The result of an arithmetic expressions is NULL if any of the input values is NULL. The result of any comparison involving a NULL value can be thought of as being false.

SQL_92 treats the results of such comparisons as unknown, which is neither true nor false. It also allows us to test whether the result of a comparison is unknown.

In general, aggregate functions treat nulls using the following rule :

All aggregate functions except count (*) ignore NULL values in their input collection.

2.15 NESTED SUB QUERIES

SQL provides a mechanism for the nesting of sub queries. A sub query is a select-from-where expression that is nested within another query. A common use of sub queries is to perform tests for :

1. Set membership
2. Set comparison
3. Set cardinality.

1. Set Membership : (in connective)

The *in* connective tests for the set membership, where the set is a collection of values produced by a select clause.

The *not in* connective tests for the absence of set membership.

As an illustration consider the following query :

(1) "Find all customers who have both a loan and an account at the bank".

Note : The result of this query can be obtained using INTERSECT operator.

```
SQL>  select customer_name
      from borrower
      where customer_name in (select customer_name from depositor);
```

i.e. find all customers having an account who are members of the set of borrowers from the bank.

(2) Find all customers who have both an account and loan at the ICICI branch.

```
SQL>  select customer_name
      from borrower, loan
      where borrower loan no = loan · loan_no and
            branch_name = 'ICICI' and
      (branch_name, customer_name) in
      (select branch_name, customer_name
          from depositor, account
          where depositor·account_no = account·account_no);
```

Example query for not in connective :

(1) Find all customers who do have a loan at the bank, but do not have an account at the bank.

```
SQL>  select customer_name
      from borrower
      where customer_name not in
          (select customer_name
          from depositor);
```

The **in** and **not** in operators can also be used on enumerated sets.

Example :

Find the customer names who have a loan at a bank and whose names are neither 'Sachin' nor 'Ajay'.

```
SQL>  select customer_name
          from borrower
```

where customer_name **not in** ('Sachin', 'Ajay');

2. Set Comparison :

SQL allows following set comparison operators :

< some	:	Less than at least one
<= some	:	Less than or equal to at least one
> some	:	Greater than at least one
>= some	:	Greater than or equal to at least one
= some	:	Equal to at least one
< > some	:	Not equal to at least one.

Example Query :

"Find the names of all branches that have assets greater than those of at least one branch located in Bombay**"**

```
SQL>  select branch_name
      from branch
      where assets > some (select assets
            from branch
            where branch_city = 'Bombay')
   Sub query(select assets
         from branch
            where branch city = Bombay)
```

generates the set of all asset values for all branches in Bombay. The > some comparison in where clause of the outer select is true if the asset value of the tuple is greater than at least one member of the set of all asset values for branches in Bombay.

SQL also supports following set of comparison operators :

< all	:	less than all
<= all	:	less than or equal to all
> all	:	greater than all
>= all	:	greater than or equal to all
= all	:	equal to all
< > all	:	not equal to all

Example Query :

Find the branch that has the highest average balance.

```
SQL>  select branch_name from account
group by branch_name having avg (balance) >= all (select avg (balance) from
account group by branch_name);
```

Test for Empty Relations :

SQL includes a feature for testing whether a sub query has any tuples in its results.

The **exists** construct returns the value true if the argument query is non-empty.

Similarly, we can test the non-existence of tuples in a sub-query by using the **not-exists** construct.

Example Query using exists construct :

"Find all customers who have both an account and a loan at the bank.**"**

```
SQL>select customer_name
      from borrower
      where exists (select  *
            from depositor
            where depositor customer_name =
            borrower·customer_name);
```

Example Query using Not exists construct :

Find all customers who have an account at all branches located in Bombay.

Note : For each customer we need to see whether the set of all branches at which that customer has an account contains the set of all branches in Bombay.

```
SQL>  select distinct customer_name
      from depositor as S
      where not exists (select branch_name
            from branch
            where branch_city = 'Bombay')
                        minus
            (select R·branch_name
            from depositor as T, account as R
            where,
T.account_number= R·account_number
                  and
  S·customer_name = T·customer_name)
  where,
            (select branch_name
            from branch
            where branch_city  =  'Bombay')
```

Finds all the branches in Bombay.

The sub query

```
(select R·branch_name
from depositor as T, account as R
where T·account_number = R·account_number
and S·customer_name  =  T·customer_name)
```

Finds all branches at which customer S·customer_name has an account.

Thus, the outer select takes each customer and tests whether the set of all branches at which the customer has an account contains the set of all branches located in Bombay.

Test for the Absence of Duplicate Tuples :

SQL includes a feature for testing whether a sub query has any duplicate tuples in its result.

The **unique** construct returns the value true if the argument sub query contains no duplicate tuples.

Example Query :

Find all customers who have only one account at ICICI branch.

```
SQL >  select T·customer_name
       from depositor as T
       where unique (select R·customer_name
       from account, depositor as R
       where  T·customer_name
              =  R·customer_name and
       R·account_no =  account·account_number
       and
       account·branch_name  =  'ICICI');
```

We can test for the existence of duplicates in a sub-query by using the **not unique** construct.

Example Query :

Find all customers who have at least two accounts at the ICICI branch.

```
SQL>  select distinct T·customer_name
      from depositor T
      where not unique (select R·customer_name
      from account, depositor as R
      where T·customer_name =  R·customer_name
      and
```

R·account_number = account·account_number
and account·branch_name = 'ICICI');

2.16 EMBEDDED SQL

Need of embedded SQL : SQL provides a powerful declarative query language. Writing queries in SQL is typically much easier than is coding the same queries in a general-purpose programming language. However, access to a database from a general purpose language is required for at least two reasons :

(1) Not all queries can be expressed in SQL since, SQL does not provide the full expressive power of a general purpose language. That is there exist queries that can be expressed in a language such as Pascal, C, Cobol, or Fortran that cannot be expressed in SQL. To write such queries, we can embed SQL within a more powerful language.

SQL is designed such that queries written in it can be optimized automatically and executed efficiently, and providing the full power of a programming language makes automatic optimization exceedingly difficult.

(2) Non-declarative actions such as printing a report, interacting with a user, or sending the results of a query to a graphical user interface, cannot be done from within SQL. Applications typically have several components and querying or updating data is only one component, other components are written in general purpose programming languages. For an integrated application, the programs written in the programming language must be able to access the database.

The SQL standard defines embedding of SQL in a variety of programming languages, such as Pascal, PL/I, C, and control.

A language in which SQL queries are embedded is referred to as a host language, and the SQL structures permitted in the host language constitute embedded SQL.

Programs written in host language can use the embedded SQL syntax to access and update data stored in a database. This form of SQL extends the programmer's ability to manipulate the database even further.

Working of Embedded SQL :

In embedded SQL all query processing is performed by the database system. The result of query is then made available to the program one tuple at a time. An embedded SQL program must be processed by a special preprocessor prior to compilation. Embedded SQL requests are replaced with host language declarations and procedure calls that allow run-time execution of the database accesses. Then the resulting program is compiled by the host language compiler.

Syntax of Embedded SQL :

To identify embedded SQL request to the preprocessor we use EXEC SQL statement.

The format is :

EXEC SQL < embedded SQL statement > END EXEC.

The exact syntax for embedded SQL requests depends on the language in which SQL is embedded. For example, a semi-colon is used instead of END-EXEC when SQL is embedded in C or Pascal.

We place the statement SQL INCLUDE in the program to identify the place where preprocessor should insert the special variables used for communication between the program and database system.

Variables of the host language can be used within embedded SQL statements, but they must be preceded by a colon (:) to distinguish them from SQL variables.

To write a query, we use *declare cursor* statement.

Example :

Consider the banking schema, we have host language and variable *amount*. The query is to find the names and cities of residence of customers who have more than amount dollars in any account.

```
EXEC SQL
declare c cursor for
select customer_name, customer_city
from depositor, customer
where depositor·customer_name =  customer.customer_name
    and
    depositor·balance > : amount
END EXEC.
```

The variable c in the example is called cursor for the query. This variable is used to identify the query in open and fetch statements.

Open statement : Open statement causes the query to be evaluated.

The open statement for the above given query is :

```
EXEC SQL open c END-EXEC
```

It causes the database system to evaluate the query and stores results within a temporary relation. If SQL query results in an error, the database system stores an error diagnostic in the SQL communication area (SQLCA) variables, whose declarations are inserted by SQL INCLUDE statement.

Fetch statement : A fetch statement causes the values of one tuple be placed in host language variables. A series of fetch statements is executed to make the results available to program. The fetch statement requires one host-language variable for each attribute of the result relation.

For our example, consider that customer_name is stored in cn and customer city in cc.

```
EXEC SQL fetch c into : cn : cc END EXEC :
```

One fetch statement return only one tuple. To obtain all tuples of the result, the program must contain a loop to iterate overall tuples. Embedded SQL assists the programmer in managing this iteration. In a relation, tuples of the result of a query are in some fixed physical order. When an open statement is executed, the cursor is set to point to the first tuple of result. When fetch is executed, the cursor is updated to point to the next tuple of the result. A variable in SQLCA is set to indicate that no further tuples remain to be processed. Thus we can use while loop to process each of the tuples.

Close statement : A close statement must be used to tell the database system to delete the temporary relation that held the result of the query.

For our example, the close statement is

```
EXEC SQL close c END EXEC
```

Embedded SQL expression for database modification can be given as :

```
EXEC SQL  <  any valid update, insert
                or delete  >  END  EXEC
```

Host language variables, preceded by a colon, may appear in SQL database modification expression. If an error arises in the execution of the statement, a diagnostic is set in the SQLCA.

2.13-2.16 Check Your Progress

Fill in the blanks

1) .....Functions is used to calculate the average.

2) Query under Query is called as.....

3) .....statements causes the query to be evaluated.

4) ..... allows program to construct & submit SQL Querries at run time.

2.17 DYNAMIC SQL

Dynamic SQL component of SQL - 92 allows programs to construct and submit SQL queries at run-time. Using dynamic SQL programs can create SQL queries as string s at run time and can execute them immediately or prepare them for subsequent use. Preparing a dynamic SQL statement compiles it, and subsequent uses of the prepared statement use the compiled version.

Example :

```
char * sqlprog  = "Update account set
            balance = balance * 1.05
    where account_no  =  ?"
EXEC SQL prepare dynprog from : sqlprog;
char account -[10] = "A = 101";
EXEC SQL execute dynprog using : account;
```

The dynamic SQL program contains a ? which is a place holder for a value that is provided when the SQL program is executed.

EXAMPLE QUERIES

(I) Consider the following database

Employee (emp_no, name, skill, pay_rate)

Position (posting_no., skill)

Duty_allocation (posting_no., emp_no, day, shift)

Find SQL queries for the following :

(1) Get complete details from Duty_allocation

```
select *
from Duty_allocation;
```

(2) Get duty allocation details for Emp_no 123461 for the month of April 1986.

```
select posting_no., shift, day
from Duty_allocation
where emp_no  =  123461 and
    Day ≥ 19860401 and Day ≤ 19860430 ;
```

(3) Find the shift details for employee 'XYZ' :

```
select posting_no., shift, day
from Duty_allocation, Employee
where Duty allocation.emp_no. = Employee.emp_no and
    Name  =  'XYZ';
```

(4) Get employees whose rate of pay is more than or equal to the rate of pay of employee 'XYZ'

```
select S.name, S.pay_rate
from Employee as S, Employee as T
where S.pay_rate >  T.pay_rate
and T.name  =  'XYZ';
```

(5) Compile all pairs of posting_nos requiring the same skill

```
select S.posting_no., T.posting_no.
from Position S, Position T
where  S.skill = T.skill
and  S.posting_no. <  T.posting_no.;
```

(6) Find the employees eligible to fill a position.

```
select Employee.emp_no., position.posting_no., position.skill
from Employee, Position
where employee·skill =  position.skill;
```

(7) Get the names and pay rates of employees with emp_no less than 123460 whose rate of pay is more than the rate of pay of at least one employee with emp_no greater than or equal to 123460.

```
    select name, pay_rate
```

```
from Employee
where emp_no < 123460 and
      pay_rate > some
      (select pay_rate
         from Employee
         where emp_no ≥ 123460);
```

(8) Get employees who are working either on the date 19860419 or 19860420.

```
select emp_no
from Duty_allocation
where Day in (19860419, 19860420);
                    OR
select emp_no
from Duty_allocation
where Day = 19860419 or Day = 19860420.
```

(9) Find the names of all employees who are assigned to all positions that require a Chef's skill.

```
select S.Name
from Employee S
where
    (select posting_no
    from Duty_allocation D
    where S.emp_no = D.emp_no)
    contains
    (select P.posting_no
    from position P
    where P.skill = 'Chef');
```

(10) Find the employees with the lowest pay rate

```
select emp_no, Name, Pay_rate
from Employee
where pay_rate ≤ all
    (select pay_rate
    from Employee)
```

(11) Get the names of Chef's paid at the minimum Pay-Rate.

```
select name
from Employee
where skill = 'Chef' and
    pay_Rate ≤ all
    (select pay_rate
    from Employee
    where skill = 'Chef')
```

(12) Find the names and the rate of pay of all employees who are allocated a duty.

```
select name, pay_rate
from Employee
where EXISTS
(select *
from Duty_allocation
where Employee.emp_no = Duty_allocation.emp_no)
```

(13) Find the names and the rate of pay of all employees who are not allocated a duty.

```
select name, pay_rate
from Employee
where NOT EXISTS
(select *
from Duty_allocation
where Employee.emp_no
    = Duty_allocation.emp_no)
```

(14) Get employees who are waiters or work at Posting-no 321

```
(select emp_no
from Employee
where skill = 'waiter')
Union
(select emp_no
from Duty_allocation
where posting_no = 321)
```

(15) Get employee numbers of persons who work at posting-no 321 but don't have the skill of waiter.

```
(select emp_no
from Duty_allocation
where posting_no = 321)
minus
(select emp_no
from Employee
where skill 'waiter')
```

(16) Get a list of employees not assigned a duty

```
(select emp_no
from Employee)
minus
(select emp_no
from Duty_allocation)
```

(17) Get a list of names of employees with the skill of Chef who are assigned a duty

```
select Name
from Employee
where emp_no in
((select emp_no
from Employee
where skill = 'Chef')
intersect
(select emp_no
from Duty_allocation));
```

(18) Get a count of different employees on each shift

```
select shift, count (distinct emp_no)
from Duty_allocation
group by shift;
```

(19) Get the employee numbers of all employees working on at least two dates.

```
select emp_no
from Duty_allocation
group by emp_no
having (count;*) > 1
```

(II) Consider the given database :

Project (project_id, proj_name, chief_arch)
Employee (Emp_id, Emp_name)
Assigned_To (Project_id, emp_id)
Find the SQL queries for the following statements :

(1) Get employee number of employees working on project C353

```
select emp_id
from Assigned_To
where projectid = 'C353';
```

(2) Get details of employees working on project C 353.

```
select A.empid, emp_name
from A.Assigned_To A, Employee
where project_id = 'C353' ;
```

(3) Obtain details of employees working on Database project

```
select Emp_name, A. Emp_id
from A. Assigned_To A, Employee
where project_id in (select P. project_id
        from P. project
        where P. project_name = 'Database');
```

(4) Get details of employees working on both C353 and C354.

```
(select Emp_name, A. emp_id
from Assigned_to A, Employee
where A.Project_id  =  C354)
intersect
(select emp_name, A.empid
from A.Assigned_To A, Employee
where project_id = 'C354');
```

(5) Get employee numbers of employees who do not work on project C 453

```
(select emp_id
from Employee)
minus
(select emp_id
from assigned_to
where project_id = 'C453');
```

(6) Get the employee numbers of employees who work on all projects.

```
select emp_id
from assigned to
where project_id = all
    (select project_id
    from project);
```

(7) Get employee numbers of employees who work on at least all those projects that employee 107 works on

```
((select emp_id
    from Assigned_To
    where project_id  =  all
     (select project_id
        from Assigned_To
        where emp_id = 107))
    minus107);
```

(8) Get employee numbers who work on at least one project that employee 107 works on.

```
((select emp_id
    from Assigned_To
    where project_id in
     (select project_id
     from Assigned to
     where emp_id = 107)
     minus 107);
```

(III) Consider the employee database :
employee (employee_name, street, city)
works (employee_name, company_name, salary)

company (company_name, city)
manages (employee_name, manager_name).

Give an expression in SQL for each of the following :

(1) Find the names of all employees who work for FBC.

```
select employee_name
from works
where company_name = 'FBC' ;
```

(2) Find the names and cities of all employees who work for FBC.

```
select employee·employee_name, city
from works, employee
where employee·employee_name = works · employee_name and
company_name = 'FBC';
```

(3) Find the names, street address, and cities of residence of all employees who work for FBC and earn more than $ 10,000.

```
select employee·employee_name, street, city
from works employee
where employee·employee_name = works·employee_name and
company_name ='FBC' and salary > 10000;
```

(4) Find all employees in the database who live in the same cities as the companies for which they work.

```
select w.employee_name
from works w, emple, comp c
where e.emp_name = w.emp_name and
C.company_name . w. company_name and e.city = city;
```

(5) Find all employees in the database who live in the same cities and on the same street as do their managers.

```
select E.employee_name
from employee E.employee T, manages
where E.employee_name= manages.employee_name
and E.street = T.street and E.city = T.city and
T.employee_name = manages.manager_name;
```

(6) Find all employees in the database who do not work for FBC.

```
(select employee_name
from employee)
minus
(select employee_name
from works
where company_name = 'FBC');
```

(7) Find all employees in the database who earn more than every employee of small bank corporation

```
select employee_name
from works
where salary > (select max (salary)
from works
where company_name = 'FBC');
```

(8) Find all employees who earn more than the average salary of all employees of their company.

```
select T.employee_name
from works T.
where salary > (select avg (S.salary)
from works S.
where T.company_name = S.company_name);
```

(9) Find the company that has the smallest payroll

```
SQL> create view payroll (compname, smallpay)
     as
     select company_name, min (salary)
     from works
     group by company name;
SQL> select company name
     from payroll
     where small_pay = (select min (small_pay) from payroll);
```

(10) Find those companies whose employees earn a higher salary, on average than the average salary at FBC

```
SQL> create view avg_salary (comp_name, av_sal)
     as
     select company_name, avg (salary)
     from works
     group by company_name
SQL> select T.comp_name
     from avg_salary T. avg_salary S.
     where S.company_name = 'FBC'
     and T.av_sal > S.av_sal;
```

(11) Find the company that has must employees

```
SQL> create view no_emp (compname, no_employee)
     as
     select company_name, count (employee_name)
     from works
     group by company_name;
SQL> select company_name
     from no_emp
     where no_employee = (select max no_emplyee)
     from no_emp)
```

2.18 SUMMARY

SQL is devided into three groups of command DDL (Data definition language),DML(Data manipulation language) DCL (Data Control language). DDL related with the structure of the objects. It has create table, alter table, drop table, create view & create index commands. DML is related with the data in the table

2.19 CHECK YOUR PROGRESS - *ANSWERS*

2.2-2.7

1)Grant & Revoke
2) Unique & Not Null
3) DDL
4)Commit
5) Having

2.8-2.12

1)Grouped
2)Distinct or Unique

2.13-2.16

1) Avg ()
2) Sub Query/Nested Query
3)Open
4) Dynamic SQL

2.20 QUESTIONS FOR SELF-STUDY

Q.1 Define the following terms :
- (i) DDL
- (ii) DML

Q.2 What are the data types in SQL ?

Q.3 Give syntax of following SQL commands :
- (i) CREATE
- (ii) ALTER
- (iii) DROP
- (iv) INSERT
- (v) DELETE
- (vi) UPDATE
- (vii) SELECT

Q.4 What are subdivisions of SQL ?

Q.5 What are the set operations of SQL-92 ? Explain with examples.

Q.6 Write a note on :
- (i) Nested sub queries
- (ii) Views in SQL
- (iii) Indexes in SQL
- (iv) DCL
- (v) Embedded SQL
- (vi) Dynamic SQL.

Q.7 Consider the insurance database :

Person(driver_id, name, address)
Car(license, model, year)
Accident(report_no, data, location)
Owns(driver_id, license)
Participated(driver_id, report_no, damage_amount)
Give an expression in SQL for each of the following :

1. Find the total number of people who owned cars that were involved in accident in 1989.
2. Find the total number of accidents in which car belonging to John Smith is involved
3. Add a new accident to the database
4. Delete the Mazda belonging to John Smith.

Q.8 Consider the schema for Presidential database

President(pres_id, last_name, first_name, political_party, state_from)
Administration(start_data, pre_id, end_data, VP_last_name, VP_first_name)
State(state_name, data_admitted, area, population, capital_city)
Write SQL queries.

Q.9 Consider the relation schemas

customer(customer_name, customer_street, customer_city) account (branch_name, account_no, balance)
Depositor(customer_name, account_no)
Give an expression in SQL for following query :
Find the average balance for each customer who lives in Harison and has at least three accounts.

Q.10 Consider the following tables :

Frequents(visitor, stall)

Servers(stall, icecream)

Likes(visitor, icecream)

Write the following queries in SQL.

1. Print the stalls that serve the ice cream that visitor john likes.
2. Print the visitors that frequently visit at least one stall that serves the ice cream they like.

2.21 SUGGESTED READINGS

Teach Yourself SQL in 21 Days **-** By Ryan K. Stephens Ronald R Plew

Using Oracle Application - By Jim Crum

NOTES

NOTES

Chapter 3

QUERY MULTIPLE TABLES

3.0 Objectives
3.1 Introduction
3.2 Joins
3.2.1 Equi-Join.
3.2.2 Non-Equi-Join.
3.2.3 Outer Join versus Inner Join
3.2.4 Joining Table to Itself.
3.3 Procedures and Functions
3.4 Creating a Procedure
3.5 Executing a Procedure
3.6 Deleting a Procedure
3.7 Functions
3.7.1 Aggregate Functions
3.7.2 Date & Time Function
3.7.3 Arithmatic Functions
3.7.4 Character Functions
3.7.5 Conversion Functions
3.7.6 Miscelleneous Functions
3.8 Summary
3.9 Check Your Progress - *Answers*
3.10 Questions for Self – Study
3.11 Suggested Readings

3.0 OBJECTIVES

After reading this chapter you will able to

- explain how to Creating procedure
- explain how to Executing procedure
- explain how to Deleting procedure
- describe Function

3.1 INTRODUCTION

Today you will learn about joins. This information will enable you to gather and manipulate data across several tables. By the end of the day, you will understand and be able to do the following :

- Perform an outer join
- Perform a left join
- Perform a right join
- Perform an equi-join
- Perform a non-equi-join
- Join a table to itself.

3.2 JOINS

One of the most powerful features of SQL is its capability to gather and manipulate data from across several tables. Without this feature you would have to store all the data elements necessary for each application in one table. Without common tables you would need to store the same data in several tables. Imagine having to redesign, rebuild, and repopulate your tables and databases every time your user needed a query with a new piece of information. The JOIN statement of SQL enables you to design smaller, more specific tables that are easier to maintain than larger tables.

Multiple Tables in a Single SELECT Statement

Like Dorothy in The Wizard of Oz, you have had the power to join tables since Day 2, "Introduction to the Query : The SELECT Statement," when you learned about SELECT and FROM. Unlike Dorothy, you do not have to click you heels together three times to perform a join. Use the following two tables, named, cleverly enough, TABLE1 and TABLE2.

INPUT :
SELECT *
FROM TABLE1

OUTPUT :

ROW	REMARKS
=======	=======
row 1	Table 1
row 2	Table 1
row 3	Table 1
row 4	Table 1
row 5	Table 1
row 6	Table 1

INPUT :
SELECT *
FROM TABLE2
OUTPUT :

ROW	REMARKS
=========	========
row 1	table 2
row 2	table 2
row 3	table 2
row 4	table 2
row 5	table 2
row 6	table 2

To join these two tables, type this :

INPUT :
SELECT *
FROM TABLE1, TABLE2
OUTPUT :

ROW	REMARKS	ROW	REMARKS
row 1	Table 1	row 1	table 2
row 1	Table 1	row 2	table 2
row 1	Table 1	row 3	table 2
row 1	Table 1	row 4	table 2
row 1	Table 1	row 5	table 2
row 1	Table 1	row 6	table 2
row 2	Table 1	row 1	table 2
row 2	Table 1	row 2	table 2
row 2	Table 1	row 3	table 2
row 2	Table 1	row 4	table 2
row 2	Table 1	row 5	table 2
row 2	Table 1	row 6	table 2
row 3	Table 1	row 1	table 2
row 3	Table 1	row 2	table 2
row 3	Table 1	row 3	table 2
row 3	Table 1	row 4	table 2
row 3	Table 1	row 5	table 2
row 3	Table 1	row 6	table 2
row 4	Table 1	row 1	table 2
row 4	Table 1	row 2	table 2
row 4	Table 1	row 3	table 2
row 4	Table 1	row 4	table 2
row 4	Table 1	row 5	table 2
row 4	Table 1	row 6	table 2
row 5	Table 1	row 1	table 2
row 5	Table 1	row 2	table 2
row 5	Table 1	row 3	table 2
row 5	Table 1	row 4	table 2
row 5	Table 1	row 5	table 2
row 5	Table 1	row 6	table 2
row 6	Table 1	row 1	table 2
row 6	Table 1	row 2	table 2
row 6	Table 1	row 3	table 2
row 6	Table 1	row 4	table 2
row 6	Table 1	row 5	table 2
row 6	Table 1	row 6	table 2

Thirty-six rows! Where did they come from ? And what kind of join is this ?

A close examination of the result of the first join shows that each row from TABLE1 was added to each row from TABLE2. An extract from this join shows what happened :

OUTPUT :

ROW	REMARKS	ROW	REMARKS
row 1	Table 1	row 1	table 2
row 1	Table 1	row 2	table 2
row 1	Table 1	row 3	table 2
row 1	Table 1	row 4	table 2
row 1	Table 1	row 5	table 2
row 1	Table 1	row 6	table 2

Notice how each row in TABLE2 was combined with row 1 in TABLE1. Congratulations! You have performed your first join. But what kind of join? An inner join? an outer join? or what? Well, actually this type of join is called a cross-join. A cross-join is not normally as useful as the other joins covered today, but this join does illustrate the basic combining property of all joins : Joins bring tables together.

Suppose you sold parts to bike shops for a living. When you designed your database, you built one big table with all the pertinent columns. Every time you had a new requirement, you added a new column or started a new table with all the old data plus the new data required to create a specific query. Eventually, your database would collapse from its own weight-not a pretty sight. An alternative design, based on a relational model, would have you put all related data into one table. Here's how your customer table would look :

INPUT :
SELECT *
FROM CUSTOMER
OUTPUT :

NAME	ADDRESS	STATE	ZIP	PHONE	REMARKS
========	=========	======	====	=======	========
TRUE WHEEL	55O HUSKER	NE	58702	555-4545	NONE
BIKE SPEC	CPT SHRIVE	LA	45678	555-1234	NONE
LE SHOPPE	HOMETOWN	KS	54678	555-1278	NONE
AAA BIKE	10 OLDTOWN	NE	56784	555-3421	JOHN-MGR
JACKS BIKE	24 EGLIN	FL	34567	555-2314	NONE

Finding the Correct Column

When you joined TABLE1 and TABLE2, you used SELECT *, which returned all the columns in both tables. In joining ORDERS to PART, the SELECT statement is a bit more complicated :

SELECT O.ORDEREDON, O.NAME, O.PARTNUM,
P.PARTNUM, P.DESCRIPTION

SQL is smart enough to know that ORDEREDON and NAME exist only in ORDERS and that DESCRIPTION exists only in PART, but what about PARTNUM, which exists in both? If you have a column that has the same name in two tables, you must use an alias in your SELECT clause to specify which column you want to display. A common technique is to assign a single character to each table, as you did in the FROM clause :

FROM ORDERS O, PART P

You use that character with each column name, as you did in the preceding SELECT clause. The SELECT clause could also be written like this :

SELECT ORDEREDON, NAME, O.PARTNUM, P.PARTNUM, DESCRIPTION

But remember, someday you might have to come back and maintain this query. It does not hurt to make it more readable. Now back to the missing statement.

3.2.1 Equi-Joins

An extract from the PART/ORDERS join provides a clue as to what is missing :

30-JUN-1996 TRUE WHEEL 42 54 PEDALS
30-JUN-1996 BIKE SPEC 54 54 PEDALS
30-MAY-1996 BIKE SPEC 10 54 PEDALS

Notice the PARTNUM fields that are common to both tables. What if you wrote the following ?

INPUT :
SELECT O.ORDEREDON, O.NAME, O.PARTNUM,
P.PARTNUM, P.DESCRIPTION

FROM ORDERS O, PART P
WHERE O.PARTNUM = P.PARTNUM
OUTPUT :

ORDEREDON	NAME	PARTNUM	PARTNUM	DESCRIPTION
1-JUN-1996	AAA BIKE	10	10	TANDEM
30-MAY-1996	BIKE SPEC	10	10	TANDEM
2-SEP-1996	TRUE WHEEL	10	10	TANDEM
1-JUN-1996	LE SHOPPE	10	10	TANDEM
30-MAY-1996	BIKE SPEC	23	23	MOUNTAIN BIKE
15-MAY-1996	TRUE WHEEL	23	23	MOUNTAIN BIKE
30-JUN-1996	TRUE WHEEL	42	42	SEATS
1-JUL-1996	AAA BIKE	46	46	TIRES
30-JUN-1996	BIKE SPEC	54	54	PEDALS
1-JUL-1996	AAA BIKE	76	76	ROAD BIKE
17-JAN-1996	BIKE SPEC	76	76	ROAD BIKE
19-MAY-1996	TRUE WHEEL	76	76	ROAD BIKE
11-JUL-1996	JACKS BIKE	76	76	ROAD BIKE
17-JAN-1996	LE SHOPPE	76	76	ROADBIKE

Using the column PARTNUM that exists in both of the preceding tables, you have just combined the information you had stored in the ORDERS table with information from the PART table to show a description of the parts the bike shops have ordered from you. The join that was used is called an equi-join because the goal is to match the values of a column in one table to the corresponding values in the second table.

You can further qualify this query by adding more conditions in the WHERE clause. For example:

INPUT/OUTPUT :
SELECT O.ORDEREDON, O.NAME, O.PARTNUM,
P.PARTNUM, P.DESCRIPTION
FROM ORDERS O, PART P
WHERE O.PARTNUM = P.PARTNUM
AND O.PARTNUM = 76

ORDEREDON	NAME	PARTNUM	PARTNUM	DESCRIPTION
1-JUL-1996	AAA BIKE	76	76	ROAD BIKE
17-JAN-1996	BIKE SPEC	76	76	ROAD BIKE
19-MAY-1996	RUE WHEEL	76	76	ROAD BIKE
11-JUL-1996	JACKS BIKE	76	76	ROAD BIKE
17-JAN-1996	LE SHOPPE	76	76	ROAD BIKE

The number 76 is not very descriptive, and you would not want your sales people to have to memorize a part number. (We have had the misfortune to see many data information systems in the field that require the end user to know some obscure code for something that had a perfectly good name. Please don't write one of those!) Here's another way to write the query :

INPUT/OUTPUT :
SELECT O.ORDEREDON, O.NAME, O.PARTNUM,
P.PARTNUM, P.DESCRIPTION
FROM ORDERS O, PART P

```
WHERE O.PARTNUM = P.PARTNUM
AND P.DESCRIPTION = 'ROAD BIKE'
```

ORDEREDON	NAME	PARTNUM	PARTNUM	DESCRIPTION
1-JUL-1996	AAA BIKE	76	76	ROAD BIKE
17-JAN-1996	BIKE SPEC	76	76	ROAD BIKE
19-MAY-1996	TRUE WHEEL	76	76	ROAD BIKE
11-JUL-1996	JACKS BIKE	76	76	ROAD BIKE
17-JAN-1996	LE SHOPPE	76	76	ROAD BIKE

Along the same line, take a look at two more tables to see how they can be joined. In this example the employee_id column should obviously be unique. You could have employees with the same name, they could work in the same department, and earn the same salary. However, each employee would have his or her own employee_id. To join these two tables, you would use the employee_id column.

EMPLOYEE_TABLE	EMPLOYEE_PAY_TABLE
employee_id	employee_id
last_name	salary
first_name	department
middle_name	supervisor
	marital_status

```
INPUT :
SELECT   E.EMPLOYEE_ID, E.LAST_NAME, EP.SALARY
FROM  EMPLOYEE_TBL E,
         EMPLOYEE_PAY_TBL EP
WHERE    E.EMPLOYEE_ID = EP.EMPLOYEE_ID
 AND      E.LAST_NAME = 'SMITH';
```

OUTPUT :

E.EMPLOYEE_ID	E.LAST_NAME	EP.SALARY
13245	SMITH	35000.00

Back to the original tables. Now you are ready to use all this information about joins to do something really useful: finding out how much money you have made from selling road bikes :

```
INPUT/OUTPUT :
SELECT SUM(O.QUANTITY * P.PRICE) TOTAL
FROM ORDERS O, PART P
WHERE O.PARTNUM = P.PARTNUM
AND P.DESCRIPTION = 'ROAD BIKE'
```

TOTAL
19610.00

With this setup, the sales people can keep the ORDERS table updated, the production department can keep the PART table current, and you can find your bottom line without redesigning your database.

Can you join more than one table? For example, to generate information to send out an invoice, you could type this statement:

INPUT/OUTPUT :
```
SELECT C.NAME, C.ADDRESS, (O.QUANTITY * P.PRICE) TOTAL
FROM ORDER O, PART P, CUSTOMER C
WHERE O.PARTNUM = P.PARTNUM
AND O.NAME = C.NAME
```

NAME	ADDRESS	TOTAL
TRUE WHEEL	55O HUSKER	1200.00
BIKE SPEC	CPT SHRIVE	2400.00
LE SHOPPE	HOMETOWN	3600.00
AAA BIKE	10 OLDTOWN	1200.00
TRUE WHEEL	55O HUSKER	2102.70
BIKE SPEC	CPT SHRIVE	2803.60
TRUE WHEEL	55O HUSKER	196.00
AAA BIKE	10 OLDTOWN	213.50
BIKE SPEC	CPT SHRIVE	542.50
TRUE WHEEL	55O HUSKER	1590.00
BIKE SPEC	CPT SHRIVE	5830.00
JACKS BIKE	24 EGLIN	7420.00
LE SHOPPE	HOMETOWN	2650.00
AAA BIKE	10 OLDTOWN	2120.00

You could make the output more readable by writing the statement like this :

INPUT/OUTPUT :
```
SELECT C.NAME, C.ADDRESS,
O.QUANTITY * P.PRICE TOTAL
FROM ORDERS O, PART P, CUSTOMER C
WHERE O.PARTNUM = P.PARTNUM
AND O.NAME = C.NAME
ORDER BY C.NAME
```

NAME	ADDRESS	TOTAL
AAA BIKE	10 OLDTOWN	213.50
AAA BIKE	10 OLDTOWN	2120.00
AAA BIKE	10 OLDTOWN	1200.00
BIKE SPEC	CPT SHRIVE	542.50
BIKE SPEC	CPT SHRIVE	2803.60
BIKE SPEC	CPT SHRIVE	5830.00
BIKE SPEC	CPT SHRIVE	2400.00
JACKS BIKE	24 EGLIN	7420.00
LE SHOPPE	HOMETOWN	2650.00
LE SHOPPE	HOMETOWN	3600.00
TRUE WHEEL	55O HUSKER	196.00
TRUE WHEEL	55O HUSKER	2102.70
TRUE WHEEL	55O HUSKER	1590.00
TRUE WHEEL	55O HUSKER	1200.00

You can make the previous query more specific, thus more useful, by adding the DESCRIPTION column as in the following example :

INPUT/OUTPUT :

```
SELECT C.NAME, C.ADDRESS,
O.QUANTITY * P.PRICE TOTAL,
P.DESCRIPTION
FROM ORDERS O, PART P, CUSTOMER C
WHERE O.PARTNUM = P.PARTNUM
AND O.NAME = C.NAME
ORDER BY C.NAME
```

NAME	ADDRESS	TOTAL	DESCRIPTION
==========	==========	=========	==============
AAA BIKE	10 OLDTOWN	213.50	TIRES
AAA BIKE	10 OLDTOWN	2120.00	ROAD BIKE
AAA BIKE	10 OLDTOWN	1200.00	TANDEM
BIKE SPEC	CPT SHRIVE	542.50	PEDALS
BIKE SPEC	CPT SHRIVE	2803.60	MOUNTAIN BIKE
BIKE SPEC	CPT SHRIVE	5830.00	ROAD BIKE
BIKE SPEC	CPT SHRIVE	2400.00	TANDEM
JACKS BIKE	24 EGLIN	7420.00	ROAD BIKE
LE SHOPPE	HOMETOWN	2650.00	ROAD BIKE
LE SHOPPE	HOMETOWN	3600.00	TANDEM
TRUE WHEEL	55O HUSKER	196.00	SEATS
TRUE WHEEL	55O HUSKER	2102.70	MOUNTAIN BIKE
TRUE WHEEL	55O HUSKER	1590.00	ROAD BIKE
TRUE WHEEL	55O HUSKER	1200.00	TANDEM

This information is a result of joining three tables. You can now use this information to create an invoice.

3.2.2 Non-Equi-Joins

Because SQL supports an equi-join, you might assume that SQL also has a non-equi-join. You would be right! Whereas the equi-join uses an = sign in the WHERE statement, the non-equi-join uses everything but an = sign. For example :

INPUT :

```
SELECT O.NAME, O.PARTNUM, P.PARTNUM,
O.QUANTITY * P.PRICE TOTAL
FROM ORDERS O, PART P
WHERE O.PARTNUM > P.PARTNUM
```

OUTPUT :

NAME	PARTNUM	PARTNUM	TOTAL
TRUE WHEEL	76	54	162.75
BIKE SPEC	76	54	596.75
LE SHOPPE	76	54	271.25
AAA BIKE	76	54	217.00
JACKS BIKE	76	54	759.50
TRUE WHEEL	76	42	73.50
BIKE SPEC	54	42	245.00
BIKE SPEC	76	42	269.50
LE SHOPPE	76	42	122.50
AAA BIKE	76	42	98.00
AAA BIKE	46	42	343.00
JACKS BIKE	76	42	343.00
TRUE WHEEL	76	46	45.75
BIKE SPEC	54	46	152.50
BIKE SPEC	76	46	167.75
LE SHOPPE	76	46	76.25
AAA BIKE	76	46	61.00
JACKS BIKE	76	46	213.50
TRUE WHEEL	76	23	1051.35
TRUE WHEEL	42	23	2803.60

...

This listing goes on to describe all the rows in the join WHERE O.PARTNUM > P.PARTNUM. In the context of your bicycle shop, this information does not have much meaning, and in the real world the equi-join is far more common than the non-equi-join. However, you may encounter an application in which a non-equi-join produces the perfect result.

3.2.3 Outer Joins versus Inner Joins

Just as the non-equi-join balances the equi-join, an outer join complements the inner join. An inner join is where the rows of the tables are combined with each other, producing a number of new rows equal to the product of the number of rows in each table. Also, the inner join uses these rows to determine the result of the WHERE clause. An outer join groups the two tables in a slightly different way. Using the PART and ORDERS tables from the previous examples, perform the following inner join:

INPUT :

```
SELECT P.PARTNUM, P.DESCRIPTION, P.PRICE,
O.NAME, O.PARTNUM
FROM PART P
JOIN ORDERS O ON ORDERS.PARTNUM = 54
```

OUTPUT :

PARTNUM	DESCRIPTION	PRICE	NAME	PARTNUM
54	PEDALS	54.25	BIKE SPEC	54
42	SEATS	24.50	BIKE SPEC	54
46	TIRES	15.25	BIKE SPEC	54
23	MOUNTAIN BIKE	350.45	BIKE SPEC	54
76	ROAD BIKE	530.00	BIKE SPEC	54
10	TANDEM	1200.00	BIKE SPEC	54

The result is that all the rows in PART are spliced on to specific rows in ORDERS where the column PARTNUM is 54. Here's a RIGHT OUTER JOIN statement :

INPUT/OUTPUT :

```
SELECT P.PARTNUM, P.DESCRIPTION, P.PRICE,
```

```
O.NAME, O.PARTNUM
FROM PART P
RIGHT OUTER JOIN ORDERS O ON ORDERS.PARTNUM = 54
```

PARTNUM	DESCRIPTION	PRICE	NAME	PARTNUM
<null>	<null>	<null>	TRUE WHEEL	23
<null>	<null>	<null>	TRUE WHEEL	76
<null>	<null>	<null>	TRUE WHEEL	10
<null>	<null>	<null>	TRUE WHEEL	42
54	PEDALS	54.25	BIKE SPEC	54
42	SEATS	24.50	BIKE SPEC	54
46	TIRES	15.25	BIKE SPEC	54
23	MOUNTAIN BIKE	350.45	BIKE SPEC	54
76	ROAD BIKE	530.00	BIKE SPEC	54
10	TANDEM	1200.00	BIKE SPEC	54
<null>	<null>	<null>	BIKE SPEC	10
<null>	<null>	<null>	BIKE SPEC	23
<null>	<null>	<null>	BIKE SPEC	76
<null>	<null>	<null>	LESHOPPE	76
<null>	<null>	<null>	LE SHOPPE	10
<null>	<null>	<null>	AAA BIKE	10
<null>	<null>	<null>	AAA BIKE	76
<null>	<null>	<null>	AAA BIKE	46
<null>	<null>	<null>	JACKS BIKE	76

This type of query is new. First you specified a RIGHT OUTER JOIN, which caused SQL to return a full set of the right table, ORDERS, and to place nulls in the fields where ORDERS.PARTNUM <> 54. Following is a LEFT OUTER JOIN statement :

```
INPUT/OUTPUT :
SELECT P.PARTNUM, P.DESCRIPTION,P.PRICE,
O.NAME, O.PARTNUM
FROM PART P
LEFT OUTER JOIN ORDERS O ON ORDERS.PARTNUM = 54
```

PARTNUM	DESCRIPTION	PRICE	NAME	PARTNUM
54	PEDALS	54.25	BIKE SPEC	54
42	SEATS	24.50	BIKE SPEC	54
46	TIRES	15.25	BIKE SPEC	54
23	MOUNTAIN BIKE	350.45	BIKE SPEC	54
76	ROAD BIKE	530.00	BIKE SPEC	54
10	TANDEM	1200.00	BIKE SPEC	54

You get the same six rows as the INNER JOIN. Because you specified LEFT (the LEFT table), PART determined the number of rows you would return. Because PART is smaller than ORDERS, SQL saw no need to pad those other fields with blanks.

Don't worry too much about inner and outer joins. Most SQL products determine the optimum JOIN for your query. In fact, if you are placing your query into a stored procedure (or using it inside a program (both stored procedures and Embedded SQL covered on Day 13, "Advanced SQL Topics"), you should not specify a join type even if your SQL implementation provides the proper syntax. If you do specify a join type, the optimizer chooses your way instead of the optimum way.