

## INTRODUCTION TO DATABASE SYSTEMS

**DATA:** Data is the raw material from which useful information is derived.

Data is a collection of facts which is unorganized but can be made organized into useful information.

Data can be defined in many ways. Information science defines data as unprocessed information

In computer science, data is anything in a form suitable for use with a computer. Data is often distinguished from programs. A program is a set of instructions that detail a task for the computer to perform. In this sense, data is thus everything that is not program code.

**INFORMATION:** Information is data that have been organized and communicated in a coherent and meaningful manner.

Data that have been processed in such a way so as to increase the knowledge of the person who uses the data is known as information.

Data is converted into information, and information is converted into knowledge. Knowledge; information evaluated and organized so that it can be used purposefully.

**META DATA:** Data that describe the properties of other data is known as Meta data

META DATA = data about data

**DATABASE:** A database is a collection of information that is organized so that it can easily be accessed, managed, and updated. In one view, databases can be classified according to types of content: bibliographic, full-text, numeric, and images.

**DBMS:** The database management system (DBMS), is a computer software program that is designed as the means of managing all databases that are currently installed on a system hard drive or network.

DBMS is a collection of interrelated data and a set of programs to access those data

Data management is a discipline that focuses on the proper generation , storage and retrieval of data.

Different types of database management systems exist, with some of them designed for the oversight and proper control of databases that are configured for specific purposes

In database management system (DBMS), data files are the files that store the database information, whereas other files, such as index files and data dictionaries, store administrative information, known as metadata.

**KNOWLEDGE:** Knowledge is the body of information and facts about a specific subject .Knowledge implies familiarity, awareness and understanding of information as it applies to an environment.

A key characteristic of knowledge is that “new” knowledge can be derived from “old” knowledge.

**DATABASE SYSTEM:** Data base system is a system to achieve an organized, store a large number of dynamical associated data, facilitate for multi-user accessing to computer hardware, software and data, that it is a computer system with database technology

- Data constitute the building blocks of information
- Information is produced by processing data
- Information is used to reveal the meaning of data.
- Accurate, relevant and timely information is the key to good decision making
- Good decision making is the key to organizational survival in a global environment

## ER MODEL

- **Attribute** - a property or description of an entity. A toy department employee entity could have attributes describing the employee's name, salary, and years of service.
- **Domain** - a set of possible values for an attribute.
- **Entity** - an object in the real world that is distinguishable from other objects.
- **Relationship** - an association among two or more entities.
- **Entity set** - a collection of similar entities such as all of the toys in the toy department.
- **Relationship set** - a collection of similar relationships
- **One-to-many relationship** - a key constraint that indicates that one entity can be associated with many of another entity. An example of a one-to-many relationship is when an employee can work for only one department, and a department can have many employees.
- **Many-to-many relationship** - a key constraint that indicates that many of one entity can be associated with many of another entity. An example of a many-to-many relationship is employees and their hobbies: a person can have many different hobbies, and many people can have the same hobby.
- **Participation constraint** - a participation constraint determines whether relationships must involve certain entities. An example is if every department entity has a manager entity. Participation constraints can either be total or partial. A total participation constraint says that every department has a manager. A partial participation constraint says that every employee does not have to be a manager.
- **Overlap constraint** - within an ISA hierarchy, an overlap constraint determines whether or not two subclasses can contain the same entity.
- **Covering constraint** - within an ISA hierarchy, a covering constraint determines where the entities in the subclasses collectively include all entities in the superclass. For example, with an Employees entity set with subclasses Hourly Employee and Salary Employee, does every Employee entity necessarily have to be within either Hourly Employee or Salary Employee?
- **Weak entity set** - an entity that cannot be identified uniquely without considering some primary key attributes of another identifying owner entity. An example is including Dependent information for employees for insurance purposes.
- **Aggregation** - a feature of the entity relationship model that allows a relationship set to participate in another relationship set. This is indicated on an ER diagram by drawing a dashed box around the aggregation.
- **Role indicator** - If an entity set plays more than one role, role indicators describe the different purpose in the relationship. An example is a single Employee entity set with a relation Reports- To that relates supervisors and subordinates.

## ER DIAGRAM OF UNIVERSITY DATABASE

Consider the following information about a university database:

1. Professors have an SSN, a name, an age, a rank, and a research specialty.
2. Projects have a project number, a sponsor name (e.g., NSF), a starting date, an ending date, and a budget.
3. Graduate students have an SSN, a name, an age, and a degree program (e.g., M.S. or Ph.D.).
4. Each project is managed by one professor (known as the project's principal investigator).
5. Each project is worked on by one or more professors (known as the project's co-investigators).
6. Professors can manage and/or work on multiple projects.
7. Each project is worked on by one or more graduate students (known as the project's research assistants).
8. When graduate students work on a project, a professor must supervise their work on the project. Graduate students can work on multiple projects, in which case they will have a (potentially different) supervisor for each one.
9. Departments have a department number, a department name, and a main office.
10. Departments have a professor (known as the chairman) who runs the department.
11. Professors work in one or more departments, and for each department that they work in, a time percentage is associated with their job.
12. Graduate students have one major department in which they are working on their degree.
13. Each graduate student has another, more senior graduate student (known as a student advisor) who advises him or her on what courses to take.

Design and draw an ER diagram that captures the information about the university. Use only the basic ER model here; that is, entities, relationships, and attributes. Be sure to indicate any key and participation constraints.

## RELATIONAL MODEL

**Basic Structure of relational model - The relational model** for database management is a data model based on predicate logic and set theory. It was invented by Edgar Codd. The fundamental assumption of the relational model is that all data are represented as mathematical n-ary **relations**, an n-ary relation being a subset of the Cartesian product of n sets.

**1) Relation** - The fundamental organizational structure for data in the relational model is the relation. A relation is a two-dimensional table made up of rows and columns. Each relation also called a table, stores data about entities.

A **relational database schema** is a collection of relation schemas, describing one or more relations

**2) Tuples** - The rows in a relation are called tuples. They represent specific occurrences (or records) of an entity. Each row consists of a sequence of values, one for each column in the table. In addition, each row (or record) in a table must be unique. A tuple variable is a variable that stand for a tuple.

**3) Attributes** – The column in a relation is called attribute. The attributes represent characteristics of an entity.

**4) Domain** – For each attribute there is a set of permitted values called domain of that attribute. For all relations ‘r’, the domain of all attributes of ‘r’ should be atomic. A domain is said to be **atomic** if elements of the domain are considered to be indivisible units.

**Database Schema** – Logical design of the database is termed as database schema.

**Database instance** – Database instance is a snapshot of the data in a database at a given instant of time.

**Relation schema** – The concept of relation schema corresponds to the programming notion of type definition. It can be considered as the definition of a domain of values. The database schema is the collection of relation schemas that define a database.

- The **relation cardinality** is the number of tuples in the relation.
- The **relation degree** is the number of fields (or columns) in the relation.

**Relation instance** – The concept of a relation instance corresponds to the programming language notion of a value of a variable. For relation instance, we actually mean the “relation” itself.

**Keys** – A key is the relational means of specifying uniqueness. The keys applicable in relational model are primary key, candidate key and super key.

**1.) Primary key** - A **primary key** is a value that can be used to identify a unique row in a table. Attributes are associated with it.

2.) Candidate key - A **candidate key** of a relation variable is a set of attributes of that relation variable such that (1) at all times it holds in the relation assigned to that variable that there are no two distinct tuples with the same values for these attributes and (2) there is not a proper subset for which (1) holds.

3.) Super key - A **superkey** is defined in the relational model as a set of attributes of a relation variable for which it holds that in all relations assigned to that variable there are no two distinct tuples that have the same values for the attributes in this set.

4.) Foreign key - A **foreign key** is a field or group of fields in a database record that point to a key field or group of fields forming a key of another database record in some (usually different) table. A relation schema,  $r_1$ , derived from an E-R schema may include among its attributes the primary key of another relation schema,  $r_2$ . This attribute is the foreign key from  $r_1$ , referencing  $r_2$ . The relation  $r_1$  is called the referencing relation of the foreign key dependency and  $r_2$  is called the referenced relation of  $r_1$ .

**Schema diagram** – A database schema, along with primary key and foreign key dependencies, can be depicted pictorially by schema diagrams. Each relation in the database schema is represented as a box, with the attributes listed inside it and the relation name above it. If there are primary key attributes, a horizontal line crosses the box, with the primary key attributes listed above the line. Foreign key dependencies appear as arrows from the foreign key attributes of the referencing relation to the foreign key attributes of the referenced relation.

---

### ER TO RELATIONAL MODEL

Consider the university database from previous exercise and the ER diagram you designed. Convert the ER diagram to Relational model(represent in table format)

**1. Table name:professors**

<u>prof-ssn</u>	name	age	rank	speciality

Primary key: prof\_ssn

**2. Table name : Depts**

<u>dno</u>	Dname	office

PRIMARY KEY: dno

**3. Table name Runs**

<u>dno</u>	<u>prof_ssn</u>

PRIMARY KEY: dno +prof\_ssn  
 FOREIGN KEY (prof\_ssn) REFERENCES Professors,  
 FOREIGN KEY (dno) REFERENCES Depts

**4. Table name: Work Dept**

<u>dno</u>	<u>prof_ssn</u>	pc_time

PRIMARY KEY (dno, prof\_ssn),  
 FOREIGN KEY (prof\_ssn) REFERENCES Professors,  
 FOREIGN KEY (dno) REFERENCES Depts )

**5. Table name: Project**

<u>pid</u>	sponsor	start_date	end_date	budget

PRIMARY KEY (pid )

**6. Table name : Graduates**

<u>grad_ssn</u>	age	name	deg_prog	major

PRIMARY KEY (grad\_ssn),  
 FOREIGN KEY (major) REFERENCES Depts

Note that the Major table is not necessary since each Graduate has only one major and so this can be an attribute in the Graduates table.

## 7. Table name: Advisor

<u>senior_ssn</u>	<u>grad_ssn</u>

PRIMARY KEY (senior\_ssn, grad\_ssn),  
 FOREIGN KEY (senior\_ssn) REFERENCES Graduates (grad\_ssn),  
 FOREIGN KEY (grad\_ssn) REFERENCES Graduates

## 8. table name : Manages

<u>pid</u>	<u>prof_ssn</u>

PRIMARY KEY (pid, prof\_ssn),  
 FOREIGN KEY (prof\_ssn) REFERENCES Professors,  
 FOREIGN KEY (pid) REFERENCES Projects

## 9. Table name : Work\_In

<u>pid</u>	<u>prof_ssn</u>

PRIMARY KEY (pid, prof\_ssn),  
 FOREIGN KEY (prof\_ssn) REFERENCES Professors,  
 FOREIGN KEY (pid) REFERENCES Projects

Observe that we cannot enforce the participation constraint for Projects in the Work In table without check constraints or assertions in SQL.

## 10. Table name : Supervises

<u>prof_ssn</u>	<u>grad_ssn</u>	<u>pid</u>

PRIMARY KEY (prof\_ssn, grad\_ssn, pid),  
 FOREIGN KEY (prof\_ssn) REFERENCES Professors,  
 FOREIGN KEY (grad\_ssn) REFERENCES Graduates,  
 FOREIGN KEY (pid) REFERENCES Projects

Note that we do not need an explicit table for the Work Proj relation since every time a Graduate works on a Project, he or she must have a Supervisor.

## INTRODUCTION TO SQL

SQL (Structured Query Language) is a database computer language designed for the retrieval and management of data in relational database management systems (RDBMS), database schema creation and modification, and database object access control management.

SQL is a programming language for querying and modifying data and managing databases. SQL was standardized first by the ANSI and (later) by the ISO. Most database management systems implement a majority of one of these standards and add their proprietary extensions. SQL allows the retrieval, insertion, updating, and deletion of data.

A database management system also includes management and administrative functions. Most -- if not all -- implementations also include a Command-line Interface (SQL/CLI) that allows for the entry and execution of the language commands, as opposed to only providing an API intended for access from a GUI.

The first version of SQL was developed at IBM by Donald D. Chamberlin and Raymond F. Boyce in the early 1970s. This version, initially called SEQUEL, was designed to manipulate and retrieve data stored in IBM's original relational database product, System R. IBM patented their version of SQL in 1985, while the SQL language was not formally standardized until 1986, by the American National Standards Institute (ANSI) as SQL-86. Subsequent versions of the SQL standard have been released by ANSI and as International Organization for Standardization (ISO) standards.

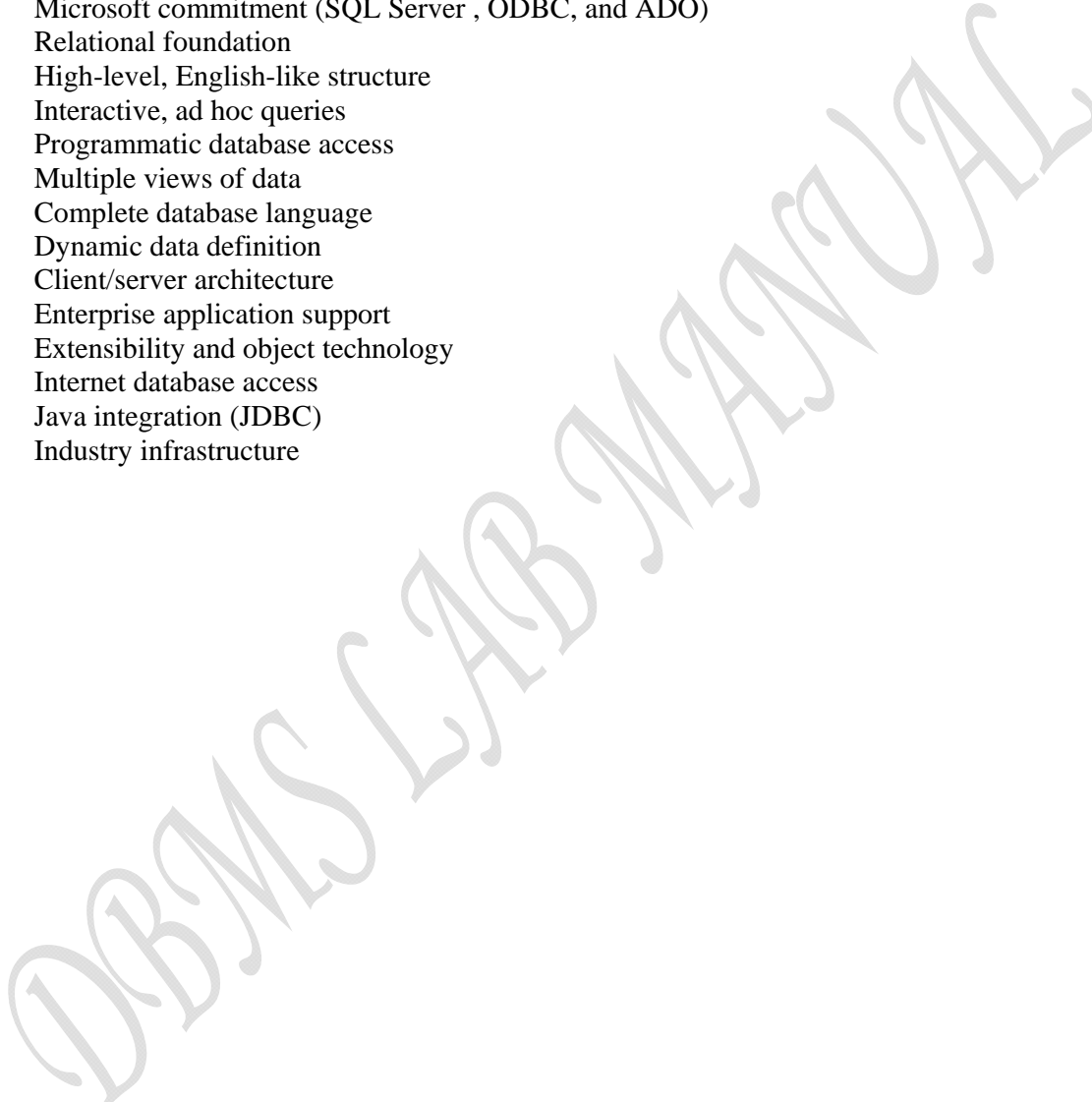
Originally designed as a declarative query and data manipulation language, variations of SQL have been created by SQL database management system (DBMS) vendors that add procedural constructs, control-of-flow statements, user-defined data types, and various other language extensions. With the release of the SQL:1999 standard, many such extensions were formally adopted as part of the SQL language via the SQL Persistent Stored Modules (SQL/PSM) portion of the standard. Common criticisms of SQL include a perceived lack of cross-platform portability between vendors, inappropriate handling of missing data (see Null (SQL)), and unnecessarily complex and occasionally ambiguous language grammar and semantics.

---



## FEATURES OF SQL

SQL is both an easy-to-understand language and a comprehensive tool for managing data. Some of the major features of SQL are

- Vendor independence
  - Portability across computer systems
  - SQL standards
  - IBM endorsement and commitment (DB2)
  - Microsoft commitment (SQL Server , ODBC, and ADO)
  - Relational foundation
  - High-level, English-like structure
  - Interactive, ad hoc queries
  - Programmatic database access
  - Multiple views of data
  - Complete database language
  - Dynamic data definition
  - Client/server architecture
  - Enterprise application support
  - Extensibility and object technology
  - Internet database access
  - Java integration (JDBC)
  - Industry infrastructure
- 

## OVERVIEW OF SQL DDL, DML AND DCL COMMANDS.

**DDL** is Data Definition Language statements.

Data Definition Language (DDL) statements are used to define the database structure or schema.

DDL Commands: Create , Alter ,Drop , Rename, Truncate

Some examples:

CREATE - to create objects in the database

ALTER - alters the structure of the database

DROP - delete objects from the database

TRUNCATE - remove all records from a table, including all spaces allocated for the records are removed

COMMENT - add comments to the data dictionary

GRANT - gives user's access privileges to database

REVOKE - withdraw access privileges given with the GRANT command

**DML** is Data Manipulation Language statements. Data Manipulation Language (DML) statements are used for managing data within schema objects

Some examples:

SELECT - retrieve data from the a database

INSERT - insert data into a table

UPDATE - updates existing data within a table

DELETE - deletes all records from a table, the space for the records remain

CALL - call a PL/SQL or Java subprogram

LOCK TABLE - control concurrency

**DCL:** Data Control Language (DCL) statements is used to create roles, permissions, and referential integrity as well it is used to control access to database by securing it.

DCL Commands: Grant, Revoke

GRANT - gives user's access privileges to database

REVOKE - withdraw access privileges given with the GRANT command

**TCL:** Transaction Control (TCL) statements are used to manage the changes made by DML statements. It allows statements to be grouped together into logical transactions.

TCL Commands: Commit, Rollback, Save point

COMMIT - save work done

SAVEPOINT - identify a point in a transaction to which you can later roll back

ROLLBACK - restore database to original since the last COMMIT

**SQL BASIC DATA TYPES**

**char (n)**: Fixed length character data (String), n characters long.

Eg: char(40)

**varchar2(n)**: variable length character string. Only the bytes used for a string require storage

Eg: varchar2(80)

**number(o,d)**: numeric datatype for integer and reals.

o : overall number of digits

d: number of digits to the right of the decimal point

Eg: number(5,2) ; cannot contain anything larger than 999.99

Data types derived from number are int[eger],dec[imal],smallint and real.

**Date**: date datatype for storing date and time. Default format for date is DD-MMM-YY

Eg: 15-AUG-1947

---

**HOW TO WRITE AND EXECUTE SQL, PL/SQL COMMANDS/PROGRAMS:**

- 1). Open your oracle application by the following navigation  
Start->all programs->Oracle Database 10g Express Edition ->Run SQL Command Line
- 2). You will be asked for user name, password.  
You have to enter user name, pass word.
- 3). Upon successful login you will get SQL prompt (SQL>).  
In two ways you can write your programs:
  - a) directly at SQL prompt (or)
  - b) in sql editor.

If you type your programs at sql prompt then screen will look like follow:

```
SQL> SELECT ename,empno,  
2 sal from  
3 emp;
```

where 2 and 3 are the line numbers and rest is the command /program.....

to execute above program/command you have to press '/' then enter.

Here editing the program is somewhat difficult; if you want to edit the previous command then you have to open sql editor (by default it displays the sql buffer contents). By giving 'ed' at sql prompt.(this is what I mentioned as a second method to type/enter the program).In the sql editor you can do all the formatting/editing/file operations directly by selecting menu options provided by it.

To execute the program which saved; do the following

```
SQL> @ programname.sql (or)
```

```
SQL> Run programname.sql
```

Then press '\ ' key and enter.

To save the day`s session ;do the following

```
SQL>commit;
```

This how we can write, edit and execute the sql command and programs.

Always you have to save your programs in your own logins.

## BASIC SQL COMMANDS

**1. The CREATE TABLE Command:** - it defines each column of the table uniquely. Each column has minimum of three attributes, a name, data type and size.

**Syntax:**

```
SQL>create table <table name> (
    <column1> <datatype><size>,
    <column2> <datatype><size>,
    .
    .
    <column n> <datatype><size>);
```

**Ex:**

```
SQL>create table student(  stunum number(4) ,
                          stuname varchar2(10),
                          stubranch char(4));
```

SQL> Table created.

Relational schema for student relation:

SQL> desc student;

```
SQL> NAME          NULL?      TYPE
-----
stunum            number(4)
stuname           varchar2(10)
stubranch         char(4)
```

**2. Adding table rows / Inserting Data into Tables:** - once a table is created the most natural thing to do is load this table with data to be manipulated later.

**Syntax 1:**

```
SQL> insert into <tablename> (<column1>,<column2>.....<column n>) values(<value1>,
    <value 2>.....<value n>);
```

**Syntax 2:**

```
SQL>insert into <tablename> values (&<column1>,&<column2>.....,&<column n>);
```

**Syntax 3:**

```
SQL>insert into <tablename> values (<val 1>,<val 2>.....,<val n>);
```

*Attributes with char/varchar datatype are placed within single quotes( ' ' ).*

**Ex 1:**

```
SQL> insert into student (stunum,stuname,stubranch) values(585,'knreddy','cse');
1 row created
```

**Ex 2:**

```
SQL>insert into student values(&snum,'&stuname','&stubranch');
Enter value for stunum: 596
Enter value for stuname:raju
Enter value for stubranch:eee
old 1: insert into student values(&snum,'&stuname','&stubranch')
new 1: insert into student values(596,'raju','eee')
1 row created.
```

SQL>/

**3. Viewing data in the tables / Listing table rows:** - once data has been inserted into a table, the next most logical operation would be to view what has been inserted.

a) Listing all rows and all columns

**Syntax:**

SQL> **select** <column1> to <column n>

2        from tablename;

(or)

SQL> **select** \*

2        from tablename;

\* → asteric means all

Filtering table data: - while viewing data from a table, it is rare that all the data from table will be required each time. Hence, sql must give us a method of filtering out data that is not required data.

a) Selected columns and all rows:

**Syntax:**

SQL>select <column1>,<column2>

2        from <tablename>;

b) selected rows and all columns:

**Syntax:**

SQL> select \*

2        from <tablename>

3        where <condition>;

c) selected columns and selected rows

**Syntax:**

SQL>select <column1>,<column2>

2        from <tablename>

3        where<condition>;

. Eliminating duplicate rows when using a select statement

Syntax: SQL> select distinct <column> from <tablename>;

**4. Delete operations.**

a) remove all rows

**Syntax:**

SQL>**delete** from <tablename>;

b) removal of a specified row/s

**Syntax:**

SQL>**delete** from <tablename>

2        where <condition>;

**5. Updating the contents of a table.**

a) updating all rows

**Syntax:**

```
SQL>update <tablename>  
2 set <column>=<exp>;
```

b) updating seleted records.

**Syntax:**

```
SQL>update <tablename>  
2 set <column>=<exp>,  
3 where <condition>;
```

**6. Modifying the structure of tables.**

a) add new columns

**Syntax:**

```
SQL>Alter table <tablename>  
2 add(<new column> <datatype(size),<newcolumn>datatype(size));
```

**Ex:**

```
SQL>alter table student add(location char(8));
```

b) Dropping a column from a table.

**Syntax:**

```
SQL>alter table <tablename> drop column <col>;
```

**Ex:**

```
SQL> alter table emp drop column sal;
```

c). Modifying existing columns.

**Syntax:**

```
SQL>alter table <tablename> modify(<column> <newdatatype>(<newsized>));
```

**Ex:**

```
SQL>alter table student modify(stuname varchar2(15));
```

**Restrictions on alter table:**

The following tasks cannot be performed when using the ALTER TABLE clause:

- change the name of table
- change the name of column
- decrease the size of a column if table data exists

**7. Renaming the tables****Syntax:**

```
SQL> Rename <oldtable> to <new table>;
```

**Ex:**

```
SQL> rename student to student;
```

**8. Truncating the tables.**

Truncate table empties a table completely.

Equivalent to DELETE command for deleting all rows

**Syntax:**

SQL>truncate table <tablename>;

**Ex:**

SQL>trunc table student1;

(or)

SQL> truncate table studetn1;

**9. Destroying tables.**

Completely destroys/drops the table

**Syntax:**

SQL>drop table <tablename>;

**Ex:**

SQL> drop table student;

**10. Sorting data in a table.**

The rows retrived from the table will be sorted in either ascending or descending order

**Syntax:**

```
SQL> select *
      2     from <tablename>
      3     order by <columnname> <[sortorder]>;
```

If sort order is mentioned as DESC(descending ), it sorts in descending order.

The default sort is ascending order

**11. Creating a table from a table****Syntax:**

```
SQL> create table <table name>(<column1>,<column2>)
      2     as selected <column1>,<column2>
      3     from <table name>;
```

**12. Finding out the table created by a user:****Syntax:**

SQL> select \* from tab;

**For Saving the data use the following command immediately after every login**

SQL> set autocommit on;

**Difference between TRUNCATE and DELETE statement:**

- TRUNCATE is a DDL statement whereas DELETE is a DML statement
- TRUNCATE deletes all records from the table whereas DELETE can also be used to selectively delete records fro a table using WHERE clause
- TRUNCATE releases the memory occupied by the records of the table whereas DELETE does not do so
- Data removed using TRUNCATE cannot be recovered whereas data removed using DELETE can be recovered (using ROLL BACK, a DCL statement).



---

## SQL CONSTRAINTS

### Types of data constraints.

a) **Not Null** constraint ensures that a column does not accept nulls.

**Syntax:** **Not null** constraint at column level

<column> <datatype>(size) not null;

b) **Unique** Constraint ensures that all values in a column are unique.

**Syntax:**

Unique constraint at column level.

<column> <datatype>(size)unique;

unique constraint at table level:

**Syntax:**

SQL>create table <tablename>( <column format> ,unique(<column1>,<column2>);

c) **Primary Key** constraint

**Syntax:** at column level

<column> <datatype>(size)primary key;

at table level.

SQL> create table <tablename>( <column format> ,primary key(column1>,<column2>);

d) **Foreign Key** constraint

**Syntax:**

at column level.

<column> <datatype>(size) references <tablename>[<column>];

**Syntax:** at table level

SQL>create table <tablename>( <column format> ,foreign key(<column1>) references

<tablename>[( <column> )

e) **Check constraint:** is used to validate data when an attribute value is entered . The CHECK constraint does precisely what its name suggests: it checks to see that a specified condition exists.

**Syntax:**

<column> <datatype>(size) check(<logical expression>)

Example-SQL constraints:

1. Create a table emp (empname varchar2 notnull;empnum number unique;empsal number check empsal>10000);

```
SQL>create table emp(empname varchar2(15) not null,
2 empnum number(5) unique,
3 empsal number(10) check(empsal > 10000));
```

2. Write the relational scheme of emp table

```
SQL>desc emp;
```

Name	Null?	Type
EMPNAME	NOT NULL	VARCHAR2(15)
EMPNUM		NUMBER(5)
EMPSAL		NUMBER(10)

3. Insert at least 3 rows in to emp table

```
SQL>insert into emp values('&empname',&empnum,&empsal);
```

```
Enter value for empname: knreddy
```

```
Enter value for empnum: 250
```

```
Enter value for empsal: 30000
```

```
old 1 : insert into emp values('&empname',&empnum,&empsal)
```

```
new 1:insert into emp values('knreddy',250,30000)
```

```
1 row created.
```

```
SQL>/
```

```
Enter value for empname: raju
```

```
Enter value for empnum: 252
```

```
Enter value for empsal: 20000
```

```
old 1 : insert into emp values('&empname',&empnum,&empsal)
```

```
new 1:insert into emp values('raju',252,20000)
```

```
1 row created.
```

```
SQL>/
```

```
Enter value for empname: ajay
```

```
Enter value for empnum: 256
```

```
Enter value for empsal: 25000
```

```
old 1 : insert into emp values('&empname',&empnum,&empsal)
```

```
new 1:insert into emp values('ajay',256,25000)
```

```
1 row created.
```

4. Query the table values

```
SQL>select * from emp;
```

EMPNAME	EMPNUM	EMPSAL
knreddy	250	30000
raju	252	20000
ajay	256	25000

5. Insert rows that violate constraints and write the error messages.

```
SQL> insert into emp values('&empname',&empnum,&empsal);
Enter value for empname:   ravi
Enter value for empnum:    250
Enter value for empsal:    20000
old 1 : insert into emp values('&empname',&empnum,&empsal)
new 1:insert into emp values('ravi',250,20000)
insert into emp values('ravi',250,20000)
*
ERROR at line 1:
ORA-00001: unique constraint (SYSTEM.SYS_C003996) violated
```

```
SQL> insert into emp values('&empname',&empnum,&empsal);
Enter value for empname:
Enter value for empnum:    259
Enter value for empsal:    20000
old 1 : insert into emp values('&empname',&empnum,&empsal)
new 1:insert into emp values(' ',259,20000)
insert into emp values(' ',259,20000)
*
ERROR at line 1:
ORA-01400:cannot insert NULL into ("SYSTEM"."EMP"."EMPNAME")
```

```
SQL>insert into emp values('&empname',&empnum,&empsal);
Enter value for empname:   rajanna
Enter value for empnum:    254
Enter value for empsal:    9000
old 1 : insert into emp values('&empname',&empnum,&empsal)
new 1:insert into emp values('rajanna',254,9000)
insert into emp values('rajanna',254,9000)
*
ERROR at line 1:
ORA-02290: CHECK CONSTRAINT(SYSTEM.SYS_C003995) violated
```

EXCERSE-1

**WRITE SQL QUERIES AND THE CORRESPONDING OUTPUT FOR THE FOLLOWING QUESTIONS:**

- 1) Create a table CLASS the following schema.

class(stunum: number,stuname:varchar2,stubbranch:varchar2,stumarks)

Apply the following constraints to the class table while creating the table.

- stunum is unique – use UNIQUE constraint
  - stuname should not be a null value—use NOT NULL constraint
  - stubbranch should contain only cse,ece,eee,civil,mechanical(i.e, the column should not allow other than these branches)-use CHECK constraint
  - stumarks should not exceed 100—use CHECK constraint
- 2) Give the relational schema of the class table
- 3) Insert at least 10 rows to the class table
- 4) Query all the values of class table
- 5) Find the stunum,stuname whose branch is cse
- 6) Find students who had secured more than 85 marks
- 7) Add AGE,LOCATION column to the class table
- 8) Query the values of table at this moment
- 9) Set the location of cse students as nandyal, ece students as kurnool, eee students as kadapa,civil students as allagadda,mechanical students as proddatur
- 10) Set the age of all students as 19
- 11) Query the values of table at this moment.
- 12) Find the students of kadapa
- 13) Delete students who belong to allagadda.
- 14) Delete age column
- 15) Write the error message trying to insert a student detail with same stunum
- 16) Query the values of the table at this moment.

1. SQL> create table class(stunum number(10) unique,  
 2 stuname varchar2(15) not null,  
 3 stubbranch varchar2(10) check ( stubbranch in('cse','ece','eee','civil','mechanical')),  
 4 stumarks number(3) check(stumarks < 100));  
 Table created.

2. SQL>desc class;

Name	Null?	Type
STUNUM		NUMBER(10)
STUNAME	NOT NULL	VARCHAR2(15)
STUBRANCH		VARCHAR2(10)
STUMARKS		NUMBER(3)

```
3. SQL> insert into class values(&stunum,&stuname','&stubbranch',&stumarks);
Enter value for stunum: 1
Enter value for stuname: abhi
Enter value for stubbranch: cse
Enter value for stumarks: 90
old 1: insert into class values(&stunum,'&stuname','&stubbranch',&stumarks)
new 1: insert into class values(1,'abhi','cse',90)
```

1 row created.

```
SQL> /
Enter value for stunum: 2
Enter value for stuname: balu
Enter value for stubbranch: ece
Enter value for stumarks: 90
old 1: insert into class values(&stunum,'&stuname','&stubbranch',&stumarks)
new 1: insert into class values(2,'balu','ece',90)
```

1 row created.

```
SQL> /
Enter value for stunum: 3
Enter value for stuname: chandra
Enter value for stubbranch: cse
Enter value for stumarks: 85
old 1: insert into class values(&stunum,'&stuname','&stubbranch',&stumarks)
new 1: insert into class values(3,'chandra','cse',85)
```

```
SQL> /
Enter value for stunum: 4
Enter value for stuname: deva
Enter value for stubbranch: cse
Enter value for stumarks: 90
old 1: insert into class values(&stunum,'&stuname','&stubbranch',&stumarks)
new 1: insert into class values(4,'deva','cse',90)
```

1 row created.

```
SQL> /
Enter value for stunum: 5
Enter value for stuname: eswar
Enter value for stubbranch: eee
Enter value for stumarks: 85
old 1: insert into class values(&stunum,'&stuname','&stubbranch',&stumarks)
new 1: insert into class values(5,'eswar','eee',85)
```

1 row created.

```
SQL> /
Enter value for stunum: 6
Enter value for stuname: naresh
Enter value for stubranch: ece
Enter value for stumarks: 80
old 1: insert into class values(&stunum,&stuname,&stubranch,&stumarks)
new 1: insert into class values(6,'naresh','ece',80)
```

1 row created.

```
SQL> /
Enter value for stunum: 7
Enter value for stuname: ganesh
Enter value for stubranch: mechanical
Enter value for stumarks: 80
old 1: insert into class values(&stunum,&stuname,&stubranch,&stumarks)
new 1: insert into class values(7,'ganesh','mechanical',80)
```

1 row created.

```
SQL> /
Enter value for stunum: 8
Enter value for stuname: mahesh
Enter value for stubranch: civil
Enter value for stumarks: 80
old 1: insert into class values(&stunum,&stuname,&stubranch,&stumarks)
new 1: insert into class values(8,'mahesh','civil',80)
```

1 row created.

```
SQL> /
Enter value for stunum: 9
Enter value for stuname: jagadesh
Enter value for stubranch: civil
Enter value for stumarks: 80
old 1: insert into class values(&stunum,&stuname,&stubranch,&stumarks)
new 1: insert into class values(9,'jagadesh','civil',80)
```

1 row created.

```
SQL> /
Enter value for stunum: 10
Enter value for stuname: karthik
Enter value for stubranch: cse
Enter value for stumarks: 85
old 1: insert into class values(&stunum,&stuname,&stubranch,&stumarks)
new 1: insert into class values(10,'karthik','cse',85)
```

1 row created.

4. SQL> select \*  
2 from class;

STUNUM	STUNAME	STUBRANCH	STUMARKS
1	abhi	cse	90
2	balu	ece	90
3	chandra	cse	85
4	deva	cse	90
5	eswar	eee	85
6	naresh	ece	80
7	ganesh	mechanical	80
8	mahesh	civil	80
9	jagadesh	civil	80
10	karthik	cse	85

10 rows selected.

5. SQL> select stunum, stuname  
2 from class  
3 where stubranch='cse';

STUNUM	STUNAME
1	abhi
3	chandra
4	deva
10	karthik

6. SQL> select \*  
2 from class  
3 where stumarks>85;

STUNUM	STUNAME	STUBRANCH	STUMARKS
1	abhi	cse	90
2	balu	ece	90
4	deva	cse	90

7. SQL> alter table class  
2 add(age number(2), location varchar2(10));

Table altered.

8. SQL> select \*  
2 from class;

---

STUNUM	STUNAME	STUBRANCH	STUMARKS	AGE	LOCATION
1	abhi	cse	90		
2	balu	ece	90		
3	chandra	cse	85		
4	deva	cse	90		
5	eswar	eee	85		
6	naresh	ece	80		
7	ganesh	mechanical	80		
8	mahesh	civil	80		
9	jagadesh	civil	80		
10	karthik	cse	85		

---

10 rows selected.

```
9. SQL> update class
    2 set location='nandyal'
    3 where stubranch='cse';
```

4 rows updated.

```
SQL> update class
    2 set location='kurnool'
    3 where stubranch='ece';
```

2 rows updated.

```
SQL> update class
    2 set location='kadapa'
    3 where stubranch='eee';
```

1 row updated.

```
SQL> update class
    2 set location='allagadda'
    3 where stubranch='civil';
```

2 rows updated.

```
SQL> update class
    2 set location='proddatur'
    3 where stubranch='mechanical';
```

1 row updated.

```
10. SQL> update class
    1 set age=19;
```

10 rows updated.



11. SQL> select \*  
2 from class;

STUNUM	STUNAME	STUBRANCH	STUMARKS	AGE	LOCATION
1	abhi	cse	90	19	nandyal
2	balu	ece	90	19	kurnool
3	chandra	cse	85	19	nandyal
4	deva	cse	90	19	nandyal
5	eswar	eee	85	19	kadapa
6	naresh	ece	80	19	kurnool
7	ganesh	mechanical	80	19	proddatur
8	mahesh	civil	80	19	allagadda
9	jagadesh	civil	80	19	allagadda
10	karthik	cse	85	19	nandyal

10 rows selected.

12. SQL> select \*  
2 from class  
3 where location='kadapa';

STUNUM	STUNAME	STUBRANCH	STUMARKS	AGE	LOCATION
5	eswar	eee	85	19	kadapa

13. SQL> delete class  
2 where location='allagadda';

2 rows deleted.

14. SQL> alter table class  
2 drop column age;

Table altered.

15. SQL> insert into class values(3,'knreddy','cse','nandyal');  
insert into class values(3,'knreddy','cse','nandyal')  
\*

ERROR at line 1:

ORA-00001: unique constraint (SYSTEM.SYS\_C003997) violated

16. SQL> select \*  
2 from class;

STUNUM	STUNAME	STUBRANCH	STUMARKS	LOCATION
1	abhi	cse	90	nandyal
2	balu	ece	90	kurnool
3	chandra	cse	85	nandyal
4	deva	cse	90	nandyal
5	eswar	eee	85	kadapa
6	naresh	ece	80	kurnool
7	ganesh	mechanical	80	proddatur
10	karthik	cse	85	nandyal

8 rows selected.

## OPERATORS IN SQL

### COMPARISON OPERATORS:

SQL provides the following comparison operators

Symbol	Meaning
=	Equal to
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
<> or !=	Not equal to

### ARITHMETIC OPERATORS:

SQL provides the following arithmetic operators. We can use arithmetic operators with table attributes in a column list or in a conditional expression.

Arithmetic operator	Description
+	Add
-	Subtract
*	Multiply
/	Divide
^	Raise to the power of (some application use ** instead of ^)

### LOGICAL OPERATORS:

SQL allows having multiple conditions in a query through the use of logical operators. The logical operators are: **AND, OR, NOT**. The logical operators are used to connect the Boolean expressions in the where clause.

### SPECIAL OPERATORS:

ANSI –standard SQL allows the use of special operators in conjunction with the WHERE clause. These special operators include:

**BETWEEN:** used to check whether an attribute value is within a range.

**IS NULL:** used to check whether an attribute value is null.

**IN:** used to check whether an attribute value matches any value within a value list.

**EXISTS:** used to check whether a sub query returns any rows.

**LIKE:** used to check whether an attribute value matches a given string pattern.

The LIKE special operator is used in conjunction with wildcards to find patterns within string attributes. Standard SQL allows to use the percent sign (%) and underscore ( \_ ) wildcard characters to make matches when the entire string is not known:

- % Means any and all following or preceding characters are eligible. For example 'J%' includes Johnson, James, and July. 'Jo %' includes Johnson, Jones. '%n' includes Johnson, Jagan, Kiran.
- \_ means any one character may be substituted for the underscore

**AGGREGATE FUNCTIONS:**

SQL can perform various mathematical summaries, such as counting the number of rows that contain a specified condition, finding the minimum or maximum values for some specified attribute, summing and averaging the values in a specified column.

Some basic SQL aggregate functions:

<u>Function</u>	<u>Output</u>
COUNT	The number of rows containing non-null values.
MIN	The minimum attribute value encountered in a given column
MAX	The maximum attribute value encountered in a given column
SUM	The sum of all values for a given column
AVG	The arithmetic mean(average) for a specified column

**GROUPING DATA:**

Frequency distribution can be created quickly and easily using the GROUP BY clause within the SELECT statement. The syntax is:

```
SELECT    column list
FROM      table list
[WHERE    condition list]
[GROUP BY column list]
[HAVING   condition list]
[ORDER BY column list [ASC|DESC]];
```

The GROUP BY clause is generally used when you have attributes columns combined with aggregate functions in the SELECT statement.

The GROUP BY clause is valid only when used in conjunction with one of the SQL aggregate functions such as COUNT, MIN, MAX, AVG and SUM.

**EXAMPLE-SQL OPERATORS**

CONSIDER THE FOLLOWING EMPLOYEE TABLE AND WRITE THE QUERIES AND CORRESPONDING RESULT FOR EACH QUERY.

**EMPLOYEE**

EMP_NUM	EMP_NAME	JOB_TYPE	HIREDATE	EMP_SAL	EMP_LOC
1001	KRISHNA	MANAGER	01-JAN-2011	50000	NANDYAL
1002		CLERK	01-JAN-2011	15000	KURNOOL
1003	RAM	CLERK	16-AUG-2011	15000	NANDYAL
1004	MAHESH	ASSTMANAGER	03-MAY-2011	35000	HYDERABAD
1005	VIGNESH	ACCOUNTANT	01-JAN-2011	20000	KADAPA
1006	NAGENDRA	MECHANIC	01-JAN-2011	10000	KADAPA
1007	KIRAN	CLERK	08-JUN-2011	15000	HYDERABAD
1008	LOKESH	ATTENDER	01-JAN-2011	12000	KURNOOL
1009	MOHAN	ATTENDER	16-AUG-2011	12000	KADAPA
1010	PRAVEEN	ADMINOFFICER	02-JAN-2011	40000	NANDYAL

1. Find the employee whose salary is greater than 25000
2. Find the name of employees whose salary is 50000
3. Find the employee who is manager and belongs to nandyal
4. Find the employee who are either manager or belongs to nandyal
5. Find employee whose salary is not less than 20000
6. Find employee whose salary is in between 20000 and 40000
7. Find the name and location of employee who location is nandyal or kurnool
8. Find the rows from employee table whose EMP\_NAME column values are null
9. Find the name of employees whose name starts with 'k'
10. Find the name of employees whose name ends with 'esh'
11. Find the name of employees in which the second character is 'a'
12. Find number of employees.
13. Find minimum salary of the employee
14. Find maximum salary of the employee
15. Find total salary of all employees
16. Find the average salary of the employees
17. Find number of different jobs in the company
18. Find employees whose salary is less than average salary
19. What is the output of the following query:  

```
SQL> SELECT EMP_NAME, EMP_LOC  
      FROM EMPLOYEE  
      GROUP BY EMP_LOC;
```
20. Find the number of employees for each job
21. What is the maximum salary for each job
22. Find the number of each job and name the column that gives number of jobs as jobnum
23. Find the number of employees of each location
24. Find the name and salary of the employee with maximum salary
25. Find the count of employees for each job so that at least two of the employees had salary greater than 10000

1. Find the employee whose salary is greater than 25000

```
SQL> select *
2 from employee
3 where emp_sal > 25000;
```

EMP_NUM	EMP_NAME	JOB_TYPE	HIREDATE	EMP_SAL	EMP_LOC
1001	krishna	manager	01-JAN-11	50000	nandyal
1004	mahesh	asstmanager	03-MAY-11	35000	hyderabad
1010	praveen	adminofficer	02-JAN-11	40000	nandyal

2. Find the name of employees whose salary is 50000

```
SQL> select emp_name
2 from employee
3 where emp_sal=50000;
```

EMP\_NAME

-----  
krishna

3. Find the employee who is manager and belongs to nandyal

```
SQL>select *
2 from employee
3 where job_type='manager' and emp_loc='nandyal';
```

EMP_NUM	EMP_NAME	JOB_TYPE	HIREDATE	EMP_SAL	EMP_LOC
1001	krishna	manager	01-JAN-11	50000	nandyal

4. Find the employee who are either manager or belongs to nandyal

```
SQL> select *
2 from employee
3 where job_type='manager' or emp_loc='nandyal';
```

EMP_NUM	EMP_NAME	JOB_TYPE	HIREDATE	EMP_SAL	EMP_LOC
1001	krishna	manager	01-JAN-11	50000	nandyal
1003	ram	clerk	16-AUG-11	15000	nandyal
1010	praveen	adminofficer	02-JAN-11	40000	nandyal

5. Find employee whose salary is not less than 20000

```
SQL> select *
2 from employee
3 where not(emp_sal < 20000);
```

(or)

```
SQL> select *
2 from employee
3 where emp_sal >= 20000;
```

EMP_NUM	EMP_NAME	JOB_TYPE	HIREDATE	EMP_SAL	EMP_LOC
1001	krishna	manager	01-JAN-11	50000	nandyal
1004	mahesh	asstmanager	03-MAY-11	35000	hyderabad
1005	vignesh	accountant	01-JAN-11	20000	kadapa
1010	praveen	adminofficer	02-JAN-11	40000	nandyal

6. Find employee whose salary is in between 20000 and 40000

```
SQL> select *
2   from employee
3   where emp_sal between 20000 and 40000;
```

EMP_NUM	EMP_NAME	JOB_TYPE	HIREDATE	EMP_SAL	EMP_LOC
1004	mahesh	asstmanager	03-MAY-11	35000	hyderabad
1005	vignesh	accountant	01-JAN-11	20000	kadapa
1010	praveen	adminofficer	02-JAN-11	40000	nandyal

7. Find the name and location of employee who location is nandyal or Kurnool

```
SQL> select *
2   from employee
3   where emp_loc in('nandyal','kurnool');
```

(or)

```
SQL> select *
2   from employee
3   where emp_loc='nandyal' or emp_loc='kurnool';
```

EMP_NUM	EMP_NAME	JOB_TYPE	HIREDATE	EMP_SAL	EMP_LOC
1001	krishna	manager	01-JAN-11	50000	nandyal
1002		clerk	01-JAN-11	15000	kurnool
1003	ram	clerk	16-AUG-11	15000	nandyal
1008	lokesh	attender	01-JAN-11	12000	kurnool
1010	praveen	adminofficer	02-JAN-11	40000	nandyal

8. Find the rows from employee table whose EMP\_NAME column values are null

```
SQL> select *
2   from employee
3   where emp_name is null;
```

EMP_NUM	EMP_NAME	JOB_TYPE	HIREDATE	EMP_SAL	EMP_LOC
1002		clerk	01-JAN-11	15000	kurnool

9. Find the name of employees whose name starts with 'k'

```
SQL> select emp_name
2   from employee
3   where emp_name like 'k%';
```

```
EMP_NAME
-----
krishna
kiran
```

10. Find the name of employees whose name ends with 'esh'

```
SQL> select emp_name
2   from employee
3   where emp_name like '%esh';
```

```
EMP_NAME
-----
mahesh
vignesh
lokesh
```

11. Find the name of employees in which the second character is 'a'

```
SQL> select emp_name
2   from employee
3   where emp_name like '_a%';
```

```
EMP_NAME
-----
ram
mahesh
nagendra
```

12. Find number of employees.

```
SQL> select count(*)
2   from employee;
```

```
COUNT(*)
-----
10
```

13. Find minimum salary of the employee

```
SQL> select min(emp_sal)
2   from employee;
```

```
MIN(EMP_SAL)
-----
10000
```



14. Find maximum salary of the employee

```
SQL> select max(emp_sal)
2   from employee;
```

```
MAX(EMP_SAL)
```

```
-----
50000
```

15. Find total salary of all employees

```
SQL> select sum(emp_sal)
2   from employee;
```

```
SUM(EMP_SAL)
```

```
-----
224000
```

16. Find the average salary of the employees

```
SQL> select avg(emp_sal)
2   from employee;
```

```
AVG(EMP_SAL)
```

```
-----
22400
```

17. Find number of different jobs in the company

```
SQL> select count(distinct job_type)
2   from employee;
```

```
COUNT(DISTINCT JOB_TYPE)
```

```
-----
7
```

18. Find employees whose salary is less than average salary

```
SQL> select *
2   from employee
4   where emp_sal < (select avg(emp_sal)
                    from employee);
```

EMP_NUM	EMP_NAME	JOB_TYPE	HIREDATE	EMP_SAL	EMP_LOC
1002		clerk	01-JAN-11	15000	kurnool
1003	ram	clerk	16-AUG-11	15000	nandyal
1005	vignesh	accountant	01-JAN-11	20000	kadapa
1006	nagendra	mechanic	01-JAN-11	10000	kadapa
1007	kiran	clerk	08-JUN-11	15000	hyderabad
1008	lokesh	attender	01-JAN-11	12000	kurnool
1009	mohan	attender	16-AUG-11	12000	kadapa

7 rows selected.

19. What is the output of the following query:

```
SQL> SELECT EMP_NAME, EMP_LOC
      FROM EMPLOYEE
      GROUP BY EMP_LOC;
```

```
select emp_name, emp_loc
```

```
*
```

```
ERROR at line 1:
```

```
ORA-00979: not a GROUP BY expression
```

20. Find the number of employees for each job

```
SQL> select job_type, count(emp_name)
      2   from employee
      3   group by job_type;
```

```
JOB_TYPE    COUNT(EMP_NAME)
-----
```

```
attender                2
asstmanager             1
clerk                   2
accountant              1
adminofficer            1
manager                 1
mechanic                1
```

7 rows selected.

21. What is the maximum salary for each job

```
SQL> select job_type, max(emp_sal)
      2   from employee
      3   group by job_type;
```

```
JOB_TYPE    MAX(EMP_SAL)
-----
```

```
attender                12000
asstmanager             35000
clerk                   15000
accountant              20000
adminofficer            40000
manager                 50000
mechanic                10000
```

7 rows selected.

22. Find the number of each job and name the column that gives number of jobs as jobnum

```
SQL> select job_type, count(job_type) as jobnum
      2   from employee
      3   group by job_type;
```

JOB_TYPE	JOBNUM
attender	2
asstmanager	1
clerk	3
accountant	1
adminofficer	1
manager	1
mechanic	1

7 rows selected.

23. Find the number of employees of each location

```
SQL> select emp_loc, count(emp_num)
2   from employee
3   group by emp_loc;
```

EMP_LOC	COUNT(EMP_NUM)
hyderabad	2
kadapa	3
nandyal	3
kurnool	2

24. Find the name and salary of the employee with maximum salary

```
SQL> select emp_name, emp_sal
2   from employee
3   where emp_sal=(select max(emp_sal) from employee);
```

EMP_NAME	EMP_SAL
krishna	50000

25. Find the count of employees for each job so that at least two of the employees had salary greater than 10000

```
SQL> select job_type ,count(emp_num)
2   from employee
3   where emp_sal>10000
4   group by job_type
5   having count(emp_num)>=2
```

JOB_TYPE	COUNT(EMP_NUM)
attender	2
clerk	3

## RELATIONAL SET OPERATORS

SQL provides three set manipulation constructs that extend the basic query

SQL supports three operators under the names UNION, INTERSECTION, MINUS (or) EXCEPT.

UNION: The UNION statement combines rows from two or more queries without including duplicate rows.

The syntax of UNION statement is:

```
Query
UNION
Query
```

In other words UNION statement combines the output of two SELECT queries.(SELECT statement must be union compatible)

UNION ALL statement can be used to produce a relation that retains the duplicate rows.

INTERSECT: The INTERSECT statement can be used to combine rows from two queries, returning only the rows that appear in both sets.

The syntax for the INTERSECT statement is:

```
Query
INTERSECT
Query
```

MINUS (or) EXCEPT: The MINUS statement in SQL combines rows from two queries and returns only the rows that appear in the first set but not in the second.

The syntax for the minus statement is :

```
Query
MINUS
Query
```

Example-SQL set operators:

Consider the following two tables

PRODUCT1

P_code	P_descript	price
123455	Flashlight	10.00
123456	Lamp	25.00
123457	Fan	100.00
123458	Bulb	20.00
123459	Grinder	500.00

PRODUCT2

P_code	P_descript	price
345678	Oven	1000.00
345679	Mixer	400.00
123459	Grinder	500.00

1. Combine the data in both the tables using UNION ALL operator
2. Avoid duplicate rows for the above question
3. Find the product descript from the two tables whose price is 500
4. List the common rows in two tables(use INTERECT)
5. List the rows that appear in the product1 table but not in product2 table
6. List the rows that appear in the product2 table but not in product1 table

1. Combine the data in both the tables using UNION ALL operator

```
SQL> select *
2  from product1
3  UNION ALL
4  select *
5  from product2;
```

P_CODE	P_DESCRIPT	PRICE
123455	flashlight	10
123456	lamp	25
123457	fan	100
123458	bulb	20
123459	grinder	500
345678	oven	1000
345679	mixer	400
123459	grinder	500

8 rows selected.

2. Avoid duplicate rows for the above question

```
SQL> select *
2  from product1
3  UNION
4  select *
5  from product2;
```

P_CODE	P_DESCRIPT	PRICE
123455	flashlight	10
123456	lamp	25
123457	fan	100
123458	bulb	20
123459	grinder	500
345678	oven	1000
345679	mixer	400

7 rows selected.

3. Find the product descriopt from the two tables whose price is 500

```
SQL> select p_descript
2  from product1
3  where price=500
4  UNION
5  select p_descript
6  from product2
7  where price=500;
```

```
P_DESCRIPT
-----
grinder
```

4. List the common rows in two tables(use INTERECT)

```
SQL> select *
2   from product1
3   INTERSECT
4   select *
5   from product2;
```

P_CODE	P_DESCRIPT	PRICE
123459	grinder	500

5. List the rows that appear in the product1 table but not in product2 table

```
SQL> select *
2   from product1
3   MINUS
4   select *
5   from product2;
```

P_CODE	P_DESCRIPT	PRICE
123455	flashlight	10
123456	lamp	25
123457	fan	100
123458	bulb	20

6. List the rows that appear in the product2 table but not in product1 table

```
SQL> select *
2   from product2
3   MINUS
4   select *
5   from product1;
```

P_CODE	P_DESCRIPT	PRICE
345678	oven	1000
345679	mixer	400

## SQL JOIN OPERATIONS

The relational join operation merges rows from two tables and returns the rows with one of the following conditions:

- Have common values in common columns(natural join)
- Meet a given join condition(equality or inequality)
- Have a common value in common columns or have no matching values(outer join)

### SQL JOIN EXPRESSIN STYLES

JOIN CLASSIFICATION	JOIN TYPE	SQL SYNTAX EXAMPLE	DESCRIPTION
CROSS	CROSS JOIN	SELECT * FROM T1,T2; (old style)  SELECT * FROM T1 CROSS JOIN T2;	Returns the Cartesian product of T1 and T2.
INNER	Old-style join	SELECT * FROM T1,T2 WHERE T1.C1=T2.C1;	Returns only the rows that meet the join condition in the where clause (old style).Only the rows with matching values are selected.
	NATURAL JOIN	SELECT * FROM T1 NATURAL JOIN T2;	Returns only the rows with matching values in the matching columns. The matching columns must have the same names and similar data types.
	JOIN USING	SELECT * FROM T1 JOIN T2 USING( C1)	Returns only the rows with matching values in the columns indicated in the USING clause
	JOIN ON	SELECT * FROM T1 JOIN T2 ON T1.C1=T2.C1;	Returns only the rows with matching values in the columns indicated in the ON clause
OUTER	LEFT JOIN	SELECT * FROM T1 LEFT OUTER JOIN T2 ON T1.C1=T2.C1;	Returns rows with matching values and includes all rows from the left table (T1) with unmatched values.
	RIGHT JOIN	SELECT * FROM T1 RIGHT OUTER JOIN T2 ON T1.C1=T2.C1;	Returns rows with matching values and includes all rows from the right table (T2) with unmatched values.
	FULL JOIN	SELECT * FROM T1 FULL OUTER JOIN T2 ON T1.C1=T2.C1;	Returns rows with matching values and includes all rows from both tables (T1and T2) with unmatched values.

Consider the tables CUSTOMER and AGENT; perform different SQL join operations and write corresponding results

CUSTOMER

CUS_CODE	CUS_NAME	CUS_ZIP	AGENT_CODE
1132445	W	145	231
1217782	A	145	125
1312243	Ra	129	167
1321242	Ro	134	125
1542311	S	134	421
1657399	V	145	231

AGENT

AGENT_CODE	AGENT_PHONE
125	9985707288
167	8985291308
231	9885434311
333	9704128379

CROSS JOIN:

SQL> select \*

(old style)

2 from customer,agent;

CUS\_CODE CUS\_NAME CUS\_ZIP AGENT\_CODE AGENT\_CODE AGENT\_PHONE

```
-----
```

1132445	w	145	231	125	9985707288
1217782	a	145	125	125	9985707288
1312243	ra	129	167	125	9985707288
1321242	ro	134	125	125	9985707288
1542311	s	134	421	125	9985707288
1657399	v	145	231	125	9985707288
1132445	w	145	231	167	8985291308
1217782	a	145	125	167	8985291308
1312243	ra	129	167	167	8985291308
1321242	ro	134	125	167	8985291308
1542311	s	134	421	167	8985291308

CUS\_CODE CUS\_NAME CUS\_ZIP AGENT\_CODE AGENT\_CODE AGENT\_PHONE

```
-----
```

1657399	v	145	231	167	8985291308
1132445	w	145	231	231	9885434311
1217782	a	145	125	231	9885434311
1312243	ra	129	167	231	9885434311
1321242	ro	134	125	231	9885434311
1542311	s	134	421	231	9885434311
1657399	v	145	231	231	9885434311
1132445	w	145	231	333	9704128379
1217782	a	145	125	333	9704128379
1312243	ra	129	167	333	9704128379
1321242	ro	134	125	333	9704128379

CUS\_CODE CUS\_NAME CUS\_ZIP AGENT\_CODE AGENT\_CODE AGENT\_PHONE

```
-----
```

1542311	s	134	421	333	9704128379
1657399	v	145	231	333	9704128379

24 rows selected.



```
SQL> select *
2   from customer cross join agent;
```

This query also results the values same as the above query which is the old style

## INNER JOINS

### Old-style join :

```
SQL> select *
2   from customer,agent
3   where customer.agent_code=agent.agent_code;
```

CUS_CODE	CUS_NAME	CUS_ZIP	AGENT_CODE	AGENT_CODE	AGENT_PHONE
1132445	w	145	231	231	9885434311
1217782	a	145	125	125	9985707288
1312243	ra	129	167	167	8985291308
1321242	ro	134	125	125	9985707288
1657399	v	145	231	231	9885434311

### NATURAL JOIN:

```
SQL> select *
2   from customer natural join agent;
```

AGENT_CODE	CUS_CODE	CUS_NAME	CUS_ZIP	AGENT_PHONE
231	1132445	w	145	9885434311
125	1217782	a	145	9985707288
167	1312243	ra	129	8985291308
125	1321242	ro	134	9985707288
231	1657399	v	145	9885434311

### JOIN USING

```
SQL> select *
2   from customer join agent using(agent_code);
```

AGENT_CODE	CUS_CODE	CUS_NAME	CUS_ZIP	AGENT_PHONE
231	1132445	w	145	9885434311
125	1217782	a	145	9985707288
167	1312243	ra	129	8985291308
125	1321242	ro	134	9985707288
231	1657399	v	145	9885434311

JOIN ON

SQL&gt; select \*

2 from customer join agent on customer.agent\_code=agent.agent\_code;

CUS_CODE	CUS_NAME	CUS_ZIP	AGENT_CODE	AGENT_CODE	AGENT_PHONE
1132445	w	145	231	231	9885434311
1217782	a	145	125	125	9985707288
1312243	ra	129	167	167	8985291308
1321242	ro	134	125	125	9985707288
1657399	v	145	231	231	9885434311

OUTER JOINLEFT OUTER JOIN:

SQL&gt; select \*

2 from customer left outer join agent on customer.agent\_code=agent.agent\_code;

CUS_CODE	CUS_NAME	CUS_ZIP	AGENT_CODE	AGENT_CODE	AGENT_PHONE
1321242	ro	134	125	125	9985707288
1217782	a	145	125	125	9985707288
1312243	ra	129	167	167	8985291308
1657399	v	145	231	231	9885434311
1132445	w	145	231	231	9885434311
1542311	s	134	421		

6 rows selected.

RIGHT OUTER JOIN

SQL&gt; select \*

2 from customer right outer join agent on customer.agent\_code=agent.agent\_code;

CUS_CODE	CUS_NAME	CUS_ZIP	AGENT_CODE	AGENT_CODE	AGENT_PHONE
1132445	w	145	231	231	9885434311
1217782	a	145	125	125	9985707288
1312243	ra	129	167	167	8985291308
1321242	ro	134	125	125	9985707288
1657399	v	145	231	231	9885434311
				333	9704128379

6 rows selected.

---

FULL OUTER JOIN

```
SQL> select *  
2 from customer full outer join agent on customer.agent_code=agent.agent_code;
```

CUS_CODE	CUS_NAME	CUS_ZIP	AGENT_CODE	AGENT_CODE	AGENT_PHONE
1321242	ro	134	125	125	9985707288
1217782	a	145	125	125	9985707288
1312243	ra	129	167	167	8985291308
1657399	v	145	231	231	9885434311
1132445	w	145	231	231	9885434311
1542311	s	134	421	333	9704128379

7 rows selected.

**Note:**

*We can use USING clause for the outer joins instead of ON.*

*If ON condition is used the output includes the same columns in two tables twice in the result*

*If USING clause is used the column appears once in the result*

**EXERCICISE**

Consider the following relational schema & instances:

Sailors (sid: integer, sname: string, rating: integer, age: real)

Boats (bid: integer, bname: string)

Reserves (sid: integer, bid: integer, day: date)

**SAILORS**

sid	sname	rating	age
22	dinesh	7	45.0
29	bramha	1	33.0
31	lokesh	8	55.5
32	akash	8	25.5
58	ramesh	10	35.0
64	hari	7	35.0
71	Mahesh	10	16.0
74	hari	9	35.0
85	abhi	3	25.5
95	boby	3	63.5

**RESERVES**

sid	bid	day
22	101	10-oct-98
22	102	10-oct-98
22	103	10-oct-98
22	104	10-jul-98
31	102	11-oct-98
31	103	11-jun-98
31	104	11-dec-98
64	101	9-may-98
64	102	9-aug-98
74	103	9-aug-98

**BOATS**

bid	bname	color
101	interlake	blue
102	interlake	red
103	clipper	green
104	marine	red

Write the SQL queries and their corresponding results for the following:

1. Find the names and ages of all sailors
2. Find all sailors with a rating above 7
3. Find the names of sailors who have reserved boat number 103
4. Find the sid's of sailors who have reserved a red boat
5. Find the names of sailors who have reserved a red boat
6. Find the colors of boats reserved by lokesh
7. Find the names of sailors who have reserved at least one boat
8. Find the name and age of sailors whose name begin with 'b' and ends with 'y' and has at least three characters.

#### UNION, INTERSECT, MINUS

9. Find the names of sailors who have reserved a red or a green boat
10. Find the names of sailors who have reserved both a red and a green boat
11. Find the sid's of all sailors who have reserved red boats but not green boats
12. Find all sid's of sailors who have a rating of 10 or reserved boat 104

#### NESTED QUERIES:

13. Find the names of sailors who have reserved boat 103
14. Find the names of sailors who have reserved a red boat

#### CORRELATED NESTED QUERIES:

15. Find the names of sailors who have reserved boat number 103

#### SET COMPARISON OPERATORS:

16. Find sailors whose rating is better than some sailor called hari
17. Find the sailor with the highest rating
18. Find the name of sailors who have reserved both a red and a green boat
19. Find the names of sailors who have reserved all boats

#### AGGREGATE OPERATORS:

20. Find the average age of sailors with a rating of 10
21. Find the name and age of the oldest sailor
22. Count the number of sailors
23. Count the number of different sailor names
24. Find the names of sailors who are older than the oldest sailor with a rating of 10

#### GROUP BY CLAUSE:

25. Find the age of the youngest sailor for each rating level
26. Find the age of the youngest sailor who is eligible to vote(i.e., is at least 18 years old) for each rating level with at least two such sailors
27. For each red boat find the number of reservations for this boat
28. Find the average age of sailors for each rating level that has at least two sailors

1. Find the names and ages of all sailors

```
SQL> select s.sname,s.age
```

```
2 from sailors s;
```

SNAME	AGE
dinesh	45
bramha	33
lokesh	55.5
akash	25.5
ramesh	35
hari	35
mahesh	16
hari	35
abhi	25.5
boby	63.5

10 rows selected.

The same query with DISTINCT clause results different.

```
SQL> select distinct s.sname, s.age
```

```
2 from sailors s;
```

SNAME	AGE
mahesh	16
boby	63.5
abhi	25.5
bramha	33
ramesh	35
hari	35
akash	25.5
dinesh	45
lokesh	55.5

9 rows selected.

2. Find all sailors with a rating above 7

```
SQL> select *
```

```
2 from sailors;
```

SID	SNAME	RATING	AGE
22	dinesh	7	45
29	bramha	1	33
31	lokesh	8	55.5
32	akash	8	25.5
58	ramesh	10	35
64	hari	7	35
71	mahesh	10	16
74	hari	9	35
85	abhi	3	25.5
95	boby	3	63.5

10 rows selected.

3. Find the names of sailors who have reserved boat number 103

```
SQL> select s.sname
  2   from sailors s, reserves r
  3   where s.sid=r.sid and r.bid=103;
```

SNAME

```
-----
dinesh
lokesh
hari
```

4. Find the sid's of sailors who have reserved a red boat

```
SQL> select r.sid
  2   from boats b, reserves r
  3   where b.bid=r.bid and b.color='red';
```

SID

```
-----
  22
  22
  31
  31
  64
```

5. Find the names of sailors who have reserved a red boat

```
SQL> select s.sname
  2   from sailors s, reserves r, boats b
  3   where s.sid=r.sid and r.bid =b.bid and b.color='red';
```

SNAME

```
-----
dinesh
dinesh
lokesh
lokesh
hari
```

6. Find the colors of boats reserved by lokesh

```
SQL> select b.color
  2   from sailors s, reserves r, boats b
  3   where s.sid=r.sid and r.bid =b.bid and s.sname='lokesh';
```

COLOR

```
-----
red
green
red
```

7. Find the names of sailors who have reserved at least one boat

```
SQL> select s.sname
  2   from sailors s, reserves r
  3   where s.sid=r.sid;
```

SNAME

-----

dinesh  
dinesh  
dinesh  
dinesh  
lokesh  
lokesh  
lokesh  
hari  
hari  
hari

10 rows selected.

8. Find the name and age of sailors whose name begin with 'b' and ends with 'y' and has at least three characters.

```
SQL> select s.sname, s.age
  2   from sailors s
  3   where s.sname like 'b_%y';
```

SNAME	AGE
boby	63.5

#### UNION, INTERSECT, MINUS

9. Find the names of sailors who have reserved a red or a green boat

```
SQL> select s.sname
  2   from sailors s, reserves r, boats b
  3   where s.sid=r.sid and r.bid=b.bid and b.color='red'
  4   union
  5   select s2.sname
  6   from sailors s2, reserves r2, boats b2
  7   where s2.sid=r2.sid and r2.bid=b2.bid and b2.color='green';
```

SNAME

-----

dinesh  
hari  
lokesh



10. Find the names of sailors who have reserved both a red and a green boat

```
SQL> select s.sname
  2   from sailors s, reserves r, boats b
  3   where s.sid=r.sid and r.bid=b.bid and b.color='red'
  4   intersect
  5   select s2.sname
  6   from sailors s2, reserves r2, boats b2
  7   where s2.sid=r2.sid and r2.bid=b2.bid and b2.color='green';
```

SNAME

```
-----
dinesh
hari
lokesh
```

11. Find the sid's of all sailors who have reserved red boats but not green boats

```
SQL> select s.sid
  2   from sailors s, reserves r,boats b
  3   where s.sid=r.sid and r.bid=b.bid and b.color='red'
  4   minus
  5   select s2.sid
  6   from sailors s2, reserves r2,boats b2
  7   where s2.sid=r2.sid and r2.bid=b2.bid and b2.color='green';
```

SID

```
-----
  64
```

12. Find all sid's of sailors who have a rating of 10 or reserved boat 104

```
SQL> select s.sid
  2   from sailors s
  3   where s.rating=10
  4   union
  5   select r.sid
  6   from reserves r
  7   where r.bid=104;
```

SID

```
-----
  22
  31
  58
  71
```

NESTED QUERIES:

13. Find the names of sailors who have reserved boat 103

```
SQL> select s.sname
2   from sailors s
3   where s.sid in ( select r.sid
4                     from reserves r
5                     where r.bid=103);
```

SNAME

-----

dinesh  
lokesh  
hari

14. Find the names of sailors who have reserved a red boat

```
SQL> select s.sname
2   from sailors s
3   where s.sid in ( select r.sid
4                     from reserves r
5                     where r.bid in (select b.bid
6                                     from boats b
7                                     where b.color='red'));
```

SNAME

-----

dinesh  
lokesh  
hari

CORRELATED NESTED QUERIES:

15. Find the names of sailors who have reserved boat number 103

```
SQL> select s.sname
2   from sailors s
3   where exists ( select *
4                 from reserves r
5                 where r.bid=103 and r.sid=s.sid);
```

SNAME

-----

dinesh  
lokesh  
hari

SET COMPARISON OPERATORS:

16. Find sailors whose rating is better than some sailor called hari

```
SQL> select s.sid
2   from sailors s
3   where s.rating >any (select s2.rating
4                        from sailors s2
5                        where s2.sname='hari');
```

```
   SID
-----
    58
    71
    74
    31
    32
```

17. Find the sailor with the highest rating

```
SQL> select s.sid
2   from sailors s
3   where s.rating >=all(select s2.rating
4                        from sailors s2)
```

```
   SID
-----
    58
    71
```

18. Find the name of sailors who have reserved both a red and a green boat

```
SQL> select s.sname
2   from sailors s, reserves r,boats b
3   where s.sid=r.sid and r.bid=b.bid and b.color='red'
4   and s.sid in( select s2.sid
5                  from sailors s2,boats b2,reserves r2
6                  where s2.sid=r2.sid and r2.bid=b2.bid and b2.color='green');
```

```
SNAME
-----
dinesh
dinesh
lokesh
lokesh
```

19. Find the names of sailors who have reserved all boats

```
SQL> select s.sname
2   from sailors s
3   where not exists(select b.bid
4                       from boats b
5                       where not exists(select r.bid
6                                       from reserves r
7                                       where r.bid=b.bid and r.sid=s.sid));
```

SNAME

```
-----
dinesh
```

### AGGREGATE OPERATORS:

20. Find the average age of sailors with a rating of 10

```
SQL> select avg(s.age)
2   from sailors s
3   where s.rating=10;
```

AVG(S.AGE)

```
-----
25.5
```

21. Find the name and age of the oldest sailor

```
SQL> select s.sname,s.age
2   from sailors s
3   where s.age=(select max(s2.age)
4                 from sailors s2);
```

SNAME            AGE

```
-----
boby            63.5
```

22. Count the number of sailors

```
SQL> select count(*)
2   from sailors s;
```

COUNT(\*)

```
-----
10
```

23. Count the number of different sailor names

```
SQL> select count(distinct s.sname)
2   from sailors s;
```

COUNT(DISTINCTS.SNAME)

```
-----
9
```

24. Find the names of sailors who are older than the oldest sailor with a rating of 10

```
SQL> select s.sname
2   from sailors s
3   where s.age>(select max(s2.age)
4             from sailors s2
5             where s2.rating=10);
```

(or)

```
SQL> select s.sname
2   from sailors s
3   where s.age> all (select s2.age
4             from sailors s2
5             where s2.rating=10);
```

SNAME

```
-----
dinesh
lokesh
boby
```

#### GROUP BY CLAUSE:

25. Find the age of the youngest sailor for each rating level

```
SQL> select s.rating, min(s.age)
2   from sailors s
3   group by s.rating;
```

```
RATING MIN(S.AGE)
-----
1          33
8          25.5
7          35
3          25.5
10         16
9          35
```

6 rows selected.

26. Find the age of the youngest sailor who is eligible to vote(i.e., is at least 18 years old) for each rating level with at least two such sailors

```
SQL>select s.rating,min(s.age) as minimumage
2   from sailors s
3   where s.age>=18
4   group by s.rating
5   having count(*)>1
```

## RATING MINIMUMAGE

8	25.5
7	35
3	25.5

27. For each red boat find the number of reservations for this boat

```
SQL> select b.bid,count(*) as reservationcount
2   from boats b,reserves r
3   where r.bid=b.bid and b.color='red'
4   group by b.bid;
```

## BID RESERVATIONCOUNT

102	3
104	2

28. Find the average age of sailors for each rating level that has at least two sailors

```
SQL> select s.rating, avg(s.age) as average
2   from sailors s
3   group by s.rating
4   having count(*)>1;
```

## RATING AVERAGE

8	40.5
7	40
3	44.5
10	25.5

---

### SQL FUNCTIONS

The data in database are the basis of critical business information. Generating information from data often requires many data manipulations. Sometimes such data manipulation involves the decomposition of data elements.

SQL functions are very useful tools. There are many types of SQL functions, such as arithmetic, trigonometric, string, date, and time functions.

NOTE: DUAL is oracles pseudo table used only for case where a table is not really needed

#### DATE FUNCTIONS

- SYSDATE: Returns today's date.

```
SQL> select sysdate
2    from dual;
```

SYSDATE

```
-----
04-MAR-12
```

- ADD\_MONTHS: Returns date after adding the number of months specified in the function.  
Syntax: ADD\_MONTHS(DATE\_VALUE,N)

```
SQL> select add_months(sysdate,4)
2    from dual;
```

ADD\_MONTH

```
-----
04-JUL-12
SQL> select add_months(sysdate,4)"addmonths"
2    from dual;
```

addmonths

```
-----
04-JUL-12
SQL> select add_months(sysdate,4) as addmonths
2    from dual;
```

ADDMONTHS

```
-----
04-JUL-12
```

- LAST\_DAY: Returns the date of the last day of the month given in a date.  
Syntax: LAST\_DAY(DATE\_VALUE)

```
SQL> select sysdate,last_day(sysdate)"lastday"
2    from dual;
```

SYSDATE lastday

```
-----    -----
04-MAR-12  31-MAR-12
```

NUMERIC FUNCTIONS:

Numeric functions can be grouped in many different ways, such as algebraic, trigonometric and logarithmic. The following are selected group of numeric functions.

- **ABS:** Returns the absolute value of a number  
Syntax: ABS(NUMERIC\_VALUE)

```
SQL> select 1.93,-1.93, abs (1.93), abs (-1.93)
2   from dual;
```

1.93	-1.93	ABS (1.93)	ABS (-1.93)
1.93	-1.93	1.93	1.93

- **POWER:** Returns m raised to the n<sup>th</sup> power. n must be an integer, else an error is returned .  
Syntax: POWER(m,n)

Eg. Find 5<sup>2</sup>

```
SQL> select power(5,2)
2   from dual;
```

```
POWER(5,2)
-----
25
```

- **ROUND:** Returns n, rounded to m places to the right of a decimal point. If m is omitted , n is rounded to zero places  
Syntax: ROUND(n,m)

```
SQL> select round(15.193,1)
2   from dual;
ROUND(15.193,1)
```

```
-----
15.2
```

```
SQL> select round(15.193)
2   from dual;
```

```
ROUND(15.193)
-----
15
```

- **SQRT:** Returns square root of n.  
Syntax: SQRT(n)

Eg. Find the square root of 625

```
SQL> select sqrt(625)
2   from dual;
```

```
SQRT(625)
-----
25
```



- **MOD:** Returns the remainder of a first number divided by second number passed a parameter. If the second number is zero, the result is the same as the first number.

Syntax: MOD(m,n)

Eg. Find 15 mod 4

```
SQL> select mod(15,4)
2   from dual;
```

```
MOD(15,4)
-----
3
```

- **FLOOR:** Returns the largest integer value that is equal to or less than a number.

Syntax: FLOOR(n)

Eg. Find floor(24.8)

```
SQL> select floor(24.8)
2   from dual;
```

```
FLOOR(24.8)
-----
24
```

- **CEIL:** Returns the smallest integer value that is greater than or equal to a number

Syntax: CEIL(n)

Eg. Find ceil(24.8)

```
SQL> select ceil(24.8)
2   from dual;
```

```
CEIL(24.8)
-----
25
```

#### STRING FUNCTIONS:

- **CONCATENATION:** (||) Concatenates data from two different character columns and returns a single column
- **LOWER:** Returns char , with all letters in lower case.  
Syntax: LOWER(char)

```
SQL> select lower('KNREDDY') as lower
2   from dual;
```

```
LOWER
-----
knreddy
```

- INITCAP: returns a string with the first letter of each word in upper case  
Syntax: INITCAP(char)

```
SQL> select initcap('nageswarareddy') as initcap
2   from dual;
```

```
INITCAP
-----
Nageswarareddy
```

- UPPER: Returns a string with all letters forced to upper  
Syntax: UPPER(char)

```
SQL> select upper('knreddy') as upper
2   from dual;
```

```
UPPER
-----
KNREDDY
```

- SUBSTR: Returns a portion of characters, beginning at character m, and going upto characters n. If n is omitted, the result returned is upto the last character in the string. The first position of character is 1.

Syntax: SUBSTR(<string>,<startposition>,<length>)

Where     string - source string  
          Start position – position for extraction  
          Length – number of characters to extract

```
SQL> select substr('databasemanagementsystems',5,4)
2   from dual;
```

```
SUBS
-----
base
```

```
SQL> select substr('databasemanagementsystems',5) as substr
2   from dual;
```

```
SUBSTR
-----
basemanagementsystems
```

- ASCII: Returns the number code that represents the specified character. If more than one character is entered , the function will return the value for the first character and ignore all of the characters after the first.  
Syntax: ASCII(<single- character>)

```
SQL> select ascii('a')
2   from dual;
```

```
ASCII('A')
-----
```

```
97
```

```
SQL> select ascii('A')
2   from dual;
```

```
ASCII('A')
-----
```

```
65
```

```
SQL> select ascii('APPLE')
2   from dual;
```

```
ASCII('APPLE')
-----
```

```
65
```

- **LENGTH:** Returns the length of a word  
Syntax: LENGTH(word)

```
SQL> select length('database')
2   from dual;
```

```
LENGTH('DATABASE')
-----
```

```
8
```

```
SQL> select length('data base')
2   from dual;
```

```
LENGTH('DATABASE')
-----
```

```
9
```

- **LTRIM:** Removes characters from the left of character with initial characters removed upto the first character not in set.  
Syntax: LTRIM(char,set)

```
SQL> select ltrim('dbms','d')
2   from dual;
```

```
LTR
----
```

```
bms
```

```
SQL> select ltrim('dbms','d')
2   from dual;
```

```
LTR
```

```
-----
```

```
bms
```

```
SQL> select ltrim('dbms','d'b')
2   from dual;
```

```
LT
```

```
---
```

```
ms
```

- **RTRIM:** Returns char, with final characters removed after the last character not in the set.  
Syntax: RTRIM(char,set)

```
SQL> select rtrim('dbms','s')
2   from dual;
```

```
RTR
```

```
-----
```

```
dbm
```

```
SQL> select rtrim('dbms','s'm')
2   from dual;
```

```
RT
```

```
---
```

```
Db
```

- **TRIM:** Removes all specified characters either from the beginning or the ending of a string  
Syntax: TRIM([leading|trailing|both{ <trim-characters> from}] <string>)  
Where leading -remove trim string from the front of the string  
Trailing -remove trim string from end of string  
Both -remove trim string from the front and end of string  
If none of the above option is chosen, the TRIM function will remove trim string from both the front and end of string.  
Trim\_character is the character that will be removed from string. If this parameter is omitted, the trim function will remove all leading and trailing spaces from string  
String - string to trim

```
SQL> select trim(' dbms ')
2   from dual;
```

```
TRIM
```

```
-----
```

```
dbms
```

```
SQL> select trim(leading 'x' from 'xxxxdbmsxxxx') as leading
2   from dual;
```

LEADING

```
-----
dbmsxxxx
```

```
SQL> select trim(trailing 'x' from 'xxxxdbmsxxxx') as trailing
2   from dual;
```

TRAILING

```
-----
xxxxdbms
```

```
SQL> select trim(both 'x' from 'xxxxdbmsxxxx')
2   from dual;
```

TRIM

```
-----
dbms
```

```
SQL> select trim(both '1' from '1123dbms2311') as both
2   from dual;
```

BOTH

```
-----
23dbms23
```

- **LPAD:** Return char-1 left padded to length n with the sequence of characters specified in char-2. If char-2 is not specified oracle uses blanks by default  
Syntax: LPAD(char-1,n,char-2)

```
SQL> select lpad('dbms',15,'@') as lpad
2   from dual;
```

LPAD

```
-----
@@@@@@@@@@@@@dbms
```

- **RPAD:** Return char-1 right padded to length n with the sequence of characters specified in char-2. If char-2 is not specified oracle uses blanks by default  
Syntax: LPAD(char-1,n,char-2)

```
SQL> select rpad('dbms',15,'@') as rpad
2   from dual;
```

RPAD

```
-----
dbms@@@@@@@@@@@@@
```

## PL/SQL

- **ORACLE** is a relational data base.
- The language used to access a relational data base is **SQL**.
- SQL is a flexible, efficient language, with features designed to manipulate and examine relational data.
- SQL is a fourth generation language. SQL is a nonprocedural language.
- **Nonprocedural means what rather than how.**

While SQL is the natural language of the DBA, it does not have any procedural capabilities such as looping & branching nor does it have any conditional checking capabilities vital for data testing before storage.

### **FOR ALL THIS, ORACLE PROVIDES PL/SQL**

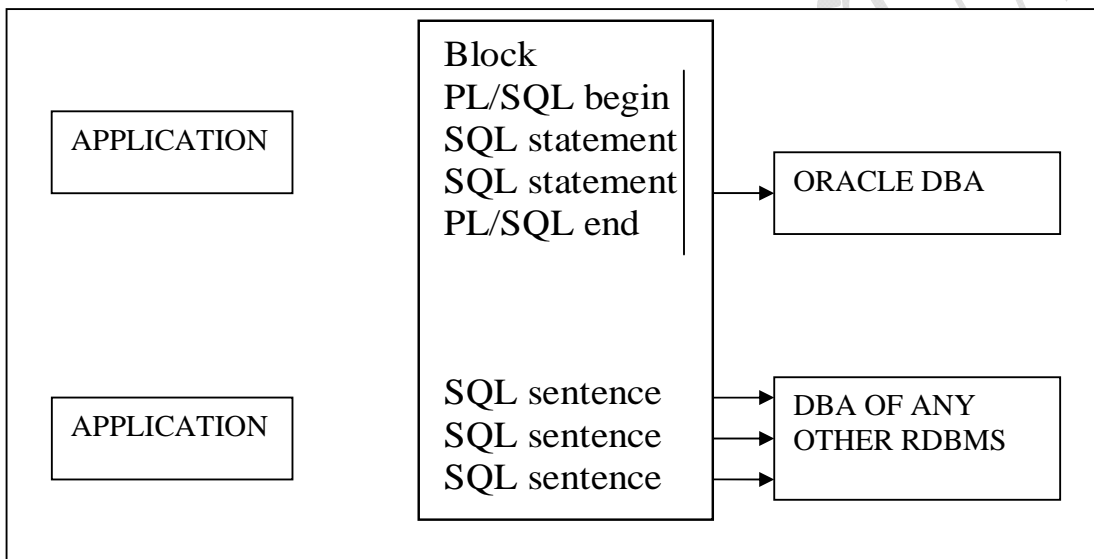
#### INTRODUCTION:-

- PL/SQL (Procedural Language/SQL) is a procedural extension of oracle-SQL.
- PL/SQL is a sophisticated programming language used to access an ORACLE data base.
- PL/SQL is integrated with the database server so that the PL/SQL code can be processed quickly and efficiently.
- The PL/SQL language includes object oriented programming techniques such as encapsulation, function overloading, and information hiding (all but inheritance).
- PL/SQL is commonly used to write data-centric programs to manipulate data in an Oracle database.
- PL/SQL bridges the gap between database technology and procedural programming languages. It can be thought of as a development tool that extends the facilities of Oracle's SQL database language. Via PL/SQL you can insert, delete, update and retrieve table data as well as use procedural techniques such as writing loops or branching to another block of code.
- PL/SQL is really an extension of SQL. It allows you to use all the SQL data manipulation statements as well as the cursor control operations and transaction processing. PL/SQL blocks can contain any number of SQL statements. It allows you to logically group a number of SQL sentence and pass them to the DBA as a single block.
- The basic construct in PL/SQL is a block. Blocks allow designers to combine logically related (SQL) statements into units.
- In a block, constants and variables can be declared and variables can be used to store query results.
- Statements in a PL/SQL block include SQL statements, control structures (loops), condition statements (if-then-else), exception handling, and calls of other PL/SQL blocks.
- PL/SQL blocks that specify procedures and functions can be grouped into packages.
- Another important feature of PL/SQL is that it offers a mechanism to process query results in a tuple-oriented way, that is, one tuple at a time. For this cursors are used. A cursor basically is a pointer to a query result and is used to record attribute values of selected tuples into variables. A cursor typically is used in combination with a loop construct such that each tuple read by the cursor can be processed individually.
- In summary the major goals of PL/SQL are to:
  - Increase the expressiveness of SQL
  - Process query results in a tuple oriented way
  - Optimize combined SQL statements
  - Develop modular database application programs
  - Reuse program code
  - Reduce the cost for maintaining and changing application

**PERFORMANCE:**

Without PL/SQL, DBA has to process SQL statements one at a time. This results in calls being made to the DBA each time an SQL statement is executed. It slows down table data processing considerably, especially when several users are firing SQL statements at the same time, as done in a multi – user environment. Each time an SQL statement is fired, it causes traffic to originate on the network and places quite a bit of overhead on the hardware.

With PL/SQL, an entire block of statements can be sent to the RDBMS engine at any one time. This dramatically reduces the communication between the developed software and the DBA (i.e. it reduces the traffic on the network)

**PERFORMANCE IMPROVEMENT:**

It is obvious that when the DBA gets SQL code as a single block, it exercises this code faster than if it got the code one sentence at a time. Hence, there is a definite improvement in the performance time of the DBA.

PL/SQL can also be used in SQL\*FORMS. Its procedural capabilities can be used for writing complex triggers that will validate data before it is placed in the table. Here, the trigger code will be treated by the DBA as a block and processed in the same manner. Via PL/SQL you can do all sorts of calculations, quickly and efficiently without the use of the DBA. This considerably improves transaction performance.

**PORTABILITY:**

Applications written in PL/SQL are portable to any computer and operating system, where ORCALE is operational. Hence, PL/SQL code blocks written for a DOS version of Oracle will run on its UNIX version, without any modifications made to it.

## USING PL/SQL BLOCKS IN THE SQL\*PLUS ENVIRONMENT

PL/SQL can also be run from within the SQL\*plus environment. After invoking SQL\*plus, you can run a PL/SQL block in any one of the following ways:

1. Key it in directly using the SQL \* PLUS editor, then run it.
2. Load it from a previously created ASCII file.

Either of the 2 methods require you to be within the SQL\*PLUS environment first.

All PL/SQL blocks starts with the reserved word DECLARE or if the block has no declaration part, it will start with the reserved word BEGIN.

Typing either of these words at the SQL \* PLUS prompt (SQL>) informs the SQL \* PLUS code (held in RAM) to do the following:

1. Clear the SQL buffer,
2. Enter into INPUT mode,
3. Ignore semicolons, i.e. the SQL statement terminator.

You can then key in your entire PL/SQL block and use the normal SQL \* PLUS editing features to edit the block. Terminating your PL/SQL block with a period (.) stores the block in the SQL buffer. If you terminate the PL/SQL block with a slash (/), it causes the PL/SQL block to be stored in the SQL buffer and then be executed.

If the SQL buffer contains an SQL statement or a PL/SQL block and you want to run it, simply type run or / (slash) at the SQL\*PLUS prompt. When the SQL statement or the PL/SQL block has finished running, you are returned to the SQL\*PLUS prompt i.e. SQL>

### The PL/SQL SYNTAX:

The character set:

The basic character set includes the following:

- Uppercase alphabets { A – Z }
- Lowercase alphabets { a – z }
- Numerals { 0 – 9 }
- Symbols: ( ) + - \* / < > = ! ; : . ‘ @ % , “ # \$ ^ & \_ \ { } ? [ ]

Words used in a PL/SQL block are called lexical units. You can freely insert blank spaces between lexical units in a PL/SQL block. The spaces have no effect on the PL/SQL block.

The ordinary symbols used in PL/SQL blocks are: ( ) + - \* / < > = ; % ‘ “ [ ] :

Compound symbols used in PL/SQL blocks are: < > != ~ = ^ = < = > = := \*\* .. || <<>>

**COMMENTS:** The comment can have 2 forms:

The comment line begins with a double hyphen (--). The entire line will be treated as a comment. The comment line begins with a slash followed by an asterisk (/\*) till the occurrence of an asterisk



followed by a slash (\*). All lines within, are treated as comments.

This form of specifying comments can be used to span across multiple lines, which means that you can use this to surround a section of a PL/SQL block that you temporarily do not want to execute.

**NOTE:** Comments cannot be nested.

### **PL/SQL DATA TYPES:**

Both PL/SQL and Oracle have their foundations in SQL. Most PL/SQL data types are native to Oracle's data dictionary. Hence, there is a very easy integration of PL/SQL code with the Oracle RDBMS.

**NUMBER** for storing numeric data

**CHAR** for storing character data

**DATE** for storing date and time data

**BOOLEAN** for storing TRUE, FALSE or NULL

**%TYPE** declares a variable or constant to have the same data type as that of a previously defined variable or of a column in a table or in a view. When referencing a table, you may name the table and column, or the owner of the table and column.

The %TYPE attribute provides for further integration. PL/SQL can use the %TYPE attribute to declare variables based on definitions of columns in a table. Hence, if a column's attributes change, the variable's attributes will change as well. This provides for data independence, reduces maintenance costs and allows programs to adapt to changes made to the table.

### **WHAT IS THE DIFFERENCE BETWEEN %TYPE AND %ROWTYPE?**

The %TYPE and %ROWTYPE constructs provide data independence, reduces maintenance costs, and allows programs to adapt as the database changes to meet new business needs.

%ROWTYPE is used to declare a **record** with the same types as found in the specified database table, view or cursor.

#### **Example: DECLARE**

```
v_EmpRecord emp%ROWTYPE;
```

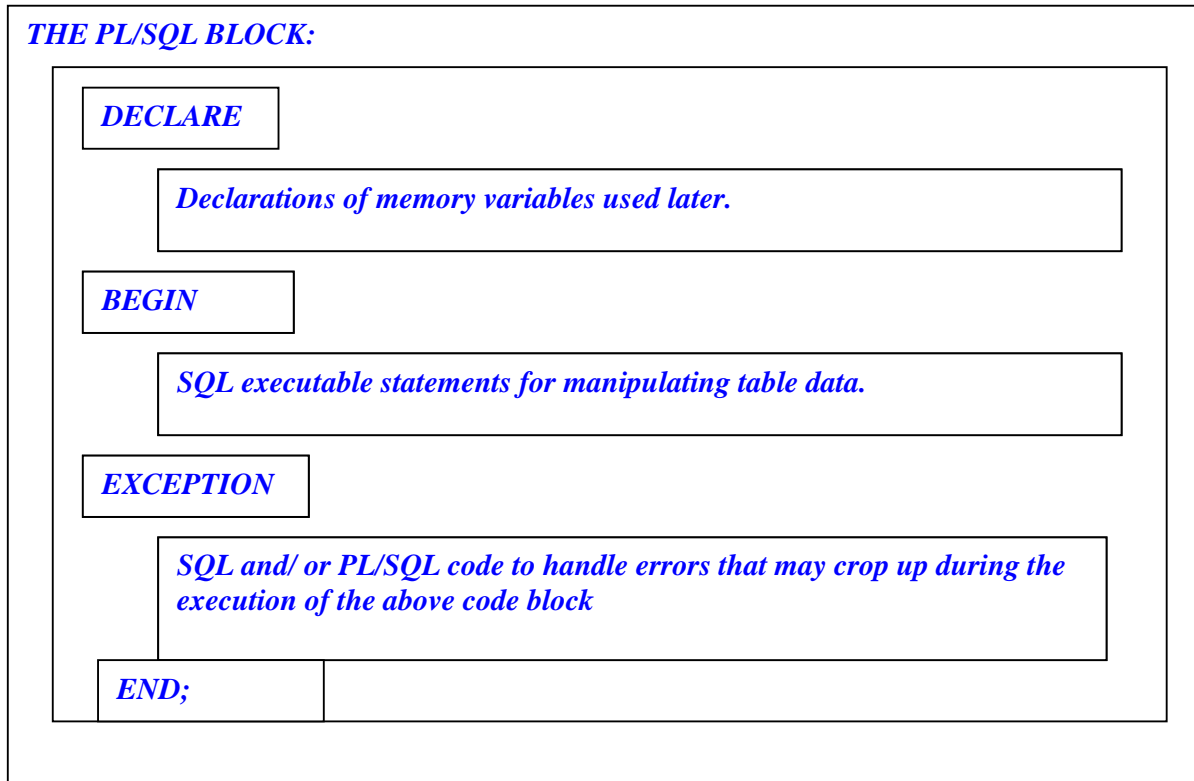
%TYPE is used to declare a **field** with the same type as that of a specified table's column.

#### **Example: DECLARE**

```
v_EmpNo emp.empno%TYPE;
```

### **VARIABLES:**

A variable name must begin with a character and can be followed by a maximum of 29 other characters. Reserved words cannot be used as variable names unless enclosed with in double quotes. Variables must be separated from each other by at least one space or by a punctuation mark. The case is insignificant when declaring variable names. A space cannot be used in variable name.

**UNDERSTANDING THE PL/SQL BLOCK STRUCTURE****AN IDENTIFIER IN THE PL/SQL BLOCK:**

The name of any Oracle object (variable, constant, record, cursor etc.) is known as an identifier. The following laws have to be followed while working with identifiers:

1. An identifier cannot be declared twice in the same block.
2. The same identifier can be declared in two different blocks.
3. If you follow the second law, the two identifiers are unique and any change in one does not affect the other. An identifier can be declared in a sub-block of another sub-block in which case it is local that sub-block alone.

**Example:****DECLARE**

```

    account number(5);
    credit_limit number(9,2);
  
```

**BEGIN****DECLARE**

```

    account char(20);
    new_balance number(9,2);
  
```

**BEGIN**

The identifiers available to this block are

```

        account char(20), credit_limit, new_balance
    END;
    DECLARE
        old_balance number(9,2);
    BEGIN
        /* The identifiers available to this block are
        account number(5), credit_limit, old_balance */
    END;
        /* The identifiers available here are account number(5),
        Credit_limit */
    END;

```

### DISPLAYING USER MESSAGES ON THE SCREEN:

Any programming tool requires a method through which messages can be displayed to the user.

**dbms\_output** is a package that includes a number of procedure & functions that accumulate information in a buffer so that it can be retrieved later. These functions can also be used to display messages to the user.

**put\_line** Put a piece of information in the buffer followed by an end-of-line marker. It can also be used to display message to the user. `put_line` expects only one parameter of character data type. If used to display message, it will be the message string.

To display messages to the user the `SERVEROUTPUT` should be set to ON. `SERVEROUTPUT` is a SQL\*PLUS environment parameter that displays the information passed as a parameter to the `put_line` function.

**Example:** Setting the server output on: **SET SERVEROUTPUT ON;**

#### Example:

```

SQL> BEGIN
  2 EXECUTE IMMEDIATE 'CREATE TABLE X(A DATE)';
  3 END;
  4 /

```

**PL/SQL procedure successfully completed.**

```

SQL> DESC X;
Name                               Null?  Type
-----
A                                     DATE

```

**NOTE:** The DDL statement in quotes should not be terminated with a semicolon.

**CONDITIONAL CONTROL IN PL/SQL:**

In PL/SQL, the if statement allows you to control the execution of a block of code. In PL/SQL you can use the IF – THEN – ELSIF – ELSE – END IF statements in code blocks that will allow you to write specific conditions under which a specific block of code will be executed.

**Syntax:** IF < condition> THEN  
    <action>  
    ELSIF <condition>  
        <action>  
    ELSE <action>  
    END IF;

**Iterative Control:**

This is the ability to repeat or skip sections of a code block.

A loop repeats a sequence of statements. You have to place the keyword loop before the first statement in the sequence of statements that you want repeated and the keywords end loop immediately after the last statement in the sequence. Once a loop begins to run, it will go on forever. Hence loops are always accompanied by a conditional statement that keeps control on the number of times the loop is executed.

You can build user defined exits from a loop, where required.

**THE WHILE LOOP:**

**Syntax:** WHILE <condition>  
    LOOP <action>  
    END LOOP;

**THE FOR LOOP:**

**Syntax:** FOR variable IN [REVERSE] start..end  
    LOOP  
        <action>  
    END LOOP;

**THE GOTO STATEMENT:**

The go to statement allows you to change the flow control within a PL/SQL block.

The entry point of the block is defined with in <<>> as shown in the above example.

**Syntax:** GOTO <action>  
  
    <<action>>  
        SQL statement;  
        SQL statement;

- THE PL/SQL BLOCK STRUCTURE:

```
[DECLARE
    <constants>
    <variables>
    <cursors>    ]
BEGIN
    <SQL statements>
    <PL/SQL statements>
[EXCEPTION
    <exception handling  ]
END;
/
```

Writing and executing a PL/SQL program:

```
SQL> ed <filename>
```

Note pad will open with the above command. Type program in the note pad and save

```
SQL>set serveroutput on
```

```
SQL>@<filename>
```

**✚ WRITE A PL/SQL PROGRAM TO ADD TWO NUMBERS**

//File name: add

DECLARE

    a number(5);

    b number(5);

    c number(5);

BEGIN

    a:=&a;

    b:=&b;

    c:=a+b;

    dbms\_output.put\_line('the sum of two numbers(a+b)=||c);

END;

/

OUTPUT:

SQL> @add

Enter value for a: 40

old 6:     a:=&a;

new 6:     a:=40;

Enter value for b: 80

old 7:     b:=&b;

new 7:     b:=80;

the sum of two numbers(a+b)=120

PL/SQL procedure successfully completed.

---

**✚ WRITE A PL/SQL PROGRAM TO FIND LARGEST NUMBER FROM THE GIVEN  
THREE NUMBERS**

```
//file name:largestnum
DECLARE
    a number(5);
    b number(5);
    c number(5);
BEGIN
    a:=&a;
    b:=&b;
    c:=&c;
    if a>b and a>c then
        dbms_output.put_line('the largest number is'||a);
    else if b>c and b>a then
        dbms_output.put_line('the largest number is'||b);
    else
        dbms_output.put_line('the largest number is'||c);
    end if;
end if;
END;
/
```

**OUTPUT:**

```
SQL> @largestnum
Enter value for a: 8
old 6:  a:=&a;
new 6:  a:=8;
Enter value for b: 4
old 7:  b:=&b;
new 7:  b:=4;
Enter value for c: 6
old 8:  c:=&c;
new 8:  c:=6;
the largest number is8
```

PL/SQL procedure successfully completed.

---

**✚ WRITE A PL/SQL PROGRAM FOR CHECKING A NUMBER IS EVEN OR ODD**

```
//file name: evenodd
DECLARE
    num number(5);
    rem number(5);
BEGIN
    num:=&num;
    rem:=mod(num,2);
    if rem=0 then
        dbms_output.put_line('Number'|| num||'is EVEN');
    else
        dbms_output.put_line('Number'|| num||'is ODD');
    end if;
END;
```

/

**OUTPUT:**

```
SQL> ed evenodd
```

```
SQL> @evenodd
```

```
Enter value for num: 123
```

```
old 5:   num:=&num;
```

```
new 5:   num:=123;
```

```
Number123is ODD
```

PL/SQL procedure successfully completed.

```
SQL> /
```

```
Enter value for num: 120
```

```
old 5:   num:=&num;
```

```
new 5:   num:=120;
```

```
Number120is EVEN
```

PL/SQL procedure successfully completed.



---

**✚ WRITE A PL/SQL PROGRAM TO FINDS SUM OF DIGITS OF A GIVEN NUMBER.**

```
//file name:digitssum
DECLARE
    num number(5);
    rem number(5);
    s number(5):=0;
    num1 number(5);
BEGIN
    num:=&num;
    num1:=num;
    while(num>0)
        loop
            rem:=mod(num,10);
            s:=s+rem;
            num:=trunc(num/10);
        end loop;
    dbms_output.put_line('Sum of digits of '||num1||' is: '||s);
END;
/
```

**OUTPUT:**

```
SQL> @digitssum
```

```
Enter value for num: 2315
```

```
old 7:   num:=&num;
```

```
new 7:   num:=2315;
```

```
Sum of digits of 2315 is: 11
```

PL/SQL procedure successfully completed.

---

**WRITE A PL/SQL PROGRAM TO DISPLAY EVEN NUMBERS UPTO CERTAIN NUMBER.**

```
//file name: evennumbers
DECLARE
    num number(5);
    i number(5);
BEGIN
    num:=&num;
    i:=1;
    while(i<=num)
        loop
            if(mod(i,2)=0) then
                dbms_output.put_line(i);
                i:=i+1;
            else
                i:=i+1;
            end if;
        end loop;
END;
/
```

**OUTPUT:**

```
SQL> @evennumbers
Enter value for num: 20
old 5:   num:=&num;
new 5:   num:=20;
2
4
6
8
10
12
14
16
18
20
```

PL/SQL procedure successfully completed.

---

**✚ WRITE A PL/SQL PROGRAM TO CHECK THE GIVEN STRING IS PALINDROME OR  
NOT.**

```
//file name: palindrome
DECLARE
    name varchar2(20);
    temp varchar2(20);
    len number(5);
BEGIN
    name:='&name';
    len:=length(name);
    while len>0
        loop
            temp:=temp||substr(name,len,1);
            len:=len-1;
        end loop;
    dbms_output.put_line('reverse of string is: '||temp);
    if(name=temp) then
        dbms_output.put_line(name||' is palindrome');
    else
        dbms_output.put_line(name||' is not palindrome');
    end if;
END;
/
```

**OUTUT:**

```
SQL> @palindrome
Enter value for name: madam
old 6:   name:='&name';
new 6:   name:='madam';
reverse of string is: madam
madam is palindrome
```

PL/SQL procedure successfully completed.

```
SQL> /
Enter value for name: abcd
old 6:   name:='&name';
new 6:   name:='abcd';
reverse of string is: dcba
abcd is not palindrome
```

PL/SQL procedure successfully completed.

---

**✚ WRITE A PL/SQL PROGRAM TO CHECK THE GIVEN NUMBER IS ARMSTRONG  
OR NOT.**

```
//file name: armstrong
DECLARE
    num number(5);
    rem number(5);
    s number(5):=0;
    num1 number(5);
BEGIN
    num:=&num;
    num1:=num;
    while(num>0)
        loop
            rem:=mod(num,10);
            s:=s+power(rem,3);
            num:=trunc(num/10);
        end loop;
    if(s=num1) then
        dbms_output.put_line(num1||' is armstrong number');
    else
        dbms_output.put_line(num1||' is not armstrong number');
    end if;
END;
/
```

**OUTPUT:**

```
SQL> @armstrong
Enter value for num: 153
old 7:   num:=&num;
new 7:   num:=153;
153 is armstrong number
```

PL/SQL procedure successfully completed.

```
SQL> /
Enter value for num: 125
old 7:   num:=&num;
new 7:   num:=125;
125 is not armstrong number
```

PL/SQL procedure successfully completed.

---

**WRITE A PL/SQL PROGRAM TO GENERATE FIBONACCI SERIES**

```
//file name: Fibonacci
DECLARE
    num number(5);
    f1 number(5);
    f2 number(5);
    f3 number(5);
    i number(5);
BEGIN
    f1:=0;
    f2:=1;
    i:=3;
    num:=&num;
    dbms_output.put_line('the fibonacci series is: ');
    dbms_output.put_line(f1);
    dbms_output.put_line(f2);
    for i in 3..num
    loop
        f3:=f1+f2;
        dbms_output.put_line(f3);
        f1:=f2;
        f2:=f3;
    end loop;
END;
/
```

**OUTPUT:**

```
SQL> @Fibonacci
```

```
Enter value for num: 10
```

```
old 11:    num:=&num;
```

```
new 11:    num:=10;
```

```
the fibonacci series is:
```

```
0
1
1
2
3
5
8
13
21
34
```

PL/SQL procedure successfully completed.

**✚ WRITE A PL/SQL PROGRAM TO PRINT THE MULTIPLICATION TABLE.**

```
//file name: multiplication
DECLARE
    i number(5);
    n number(5);
BEGIN
    n:=&n;
    for i in 1..10 loop
        dbms_output.put_line( n || ' * ' || i || ' = ' || n*i);
    end loop;
END;
/
```

**OUTPUT:**

```
SQL> @multiplication
```

```
Enter value for n: 25
```

```
old 5:    n:=&n;
```

```
new 5:    n:=25;
```

```
25 * 1 = 25
```

```
25 * 2 = 50
```

```
25 * 3 = 75
```

```
25 * 4 = 100
```

```
25 * 5 = 125
```

```
25 * 6 = 150
```

```
25 * 7 = 175
```

```
25 * 8 = 200
```

```
25 * 9 = 225
```

```
25 * 10 = 250
```

```
PL/SQL procedure successfully completed.
```

**WRITE A PL/SQL PROGRAM TO CONVERT FARENHEIT TO CELSIUS**

```
//file name:Celsius
DECLARE
    fah number(6,2);
    cels number(6,2);
    zero_error exception;
BEGIN
    fah:=&fah;
    if fah<=0 then
        raise zero_error;
    end if;
    cels:=fah-32*(9/5);
    dbms_output.put_line('celsius='||cels);
EXCEPTION
when zero_error then
dbms_output.put_line('invalid number');
END;
/
```

**OUTPUT:**

```
SQL> @celsius
Enter value for fah: 90
old 6:   fah:=&fah;
new 6:   fah:=90;
celsius=32.4
```

PL/SQL procedure successfully completed.

---

**✚ WRITE A PL/SQL PROGRAM TO FIND OUT THE REVERSE OF A NUMBER**

```
//file name:reverse
DECLARE
    num number(10);
    num1 number(10);
    rem number(10);
    s number(10);
BEGIN
    num:=&num;
    num1:=num;
    s:=0;
    while num>0 loop
        rem:=mod(num,10);
        s:=s*10+rem;
        num:=trunc(num/10);
    end loop;
    dbms_output.put_line('The reverse of the number'||num1||' is '||s);
END;
/
```

**OUTPUT:**

```
SQL> @reverse
Enter value for num: 12345
old 7:   num:=&num;
new 7:   num:=12345;
The reverse of the number12345 is 54321
```

PL/SQL procedure successfully completed.

```
SQL> /
Enter value for num: 123456789
old 7:   num:=&num;
new 7:   num:=123456789;
The reverse of the number123456789 is 987654321
```

PL/SQL procedure successfully completed.



---

**✚ WRITE A PL/SQL PROGRAM TO CALCULATE SIMPLE INTEREST**

```
// file name: interest
DECLARE
    p number(10);
    t number(10);
    r number(10);
    si number(10);
BEGIN
    p:=&p;
    t:=&t;
    r:=&r;
    si:=(p*t*r)/100;
    dbms_output.put_line('simple interest is ' || si);
END;
```

**OUTPUT:**

```
SQL> @interest
Enter value for p: 5000
old 7:  p:=&p;
new 7:  p:=5000;
Enter value for t: 12
old 8:  t:=&t;
new 8:  t:=12;
Enter value for r: 2
old 9:  r:=&r;
new 9:  r:=2;
simple interest is 1200
```

PL/SQL procedure successfully completed.

```
SQL> /
Enter value for p: 100000
old 7:  p:=&p;
new 7:  p:=100000;
Enter value for t: 24
old 8:  t:=&t;
new 8:  t:=24;
Enter value for r: 2
old 9:  r:=&r;
new 9:  r:=2;
simple interest is 48000
```

PL/SQL procedure successfully completed.

---

**WRITE A PL/SQL PROGRAM TO CHECK WHETHER THE GIVEN NUMBER IS PRIME OR NOT**

```
//file name: prime
DECLARE
    a number;
    c number:=0;
    i number;
BEGIN
    a:=&a;
    for i in 1..a
    loop
        if mod(a,i)=0 then
            c:=c+1;
            end if;
        end loop;
    if c=2 then
        dbms_output.put_line(a ||'is a prime number');
    else
        dbms_output.put_line(a ||'is not a prime number');
    end if;
END;
/
```

**OUTPUT:**

```
SQL> @prime
Enter value for a: 11
old 6:  a:=&a;
new 6:  a:=11;
11 is a prime number
```

PL/SQL procedure successfully completed.

```
SQL> /
Enter value for a: 25
old 6:  a:=&a;
new 6:  a:=25;
25 is not a prime number
```

PL/SQL procedure successfully completed.

**✚ WRITE A PL/SQL PROGRAM WHICH ACCEPTS THE STUDENTS NAME, NUMBER AND HIS MARKS AND DISPLAY TOTAL MARKS & GRADE.**

```
//file name : student
DECLARE
    sno number(10);
    name varchar2(30);
    sub1 number(10);
    sub2 number(10);
    sub3 number(10);
    tot number(10);
    aveg number(10);
BEGIN
    sno:=&Student_Number;
    name := '&Student_Name';
    sub1 := &subject1;
    sub2 := &subject2;
    sub3 := &subject3;
    tot:=sub1+sub2+sub3;
    aveg:=tot/3;
    dbms_output.put_line('_____');
    dbms_output.put_line('Student Number   :'||sno);
    dbms_output.put_line('Student Name     :'||name);
    dbms_output.put_line('Student Sub1 Marks :'||sub1);
    dbms_output.put_line('Student Sub2 Marks :'||sub2);
    dbms_output.put_line('Student Sub3 Marks :'||sub3);
    if sub1>=40 and sub2>=40 and sub3>=40 then
        if aveg>=70 then
            dbms_output.put_line('Student got Distinction');
        elsif aveg>=60 then
            dbms_output.put_line('Student got First Class');
        elsif aveg>=50 then
            dbms_output.put_line('Student got Second Class');
        elsif aveg>=40 then
            dbms_output.put_line('Student got Third Class');
        end if;
    else
        dbms_output.put_line('Student is Failed');
    end if;
    dbms_output.put_line('_____');
END;
/
```

OUTPUT:

SQL> @student

Enter value for student\_number: 2315

old 10: sno:=&Student\_Number;

new 10: sno:=2315;

Enter value for student\_name: knreddy

old 11: name := '&Student\_Name';

new 11: name := 'knreddy';

Enter value for subject1: 86

old 12: sub1 := &subject1;

new 12: sub1 := 86;

Enter value for subject2: 89

old 13: sub2 := &subject2;

new 13: sub2 := 89;

Enter value for subject3: 90

old 14: sub3 := &subject3;

new 14: sub3 := 90;

---

Student Number :2315

Student Name :knreddy

Student Sub1 Marks :86

Student Sub2 Marks :89

Student Sub3 Marks :90

Student got Distinction

---

PL/SQL procedure successfully completed.

**✚ WRITE A PL/SQL PROGRAM FOR INSERTING ROWS INTO EMPDET  
TABLE WITH THE FOLLOWING CALCULATION**

**HRA=50% OF BASIC**

**DA=20% OF BASIC**

**PF=7% OF BASIC**

**NETPAY=BASIC+DA+HRA-PF**

```
SQL> create table empdet(  
2 empno number(3),  
3 empname varchar2(12),  
4 deptno number(3),  
5 basic number(6),  
6 hra number(5),  
7 da number(5),  
8 pf number(5),  
9 netpay number(10));
```

Table created.

```
// file name: employee
```

```
DECLARE
```

```
empno1 empdet.empno%type;  
empname1 empdet.empname%type;  
deptno1 empdet.deptno%type;  
basic1 empdet.basic%type;  
hra1 empdet.hra%type;  
da1 empdet.da%type;  
pf1 empdet.pf%type;  
netpay1 empdet.netpay%type;
```

```
BEGIN
```

```
empno1:=&empno1;  
empname1:='&empname1';  
deptno1:=&deptno1;  
basic1:=&basic1;  
hra1:=(basic1*50)/100;  
da1:=(basic1*20)/100;  
pf1:=(basic1*7)/100;  
netpay1:=basic1+hra1+da1-pf1;
```

```
insert into empdet values(empno1,'empname1',deptno1,basic1,hra1,da1,pf1,netpay1);
```

```
END;
```

```
/
```

OUTPUT:

```
SQL> @employee;
Enter value for empno1: 101
old 11: empno1:=&empno1;
new 11: empno1:=101;
Enter value for empname1: knreddy
old 12: empname1:='&empname1';
new 12: empname1:='knreddy';
Enter value for deptno1: 5
old 13: deptno1:=&deptno1;
new 13: deptno1:=5;
Enter value for basic1: 12000
old 14: basic1:=&basic1;
new 14: basic1:=12000;
```

PL/SQL procedure successfully completed.

```
SQL> select * from empdet;
```

EMPNO	EMPNAME	DEPTNO	BASIC	HRA	DA	PF	NETPAY
101	empname1	5	2000	6000	2400	840	19560

**✚ WRITE A PL/SQL PROGRAM TO CALCULATE ELECTRICITY BILLS BY THE FOLLOWING DETAILS**

<b>CATEGORY</b>	<b>UNIT</b>	<b>RATE/UNIT</b>
<b>DOMESTIC</b>	<b>&lt;=100</b>	<b>2.20</b>
	<b>&gt;100</b>	<b>3.00</b>
<b>INDUSTRIAL</b>	<b>&lt;=100</b>	<b>3.20</b>
	<b>&gt;100</b>	<b>3.60</b>
<b>COMMERCIAL</b>	<b>&lt;=100</b>	<b>2.50</b>
	<b>&gt;100</b>	<b>3.40</b>

```
// file name: elecbill
DECLARE
    catg varchar2(15);
    units number(4);
    bill number(6,2);
    invalid_input exception;
BEGIN
    catg:='&catg';
    units:=&units;
    if(catg!='domestic' AND catg!='industrial' AND catg!='commercial')
    OR(units<=0) then
        raise invalid_input;
    end if;
    if catg='domestic' then
        if units<=100 then
            bill:=units*2.20;
        else
            bill:=units*3.00;
        end if;
    else if catg='industrial' then
        if units<=100 then
            bill:=units*3.20;
        else
            bill:=units*3.60;
        end if;
    else if catg='commercial' then
        if units<=100 then
            bill:=units*2.50;
        else
            bill:=units*3.40;
        end if;
    end if;
end if;
end if;
```

```
        end if;
dbms_output.put_line('The bill amount is: '|| bill);
EXCEPTION
    when invalid_input then
dbms_output.put_line('INVALID CATEGORY OR UNITS');
END;
/
```

OUTPUT:

```
SQL> @elecbill
Enter value for catg: domestic
old 7:   catg:='&catg';
new 7:   catg:='domestic';
Enter value for units: 95
old 8:   units:='&units';
new 8:   units:=95;
The bill amount is: 209
```

PL/SQL procedure successfully completed.

```
SQL> /
Enter value for catg: college
old 7:   catg:='&catg';
new 7:   catg:='college';
Enter value for units: 102
old 8:   units:='&units';
new 8:   units:=102;
INVALID CATEGORY OR UNITS
```

PL/SQL procedure successfully completed.

```
SQL> /
Enter value for catg: industrial
old 7:   catg:='&catg';
new 7:   catg:='industrial';
Enter value for units: 250
old 8:   units:='&units';
new 8:   units:=250;
The bill amount is: 900
```

PL/SQL procedure successfully completed.



## PROCEDURES

- A stored procedure is a named collection of procedural and SQL statements
- Advantages of procedures:
  - Stored procedures substantially reduce network traffic and increase performance
  - Stored procedures help reduce code duplication by means of code isolation and code sharing, thereby minimizing the chance of errors and the cost of application development and maintenance

- To create a stored procedure, the following syntax is used:

```
CREATE OR REPLACE PROCEDURE <procedure name>
```

```
[(argument [IN/OUT] data type, --- --- )]
```

```
[IS/AS]
```

```
BEGIN
```

```
    PL/SQL or SQL statements;
```

```
    -----
```

```
END;
```

To execute the stored procedure you must use the following syntax;

```
EXEC procedure_name[(parameter list)];
```

---

**✚ WRITE A PROCEDURE TO INSERT NEW RECORD INTO THE TABLE**

```
SQL> create table cricket(  
2   cno number(4),  
3   cname varchar(10),  
4   country varchar(10));
```

Table created.

Procedure name: cricket

```
CREATE or REPLACE procedure crickinfo(cno IN number,cname IN varchar2,country IN  
varchar2) AS  
BEGIN  
insert into cricket values(cno,cname,country);  
dbms_output.put_line('one row inserted');  
END;  
/
```

OUTPUT:

```
SQL> @cricket  
Procedure created.
```

```
SQL> exec crickinfo(1,'sachin','india')  
one row inserted  
PL/SQL procedure successfully completed.
```

```
SQL> /  
Procedure created.
```

```
SQL> exec crickinfo(2,'gilcris','australia')  
one row inserted  
PL/SQL procedure successfully completed.
```

```
SQL> exec crickinfo(3,'lara','westindies')  
one row inserted  
PL/SQL procedure successfully completed.
```

```
SQL> select * from cricket;
```

CNO	CNAME	COUNTRY
1	sachin	india
2	gilcris	australia
3	lara	westindies

**FUNCTIONS**

- A stored function is basically a named group of procedural and SQL statements that returns a value (indicated by a RETURN statement in its program code).
- To create a function, you use the following syntax:

```
CREATE FUNCTION <function name>(arguments IN datatype, _ _ _ _ _ _ _ _ _ _ )
RETURN datatype
[IS]
BEGIN
    PL/SQL statements;
    _ _ _ _ _
    RETURN (value or expression);
END;
/
```

**✚ WRITE A FUNCTION TO ADD TWO NUMBERS**

```
CREATE OR REPLACE FUNCTION sumtwonum(a number, b number)
return number
is
BEGIN
return a+b;
end;
/
```

**OUTPUT:**

```
SQL> @sumtwonum
Function created.
```

```
SQL> select sumtwonum(10,20)
2 from dual;
```

```
SUMTWONUM(10,20)
```

```
-----
```

```
30
```