**Overview of Database**

A Database is a collection of related data organized in a way that data can be easily accessed, managed and updated. Any piece of information can be a data, for example name of your school. Database is actually a place where related piece of information is stored and various operations can be performed on it.

A DBMS is a software that allows creation, definition and manipulation of database. Dbms is actually a tool used to perform any kind of operation on data in database. Dbms also provides protection and security to database. It maintains data consistency in case of multiple users. Here are some examples of popular Dbms are MySQL, Oracle, Sybase, Microsoft Access and IBM DB2 etc.

**Functions of DBMS**

* Provides data Independence
* Concurrency Control
* Provides Recovery services
* Provides Utility services
* Provides a clear and logical view of the process that manipulates data.

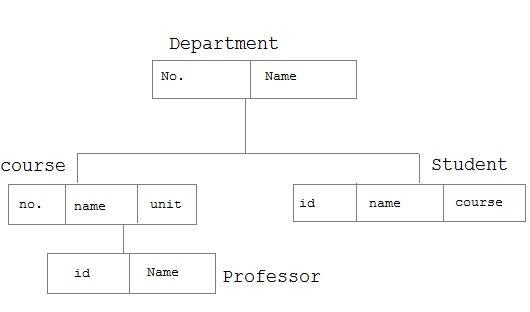
**Database Model**

A Database model defines the logical design of data. The model describes the relationships between different parts of the data. In a database design, these models have been in use.

* Hierarchical Model
* Network Model
* Relational Model
* Object Relational Model

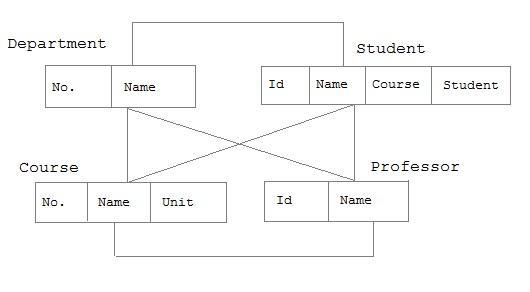
**Hierarchical Model**

In this model each entity has only one parent but can have several children. At the top of hierarchy there is only one entity which is called Root.



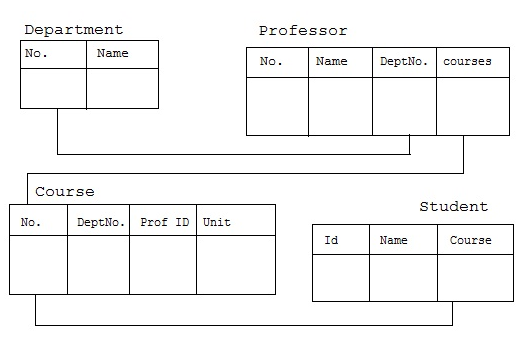
**Network Model**

In the network model, entities are organised in a graph, in which some entities can be accessed through several path.

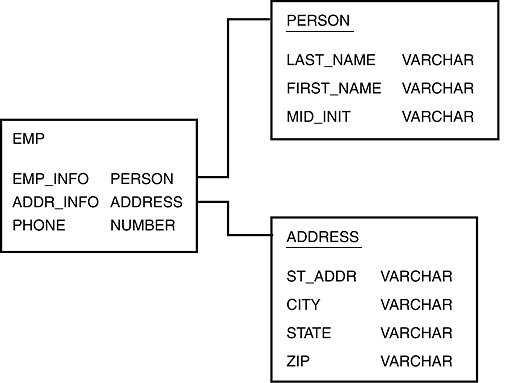
****

**Relational Model**

In this model, data is organized in two-dimensional tables called relations. The tables or relation are related to each other.



**Object Relational Model**

The basic goal for the Object-relational database is to bridge the gap between relational databases and the object-oriented modeling techniques used in programming languages such as Java, C++, Visual Basic .NET or C#. The features of Object Oriented Programming (OOP) and the features of Relational Model together is Object Relational Model.

**Codd's Rule**

E.F.T Codd was a Computer Scientist who invented Relational model for Database management. Based on relational model, Relation database was created. Codd proposed 13 rules popularly known as Codd's 12 rules to test DBMS's concept against his relational model. Codd's rule actually define what quality a DBMS requires in order to become a Relational Database Management System (RDBMS). Till now, there is hardly any commercial product that follows all the 13 Codd's rules. Even Oracle follows only eight and half (8.5) out of 13. The Codd's 12 rules are as follows.

**Rule zero**

This rule states that for a system to qualify as an RDBMS, it must be able to manage database entirely through the relational capabilities.

**Rule 1: Information Rule**

All information (including metadata) is to be represented as stored data in cells of tables. The rows and columns have to be strictly unordered.

**Rule 2: Guaranteed Access**

Each unique piece of data (atomic value) should be accessible by

Table Name + Primary key (Row) + Attribute (column).

NOTE: Ability to directly access via POINTER is a violation of this rule.

**Rule 3: Systematic Treatment of NULL**

Null has several meanings, it can mean missing data, not applicable or no value. It should be handled consistently. Primary key must not be null. Any expression on NULL must result a Null.

**Rule 4: Active Online Catalog**

Database dictionary (catalog) must have description of Database. Catalog to be governed by same rule as rest of the database. The same query language to be used on catalog as on application database.

**Rule 5: Comprehensive Data Sub-Language Rule**

One well defined language must be there to provide all manners of access to data. Example: SQL. If a file supporting table can be accessed by any manner except SQL interface, then its a violation to this rule.

**Rule 6: View Updating Rule**

All view that are theoretically updatable should be updatable by the system

**Rule 7: Relational Level Operation**

There must be Insert, Delete, and Update operations at each level of relations. Set operation like Union, Intersection and minus should also be supported.

**Rule 8: Physical Data Independence**

The physical storage of data should not matter to the system. If say, some file supporting table were renamed or moved from one disk to another, it should not affect the application.

**Rule 9: Logical Data Independence**

If there is change in the logical structure (table structures) of the database the user view of data should not change. Say, if a table is split into two tables, a new view should give result as the join of the two tables. This rule is most difficult to satisfy.

**Rule 10: Integrity Independence**

The database should be able to enforce its own integrity rather than using other programs. Key and Check constraints, trigger etc. should be stored in Data Dictionary. This also make RDBMS independent of front-end.

**Rule 11: Distribution Independence**

A database should work properly regardless of its distribution across a network. This lays foundation of distributed database.

**Rule 12: Non Subversion Rule**

If low level access is allowed to a system it should not be able to subvert or bypass integrity rule to change data. This can be achieved by some sort of looking or encryption.

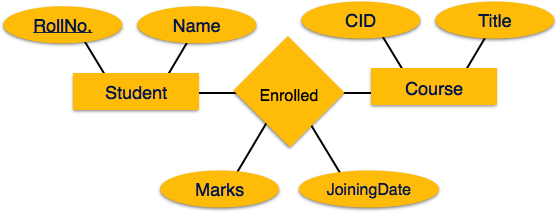
**Entity Relationship (ER) Model**

ER-Diagram is a visual representation of data that describes how data is organised and is related to each other. An ER Model, when conceptualized into diagrams, gives a good overview of entity-relationship, which is easier to understand. ER diagrams can be mapped to relational schema, that is, it is possible to create relational schema using ER diagram. We cannot import all the ER constraints into relational model, however an approximate schema can be generated.

Commonly used symbols while working with E-R diagrams are:



Entity Relationship Attribute

A sample of an E-R diagram is given below

**Normalization of Database**

Database Normalisation is a technique of organizing the data in the database. Normalization is a systematic approach of decomposing tables to eliminate data redundancy and undesirable characteristics like Insertion, Update and Deletion Anomalies (i.e., abnormalities or inconsistencies). It is a multi-step process that puts data into tabular form by removing duplicated data from the relation tables.

Normalization is used for mainly two purpose,

* Eliminating redundant (useless) data.
* Ensuring data dependencies make sense i.e. data is logically stored.

Without Normalization, it becomes difficult to handle and update the database, without facing data loss. Insertion, Updation and Deletion Anomalies are very frequent if Database is not normalized. To understand these anomalies let us take an example of Student table.

Normalization rule are divided into following normal form.

1. First Normal Form
2. Second Normal Form
3. Third Normal Form
4. BCNF
5. **First Normal Form (1NF)**

As per First Normal Form, no two Rows of data must contain repeating group of information i.e. each set of column must have a unique value, such that multiple columns cannot be used to fetch the same row. Each table should be organized into rows, and each row should have a primary key that distinguishes it as unique.

The Primary key is usually a single column, but sometimes more than one column can be combined to create a single primary key. For example consider a table which is not in First normal form.

| **Student** | **Age** | **Subject** |
| --- | --- | --- |
| Dinesh | 15 | Biology, Maths |
| Kiran | 14 | Maths |
| Vijay | 17 | Maths |

In First Normal Form, any row must not have a column in which more than one value is saved, like separated with commas. Rather than that, we must separate such data into multiple rows

**Student Table following 1NF should be as follows:**

| **Student** | **Age** | **Subject** |
| --- | --- | --- |
| Dinesh | 15 | Biology |
| Dinesh | 15 | Maths |
| Kiran | 14 | Maths |
| Vijay | 17 | Maths |

By using the First Normal Form, data redundancy increases, as there will be many columns with same data in multiple rows but each row as a whole will be unique.

1. **Second Normal Form (2NF)**

As per the Second Normal Form there must not be any partial dependency of any column on primary key. It means that for a table that has concatenated primary key, each column in the table that is not part of the primary key must depend upon the entire concatenated key for its existence. If any column depends only on one part of the concatenated key, then the table fails Second normal form.

In example of First Normal Form there are two rows for Dinesh, to include multiple subjects that he has opted for. While this is searchable, and follows First normal form, it is an inefficient use of space. Also in the above Table in First Normal Form, while the candidate key is {Student, Subject}, Age of Student only depends on Student column, which is incorrect as per Second Normal Form. To achieve second normal form, it would be helpful to split out the subjects into an independent table, and match them up using the student names as foreign keys.

**New Student Table following 2NF will be:**

| **Student** | **Age** |
| --- | --- |
| Dinesh | 15 |
| Kiran | 14 |
| Vijay | 17 |

**New Subject Table introduced for 2NF will be:**

| **Student** | **Subject** |
| --- | --- |
| Dinesh | Biology |
| Dinesh | Maths |
| Kiran | Maths |
| Vijay | Maths |

In Subject Table the candidate key will be {Student, Subject} column. Now, both the above tables qualifies for Second Normal Form and will never suffer from Update Anomalies. Although there are a few complex cases in which table in Second Normal Form suffers Update Anomalies, and to handle those scenarios Third Normal Form is there.

1. **Third Normal Form (3NF)**

**Third Normal form** applies that every non-prime attribute of table must be dependent on primary key, or we can say that, there should not be the case that a non-prime attribute is determined by another non-prime attribute. So this *transitive functional dependency* should be removed from the table and also the table must be in **Second Normal form**. For example, consider a table with following fields.

**Student\_Detail Table:**

| Student\_id | Student\_Name | DOB | Street | City | State | Pincode |
| --- | --- | --- | --- | --- | --- | --- |

In this table Student\_id is Primary key, but street, city and state depends upon Zip. The dependency between zip and other fields is called **transitive dependency**. Hence to apply **3NF**, we need to move the street, city and state to new table, with **Zip** as primary key.

**New Student\_Detail Table:**

| Student\_id | Student\_Name | Pincode |
| --- | --- | --- |

**Address Table:**

| Pincode | Street | City | State |
| --- | --- | --- | --- |

The advantage of removing transitive dependency is,

* Amount of data duplication is reduced.
* Data integrity achieved.

1. **Boyce and Codd Normal Form (BCNF)**

Boyce and Codd Normal Form is a higher version of the Third Normal form. This form deals with certain type of anomaly that is not handled by 3NF. A 3NF table which does not have multiple overlapping candidate keys is said to be in BCNF.

**SQL**

**(Structured Query Language)**

Structure Query Language (SQL) is a language used for storing and managing data in RDBMS. SQL was the first commercial language introduced for E.F.T Codd's Relational model. Today almost all RDBMS (MySQL, Oracle, Informix, Sybase, and MS Access) uses SQL as the standard database language. SQL is used to perform all type of data operations in RDBMS.

**Types of commands in SQL**

| DDL | (Data Definition Language)  Create, Alter, Drop, Truncate, Rename, Flashback, Purge |
| --- | --- |
| DML | (Data Manipulation Language)  Insert, Update, Delete (They are called as *Transactions*) |
| DQL/DRL | (Data Query/Retrieval Language)  Select |
| TCL | (Transaction Control Language)  Commit, Rollback, Savepoint |
| DCL | (Data Control Language)  Grant, Revoke |

**Logging on to Oracle SQL**

Click Start 🡪 Type “Run SQL Command Line” in the search box

At the SQL Prompt Type the following

SQL> CONNECT *Username*/*Password* (Username is “HR” and Password is “manager”)

SQL> CONNECT HR/manager **(OR)** SQL> CONN HR/manager

(**HR** is the default sample user in which the sample tables are already created, most of the further examples will be based on these sample tables)

While connecting to Oracle, it might show a few possible errors as given below:

Error 1) Oracle initialization or shutdown in progress

Solution: just wait for 1 or 2 minutes and connect again

Error 2) Invalid username or password - logon denied

Solution: recheck the spelling or capitalization of password

(Remember, Username is not case sensitive but, password is case sensitive)

Error 3) TNS: Protocol adapter error (means oracle database could not be loaded into RAM)

Solution: In the start menu, search box type "Start database" and press enter

Once the database loading is done, connect to the user

Error 4) The account is locked (this might be given for HR user)

Solution: SQL> connect system/manager

SQL> alter user hr identified by manager account unlock;

SQL> DISCONNECT

Now try to connect hr user

SYSTEM is a name of the DBA/Admin user for which the password is set at the time of installing Oracle. In our Lab, “**manager**” is the password for the user SYSTEM

SQL> DISCONNECT (or) DISC

This command is used to log off from the oracle database

SQL> SELECT \* FROM TAB;

(TAB is a pre-defined table that shows the list of tables of the current user)

**SELECT**: It is a statement is used to select the specific data from table(s).

SELECT \*/Col.Name(s)/Expression as “Alias Name”

FROM <TableName>

WHERE <condition>;

In the above syntax:

* The symbol \* means ALL columns of the given table
* Expression means any calculations
* Alias name means any text to be displayed as column/expression heading

SELECT \* FROM EMPLOYEES;

SELECT \* FROM DEPARTMENTS;

SELECT \* FROM COUNTRIES;

SELECT \* FROM LOCATIONS;

SELECT \* FROM REGIONS;

SELECT \* FROM JOBS;

SELECT \* FROM JOB\_HISTORY;

DESCRIBE: This command shows the column definitions of the given table.

SQL> DESCRIBE/DESC <TableName>

SQL> DESCRIBE EMPLOYEES

SQL> DESCRIBE DEPARTMENTS

SQL> DESCRIBE COUNTRIES

SQL> DESCRIBE LOCATIONS

SQL> DESCRIBE REGIONS

SQL> DESCRIBE JOBS

SQL> DESCRIBE JOB\_HISTORY

Selecting specific columns or expression from the table data.

SELECT EMPLOYEE\_ID, FIRST\_NAME FROM EMPLOYEES;

SELECT FIRST\_NAME, SALARY, DEPARTMENT\_ID FROM EMPLOYEES;

SELECT FIRST\_NAME, SALARY, SALARY+1000 FROM EMPLOYEES;

SELECT EMPLOYEE\_ID, UPPER(FIRST\_NAME), HIRE\_DATE FROM EMPLOYEES;

We can create alias name (alternative names) for the columns to be displayed on the output instead of the column name which are existing in the table. The alias names are given within double quotes. If the alias name is a single word then the double quotes is not compulsory.

SELECT EMPLOYEE\_ID, FIRST\_NAME FROM EMPLOYEES;

SELECT EMPLOYEE\_ID, FIRST\_NAME as “EMP NAME”, SALARY, SALARY\*10/100 as “BONUS” FROM EMPLOYEES;

When the statement contains any mistakes or we want to make corrections in the latest SQL statement then type

SQL> ED (or) EDIT

It will open the editor in which we can make the corrections

Do not give semicolon (;) and Do not remove the slash (/) in the last line

Save the command and close the editor File 🡪 Exit

SQL> **R** or **RUN** or **/** (to execute the statement)

NOTE: In order to use *ED* command you should have permissions on the Oracle folder that you will not have by default. To set these permissions:

Open “My Computer” 🡪 go to “C:” (drive in which oracle is installed) 🡪 Right click on “Oraclexe” folder 🡪 Properties 🡪 Select “Security” Tab 🡪 click “Edit” 🡪 check “Full Control” under “Allow” heading 🡪 Click “OK” 🡪 Click “OK”

**Operators used in *where* condition**

| Relational operators | > < >= <= = != or <> |
| --- | --- |
| Logical Operators | and or not |
| Special operators | Between Like In Is Null |

**Relational operators**

SELECT \* FROM EMPLOYEES WHERE EMPLOYEE\_ID=104;

SELECT \* FROM EMPLOYEES WHERE SALARY>10000;

SELECT \* FROM EMPLOYEES WHERE DEPARTMENT\_ID<100;

SELECT \* FROM DEPARTMENTS WHERE LOCATION\_ID!=1700;

Character values are compared with ASCII Codes.

(A-Z is 65-90, a-z is 97-122) and therefore character values are case sensitive

SELECT \* FROM EMPLOYEES WHERE FIRST\_NAME = ‘David’;

SELECT \* FROM EMPLOYEES WHERE FIRST\_NAME = ‘david’;

SELECT \* FROM EMPLOYEES WHERE FIRST\_NAME = ‘DAVID’;

SELECT \* FROM EMPLOYEES WHERE UPPER(FIRST\_NAME) = ‘DAVID’ ;

SELECT \* FROM EMPLOYEES WHERE LOWER(FIRST\_NAME) = ‘david’ ;

(UPPER / LOWER function converts to CAPITAL CASE or small case and compare)

SELECT \* FROM EMPLOYEES WHERE FIRST\_NAME <= ‘DAVID’;

(Although the above statement is valid according to syntax, but may not be suitable for our requirement and therefore not used)

SELECT \* FROM EMPLOYEES WHERE HIRE\_DATE = ’10-JUN-05’;

(Only *date and character values* must be given in single quotes)

(Among date values, an old date is considered to be smaller than the new date. In oracle the standard or default date format is “DD-MON-YY”)

SELECT \* FROM EMPLOYEES WHERE HIRE\_DATE >= ’10-JUN-05’;

SELECT \* FROM EMPLOYEES WHERE HIRE\_DATE != ’10-JUN-05’;

**Logical operators**

*and* shows the output when both the conditions are true

*or* shows the output when at least any one of the conditions are true

*not* shows the output when given conditions are false

SELECT \* FROM EMPLOYEES

WHERE HIRE\_DATE<'07-JUN-04' AND SALARY>10000;

SELECT \* FROM EMPLOYEES

WHERE HIRE\_DATE<'07-JUN-04' OR SALARY>10000;

SELECT \* FROM EMPLOYEES

WHERE EMPLOYEE\_ID=201 OR EMPLOYEE\_ID=205 OR EMPLOYEE\_ID=174;

SELECT \* FROM EMPLOYEES

WHERE UPPER(FIRST\_NAME)='DAVID' OR UPPER(FIRST\_NAME)='PETER';

SELECT \* FROM EMPLOYEES

WHERE DEPARTMENT\_ID=30 OR JOB\_ID='PU\_CLERK';

SELECT \* FROM EMPLOYEES

WHERE **NOT (**DEPARTMENT\_ID=30 OR JOB\_ID='PU\_CLERK'**)**;

SELECT \* FROM EMPLOYEES

WHERE **NOT** **(**HIRE\_DATE='07-JUN-04' AND SALARY>10000**)**;

**Special operators**

*between* to compare the range of values for numbers or date values

*in* to compare many values for one column

*like* to compare starting with, ending with, containing given character(s)

‘%’ Percentage symbol represents many characters

‘\_’ Underscore symbol represents one character

*is Null* to compare NULL values of a column

SELECT \* FROM EMPLOYEES

WHERE HIRE\_DATE BETWEEN '07-JUN-04' AND ’15-JUN-05’;

SELECT \* FROM EMPLOYEES

WHERE SALARY BETWEEN 10000 AND 12000;

SELECT \* FROM EMPLOYEES

WHERE FIRST\_NAME BETWEEN ‘D’ AND ‘K’; (NOT SUGGESTED AS ASCII IS CONSIDERED)

SELECT \* FROM EMPLOYEES

WHERE DEPARTMENT\_ID BETWEEN 30 AND 70;

SELECT \* FROM EMPLOYEES

WHERE EMPLOYEE\_ID IN (106, 145, 178, 200, 205);

SELECT \* FROM EMPLOYEES

WHERE UPPER(FIRST\_NAME) IN(‘DAVID’,’PETER’,’SUSAN’);

SELECT \* FROM EMPLOYEES

WHERE HIRE\_DATE IN(‘17-AUG-07’, ’07-JUN-04’, ’21-JUN-05’);

SELECT EMPLOYEE\_ID, FIRST\_NAME FROM EMPLOYEES WHERE UPPER(FIRST\_NAME) LIKE ‘D%’;

(Names starting with ‘D’)

SELECT EMPLOYEE\_ID, FIRST\_NAME FROM EMPLOYEES WHERE UPPER(FIRST\_NAME) LIKE ‘%D’;

(Names ending with ‘D’)

SELECT EMPLOYEE\_ID, FIRST\_NAME FROM EMPLOYEES WHERE UPPER(FIRST\_NAME) LIKE ‘%D%’;

(Names containing ‘D’)

SELECT EMPLOYEE\_ID, FIRST\_NAME FROM EMPLOYEES WHERE UPPER(FIRST\_NAME) LIKE ‘\_\_\_\_’;

(Names containing 4 characters)

SELECT EMPLOYEE\_ID, FIRST\_NAME FROM EMPLOYEES WHERE UPPER(FIRST\_NAME) LIKE ‘\_D\_\_’;

(Names containing 4 characters and 2nd character is ‘D’)

SELECT EMPLOYEE\_ID, FIRST\_NAME FROM EMPLOYEES WHERE UPPER(FIRST\_NAME) LIKE ‘\_D%’;

(Names containing 2nd character is ‘D’ and further any no of chars.)

SELECT EMPLOYEE\_ID, FIRST\_NAME FROM EMPLOYEES WHERE UPPER(FIRST\_NAME) LIKE ‘%D\_’;

(Names containing any no of chars. and 2nd last character is ‘D’)

NULL is a special value which is NOT space and NOT zero. NULL is empty value. To compare NULL values we use *is Null*.

SELECT EMPLOYEE\_ID, FIRST\_NAME, DEPARTMENT\_ID

FROM EMPLOYEES WHERE DEPARTMENT\_ID IS NULL;

SELECT EMPLOYEE\_ID, FIRST\_NAME, DEPARTMENT\_ID

FROM EMPLOYEES WHERE DEPARTMENT\_ID IS NOT NULL;

NOT can be prefixed with all special operators

NOT between, NOT in, NOT like, is NOT null

**Commonly used Data Types in SQL**

1. Number To store numeric values. Maximum 38 digits

Ex: Rno number(5),

avrg number(6,2) (6 is precision(total width) 2 is scale)

For example, a specification of (10,-2) means to round to hundreds

1. Char To store character values of fixed width. Maximum 2000 chars.

Ex: St\_Name char(10)

(Within this width if 10 is not used then that unused space is wasted)

1. Varchar2 To store character values of variable width. Max 4000 chars.

Ex: St\_Name Varchar2(10)

(Within this width if 10 is not used then that unused space is NOT wasted and adjusted for the next values)

1. Date Store date & time values. Takes 9 bytes (7 for date 2 for time)

Ex: Dob Date, manfDate Date, arrTime Date

(the default date format is dd-mon-yy, when we don’t give the time value, it take 12:00:00 (midnight)

1. Long To store large continuous text for each record. Max 4000 chars

Ex: remarks long

1. Raw To store images (photographs). Max 2000 Bytes

Ex: photo raw

1. Long Raw To store images of larger size Max 2GB
2. BLOB **B**inary **L**arge **Ob**ject To store audio / video files Max 4 GB
3. CLOB **C**haracter **L**arge **Ob**ject To store large continuous text Max 4GB

**CREATE TABLE**: To create a new table with the given column specification.

*Method 1*: To create a new table with the given columns and the data types.

CREATE TABLE <TABLENAME> (

<COL1> <DATATYPE>(WIDTH),

<COL2> <DATATYPE>(WIDTH),

. . . . . .

. . . . . . . . . . .);

Eg: CREATE TABLE FRIENDS(

SNO NUMBER(4),

NAME VARCHAR2(10),

DOB DATE,

PHONE NUMBER(10),

AVRG NUMBER(6,2));

SELECT \* FROM TAB;

DESC FRIENDS;

Try creating some more tables with columns as given below:

Cricketers: Sno, Name, Dob, Country, Matches, Runs, AvrgPerformance

Students: Rno, Name, Course, Doj, Fees, Time, Faculty

*Method 2*: To create a new table with the values of another existing table. In this method a new table is created with the output of the select statement. The columns of the new table will be created with the same data types and width as in the existing table.

CREATE TABLE <TABLENAME> [(COL1,<COL2>. . .)] AS <SELECT STATEMENT>;

CREATE TABLE PROJECT1 AS SELECT \* FROM EMPLOYEES WHERE EMPLOYEE\_ID IN (108, 145, 193, 195, 200);

SELECT \* FROM PROJECT1;

DESCRIBE PROJECT1;

CREATE TABLE PROJECT2 AS SELECT EMPLOYEE\_ID, FIRST\_NAME, HIRE\_DATE, SALARY FROM EMPLOYEES WHERE EMPLOYEE\_ID IN (108, 145, 193, 195, 200);

SELECT \* FROM PROJECT2;

DESCRIBE PROJECT2;

CREATE TABLE PROJECT3 (EMPNO, EMPNAME, JOINDATE, PAY) AS SELECT EMPLOYEE\_ID, FIRST\_NAME, HIRE\_DATE, SALARY FROM EMPLOYEES WHERE EMPLOYEE\_ID IN (108, 145, 193, 195, 200);

SELECT \* FROM PROJECT3;

DESCRIBE PROJECT3;

*Method 3*: To create a new table with the same STRUCTURE of another existing table having no rows. To implement this the same above syntax can be followed but give a filter (where condition) which extracts NO rows, ie., use a condition which is NEVER true.

CREATE TABLE PROJECT4 (EMPNO, EMPNAME, JOINDATE, PAY) AS SELECT EMPLOYEE\_ID, FIRST\_NAME, HIRE\_DATE, SALARY FROM EMPLOYEES WHERE 1=2;

SELECT \* FROM PROJECT4; (No rows are shown)

DESCRIBE PROJECT4;

Here a table is created first and then rows that satisfy the condition will be copied into the table. In the above example the *condition* (1=2) is a NEVER true therefore table is created with NO rows.

**INSERT**: This command is used to insert records/rows into the table. The new rows can be added only at the end. (records cannot be inserted in between the existing)

*Method 1*: To insert values into the table with user given values

INSERT INTO <TABLENAME>[(COL1,COL2,. . . . .)] VALUES(VAL1, VAL2, . . . .);

INSERT INTO FRIENDS VALUES(1,’PETER’,’10-JUN-89’, 9776656363);

INSERT INTO FRIENDS VALUES(2,’JOHN’,NULL,NULL);

In the above method we have to insert values for all columns of the table.

INSERT INTO FRIENDS(NAME,DOB) VALUES

(’PETER’,’10-JUN-89’);

In this method we have to give values for only those columns whose column names are specified.

*Method 2*: To insert values into a table from the values of another table.

INSERT INTO <TABLENAME>[(COL1,COL2,. . . . .)]

<SELECT STATEMENT>;

EG: INSERT INTO FRIENDS (SNO,NAME) SELECT RNO,SNAME FROM STUDENTS WHERE RNO IN (2,4,5,8);

Using ‘&’ for inserting values into tables

‘&’ symbol asks for value from the user

EG: INSERT INTO FRIENDS VALUES(&SNO, ‘&NAME’, ‘&DOB’, &PHONE);

**UPDATE**: To make changes with the data of the table.

UPDATE <TABLENAME> SET <COL>=VALUE/EXPRESSION

[WHERE <CONDITION>];

‘Where’ condition is optional. If the where condition is not given then the update command is for all rows of the table

EG: UPDATE FRIENDS SET NAME = UPPER(NAME);

UPDATE FRIENDS SET PHONE=9876543210 WHERE SNO=1;

UPDATE FRIENDS SET DOB=’18-OCT-95’ WHERE SNO=103;

UPDATE FRIENDS SET SNO=300 WHERE NAME=’DAVID’;

UPDATE FRIENDS SET PHONE = NULL; (all phone nos. are replaced with empty)

UPDATE FRIENDS SET PHONE = NULL WHERE DOB IS NULL;

( = NULL is used to assign and IS NULL is used to compare)

UPDATE EMPLOYEES SET SALARY=SALARY+1000 WHERE DEPARTMENT\_ID=30; (Increment salary by 1000 for 30 department)

UPDATE EMPLOYEES SET SALARY=SALARY+(SALARY\*12.5/100) WHERE DEPARTMENT\_ID=40; (Increment salary by 12.5% for 40 dept.)

**DELETE**: To remove the rows from the table (**temporarily**).

DELETE \* FROM <TABLENAME> [WHERE <CONDITION>];

If the ‘where condition’ is not given it delete all rows of the table.

EG: DELETE \* FROM EMPLOYEES WHERE EMPLOYEE\_ID=109;

DELETE \* FROM EMPLOYEES WHERE HIRE\_DATE<=’01-JAN-91’;

DELETE \* FROM EMPLOYEES; (Removes all rows of the employees table)

DELETE \* FROM EMPLOYEES WHERE DEPARTMENT\_ID IS NULL;

**ALTER**: This command is used to make changes in the table structure. We can change the column definitions by using this command.

Using ALTER command we can:

1. Rename a column
2. Drop a column
3. Add a column
4. Modify the data type or width of a column
5. Rename a column

ALTER TABLE <TABLENAME> RENAME COLUMN <ColName> TO <NewColName>;

Eg: ALTER TABLE FRIENDS RENAME COLUMN PHONE TO MOBILE;

DESCRIBE FRIENDS;

1. Drop a column

ALTER TABLE <TABLENAME> DROP COLUMN <ColName>;

Eg: ALTER TABLE FRIENDS DROP COLUMN EMAIL;

1. Add a column

New columns can be added only at the end. Cannot be inserted in between

ALTER TABLE <TABLENAME> ADD(

<ColName> <DATATYPE>(WIDTH),

<ColName> <DATATYPE>(WIDTH),

. . . . . . . . . . . );

Eg: ALTER TABLE FRIENDS ADD(EMAIL VARCHAR2(30), CITY VARCHAR2(20));

1. Modify the data type or width of a column

ALTER TABLE <TABLENAME> MODIFY(

<ColName> <DATATYPE>(WIDTH),

<ColName> <DATATYPE>(WIDTH),

. . . . . . . . . . . );

NOTE: The column to be modified must be empty to change the data type or to decrease the precision or scale.

**DROP**: This command is used to remove a table permanently. When a table is dropped, all its data will be lost.

DROP TABLE <TABLENAME>;

EG:

CREATE TABLE FRIENDS2 AS SELECT \* FROM FRIENDS;

DROP TABLE FRIENDS2;

**RENAME**: This command is used to change the table name.

RENAME <TABLENAME> TO <New TABLENAME>;

Eg: RENAME FRIENDS TO ALLFRIENDS;

SELECT \* FROM FRIENDS; (This command does not work)

SELECT \* FROM ALLFRIENDS; (This command works)

**TRUNCATE**: This command is used to delete **ALL the rows** of the table permanently. We cannot get back the deleted rows. The table is not deleted.

TRUNCATE TABLE <TABLENAME>;

Eg: TRUNCATE TABLE FRIENDS2;

SELECT \* FROM FRIENDS2; (No records will be shown)

**TCL COMMANDS (Transaction Control Language)**

COMMIT ROLLBACK SAVEPOINT

Transaction means any DML operation (Insert, Update, Delete)

**COMMIT:** Makes all the transactions of the client system permanent to the server. It will refresh the client logical schema

COMMIT;

**ROLLBACK:** This command will undo all the transactions (DML operations) done unto the latest commit operation. This command leaves all transactions of the client system and gets a fresh copy of the current user’s logical schema to the client machine from the database server.

NOTE: Rollback cannot be possible after commit.

COMMIT;

INSERT INTO . . . . .

DELETE FROM . . . . .

UPDATE . . . . . . . .

ROLLBACK;

This will undo all the above 3 transactions (Insert, Delete, Update) and will get back to the same state as at the commit state.

Rollback is not specific to any table or any statement.

**SAVEPOINT:** This command will create an identification point with a savepoint name, so that all the transactions up to a given *savepoint* can be rolled back. We can create multiple savepoints, however the transactions between two savepoints cannot be rolled back. Rollback is always from the current position to a given savepoint. All transactions (DML operations) up to that savepoint will be rolled back including the other savepoints (if any) in the sequence. For example:

COMMIT;

INSERT INTO . . . . .

DELETE FROM . . . . . 

SAVEPOINT P1;

UPDATE . . . . . . . .

DELETE . . . . . . . .

INSERT INTO . . . . . .

SAVEPOINT P2;

INSERT INTO . . . . .

UPDATE . . . . .

(at this point we have 3 options as given below)

ROLLBACK TO P2;/ROLLBACK TO P1;/ROLLBACK;

NOTE:

Any of the following command will execute a commit operation automatically without giving a commit command.

CREATE, ALTER, DROP, TRUNCATE, RENAME, FLASHBACK, PURGE, CONNECT, DISCONNECT, EXIT

We can also set commit operation to be executed after every DML operation.

SET AUTOCOMMIT ON/OFF;

**DCL COMMANDS (Data Control Language)**

GRANT REVOKE

**GRANT:** This command is used to grant/give permissions (privileges) on a table to one or many users.

GRANT <privileges> ON <TABLENAME> TO <USERNAMES>;

**REVOKE:** This command is used to revoke/take back permissions (privileges) on a table from one or many users.

REVOKE <privileges> ON <TABLENAME> FROM <USERNAMES>;

Privileges that can be granted or revoked are:

Select, Insert, Delete, Update, Alter, Index, References, ALL

The user who gives permissions is called GRANTOR and the user who receives is called GRANTEE.

If user ‘A’ grants permissions to user ‘B’ then ‘A’ is the Grantor and ‘B’ is Grantee.

The grantee (B) has to refer the grantor’s (A) table as <GrantorName.TableName>

1. GRANT SELECT, INSERT, UPDATE ON FRIENDS TO B;
2. SELECT \* FROM A.FRIENDS;

B> INSERT INTO A.FRIENDS VALUES(. . . . );

B> UPDATE A.FRIENDS SET . . . . ;

B> DELETE FROM A.FRIENDS; (This command does not work because delete permission is not given. It show an error as “Insufficient privileges”)

1. REVOKE ALL ON FRIENDS FROM B;

B> UPDATE A.FRIENDS SET . . . . ;

1. DELETE FROM A.FRIENDS;

(This above two commands do not work because permissions are taken back)

Information about the permissions given/received is present in the following tables.

SELECT \* FROM USER\_TAB\_PRIVS\_RECD; (Permissions received)

SELECT \* FROM USER\_TAB\_PRIVS\_MADE; (Permissions given)

**Managing users accounts (DBA Activities)**

New users can be created by any user having DBA privileges. SYSTEM is a default DBA user already created in the oracle environment. We have to first login to the DBA user account, and then start working with commands to manage the users.

CONNECT SYSTEM/MANAGER (MANAGER is the assumed password)

SELECT \* FROM ALL\_USERS; (This table shows the existing users list)

**Creating New User**

CREATE USER <username> IDENTIFIED BY <password>;

CREATE USER SACHIN IDENTIFIED BY SACHIN123;

**Assigning basic level permissions**

GRANT CONNECT, RESOURCE TO <username>;

GRANT CONNECT, RESOURCE TO SACHIN;

GRANT DBA TO SACHIN; (This command makes user SCAHIN as DBA)

A DBA user can directly access any user’s data. He does NOT need any privileges to be given on any data of any user.

**Changing password**

Password of a user can be changed by the *user himself* or by a *DBA user*:

ALTER USER <username> IDENTIFIED BY <password>;

ALTER USER SACHIN IDENTIFIED BY SACH123$;

**Removing a user**

User can be dropped if the user account is not needed by a *DBA user* as:

DROP USER <username> [CASCADE];

DROP USER SACHIN CASCADE;

*CASCADE* is used to drop all objects in the user's schema before dropping the user. You must specify *CASCADE* clause to drop a user whose schema contains any objects.

**Functions in SQL**

A Function is a predefined set of statements which takes values as arguments and provides a result.



Row Functions : They produce one output for each row of the table.

Column Functions : They produce one output for entire column of the table.

**Row Functions are of 5 types:**

* Number/Mathematical Functions
* Character/String Functions
* Conversion Functions
* Date & Time Functions
* Miscellaneous Functions

**Number/Mathematical Functions**

Abs() : Returns the absolute value (+ve value). It converts the given number to positive value if the given value is positive or negative.

SELECT ABS(-5) FROM DUAL;

Output: 5

Sqrt() : Calculate the square root of a given number.

SELECT SQRT(64) FROM DUAL;

Output: 8

Power() : Returns the power of number raised to another number.

SELECT POWER(5,3) FROM DUAL;

Output: 125 (same as 5\*5\*5 🡪 53)

Mod() : Returns the remainder of division.

SELECT MOD(9,2) FROM DUAL;

Output: 1 (9 ÷ 2 quotient is 4 and remainder is 1)

SELECT MOD(2,9) FROM DUAL;

Output: 2 (in mod(x,y) if x < y then x is the remainder)

Ceil() : Returns the ceiling value (next coming integer) of the given number.

SELECT CEIL(5.3) FROM DUAL;

Output: 6 (6 is the ceiling for all values between 5 and 6)

SELECT CEIL(5.8) FROM DUAL;

Output: 6 (6 is the ceiling for all values between 5 and 6)

Floor() : Returns the floor value (integer value) of the given number.

SELECT FLOOR(5.3) FROM DUAL;

Output: 5 (5 is the floor value for all values between 5 and 6)

SELECT FLOOR(5.8) FROM DUAL;

Output: 5 (5 is the floor value for all values between 5 and 6)

Sign() : Returns the sign of a number. This functions return 1 if the number is positive or returns -1 if the number is negative or returns 0 if the number is 0.

SELECT SIGN(-8) FROM DUAL;

Output: -1

SELECT SIGN(8) FROM DUAL;

Output: 1

SELECT SIGN(0) FROM DUAL;

Output: 0

Sin()/Cos()/Tan() : Returns the trigonometric value of the given degrees.

SELECT SIN(45) FROM DUAL;

Output: .8509 (same as 1/√2, value is calculated to 38 decimals accuracy)

Round() : To ROUND off a number to given number decimals.

SELECT ROUND(6485.7362,2) FROM DUAL;

Output: 6485.74 (as 3rd decimal is > 5 the 2nd decimal is increased)

SELECT ROUND(6485.7362,3) FROM DUAL;

Output: 6485.736 (as 4th digit is < 5 it remains as .736)

SELECT ROUND(6485.7362,0) FROM DUAL;

Output: 6486 (as 1st decimal is > 5 it increases the number)

SELECT ROUND(6485.7362,-1) FROM DUAL;

Output: 6490 (-1 indicates nearest 10s)

SELECT ROUND(6485.7362,-2) FROM DUAL;

Output: 6500 (-2 indicates nearest 100s)

SELECT ROUND(6485.7362,-3) FROM DUAL;

Output: 6000 (-3 indicates nearest 1000s)

Trunc() : To CUT off a number to given number decimals.

SELECT TRUNC(6485.7362,2) FROM DUAL;

Output: 6485.73 (any value after 2 decimals is cut off)

SELECT TRUNC(6485.7362,3) FROM DUAL;

Output: 6485.736 (any value after 3 decimals is cut off)

SELECT TRUNC(6485.7362,0) FROM DUAL;

Output: 6485 (any value after 0 decimals is cut off)

SELECT TRUNC(6485.7362,-1) FROM DUAL;

Output: 6480 (-1 indicates Lower 10)

SELECT TRUNC(6485.7362,-2) FROM DUAL;

Output: 6400 (-2 indicates Lower 100)

SELECT TRUNC(6485.7362,-3) FROM DUAL;

Output: 6000 (-3 indicates Lower 1000)

Greatest() : Returns the highest value of the given values.

SELECT GREATEST(55,73,36,84,47) FROM DUAL;

Output: 84 (the highest value in the list)

Least() : Returns the smallest value of the given values.

SELECT LEAST(55,73,36,84,47) FROM DUAL;

Output: 36 (the smallest value in the list)

**Character/String Functions**

Length() : Returns the length of a string including the spaces contained.

SELECT LENGTH(‘Hello HOW are YOU’) FROM DUAL;

Output: 17

Upper() : Returns the text converted into upper case (CAPITAL CASE).

SELECT UPPER(‘Hello HOW are YOU’) FROM DUAL;

Output: HELLO HOW ARE YOU

Lower() : Returns the text converted into lower case (small case letters).

SELECT LOWER(‘Hello HOW are YOU’) FROM DUAL;

Output: hello how are you

Initcap() : Capitalizes the first character or each word (Title Case Letters).

SELECT INTICAP(‘Hello HOW are YOU’) FROM DUAL;

Output: Hello How Are You

Concat() : Concatenates (joins) two strings.

SELECT CONCAT(‘HELLO’,‘ HOW ARE YOU’) FROM DUAL;

Output: HELLO HOW ARE YOU

Ascii() : Returns the ASCII code of the given ASCII character. These are 256.

ASCII CODES A to Z are 65 to 90 a to z are 97 to 122

SELECT ASCII(‘H’) FROM DUAL;

Output: 72

Chr() : Returns the ASCII character of the given ASCII code.

SELECT CHR(72) FROM DUAL;

Output: H

Substr() : Returns the specific no. characters from given position of a string.

SELECT SUBSTR(‘COMPUTER’,4,3) FROM DUAL;

Output: PUT (4 is position, 3 is no. of characters)

Ltrim() : Removes blanks at the left-end of a string.

SELECT LTRIM(‘ HELLO ’) FROM DUAL;

Output: ‘HELLO ’

Rtrim() : Removes blanks at the right-end of a string.

SELECT RTRIM(‘ HELLO ’) FROM DUAL;

Output: ‘ HELLO’

Trim() : Removes blanks at the both ends of a string.

SELECT TRIM(‘ HELLO ’) FROM DUAL;

Output: ‘HELLO’

Lpad() : To fill the unused space of given width with given character on left.

SELECT LPAD(‘Hello’,9,’\*’) FROM DUAL;

Output: ‘\*\*\*\*HELLO’

Rpad() : To fill the unused space of given width with given character on right.

SELECT RPAD(‘Hello’,9,’\*’) FROM DUAL;

Output: ‘HELLO\*\*\*\*’

Replace() : Replaces every occurrence of the second argument in string by the third argument (or by nothing if third argument is not given).

SELECT REPLACE(‘HELLO’,’LL’,’R’) FROM DUAL;

Output: ‘HERO’

SELECT REPLACE(‘HELLO’,’LL’) FROM DUAL;

Output: ‘HEO’

Reverse() : To reverse a given string.

SELECT REVERSE(‘HELLO HOW ARE YOU’) FROM DUAL;;

Output: UOY ERA WOH OLLEH

Translate() : Translates every character of the second argument in string by the third argument.

SELECT TRANSLATE(‘entertainment’,’et’,’XY’) FROM DUAL;

Output: ‘XnYXrYainmXnY’ (X in place of e and Y in place of t)

Soundex() : To compare the phonetic sound of given strings.

SELECT FIRST\_NAME FROM EMPLOYEES WHERE FIRST\_NAME= ‘SHELEY’; (may not show the output as name does not match exactly)

SELECT FIRST\_NAME FROM EMPLOYEES WHERE SOUNDEX(FIRST\_NAME) = SOUNDEX(‘SHELEY’);

(This shows the names that sound similar to the value on right side of =)

**Conversion Functions**

These functions are to convert the values of one data type to another data type.

To\_date() : To convert character value to a date format.

To\_number() : To convert character value to a number format.

To\_char() : To convert number value or date value to character format.

Working with Date values

sysdate : To show the current date of the system in the format “dd-mon-yy”.

SELECT SYSDATE FROM DUAL;

Output: 10-SEP-14

Date Formats

D : Day no. of the week. (1 to 7 which is Sun to Sat)

DD : Day no. of the Month. (1 to 31)

DDD : Day no. of the Year. (1 to 366)

W : Week number in the month. (1-5)

WW : Week number in the year. (1-53)

DY : Day name in short. (Sun, Mon……Sat)

DAY : Day name in full. (Sunday, Monday……Saturday)

MM : Month no. of the year. (1 to 12)

MON : Month Name in short. (Jan, Feb…...Dec)

MONTH : Month Name in full. (January, February…...December)

YY : Year no. in short. (99, 00, 05, 14)

YYYY : Year no. in full. (1999, 2000, 2005, 2014)

YEAR : Year no. spelt out. (Nineteen Ninety Nine, Twenty Fourteen)

HH : Hours in 12 Hours format. (1,2,3,4…12, 1,2,3,…)

HH24 : Hours in 24 Hours format. (1,2,3,4…12,13,14,15…23,00, 1,2…)

MI : Minutes. (0 to 59)

SS : Seconds. (0 to 59)

SELECT TO\_CHAR(<DateValue>,<DateFormat>)FROM<TableName>

SELECT TO\_CHAR(SYSDATE,’DAY, DD-MONTH-YYYY’) FROM DUAL;

Output: WEDNESDAY, 10-SEPTEMBER-2014

SELECT TO\_CHAR(SYSDATE,’HH:MI:SS’) FROM DUAL;

Output: 04:59:40

SELECT TO\_CHAR(SYSDATE,’HH24:MI:SS’) FROM DUAL;

Output: 16:59:40

Data values are displayed in the default format as ‘DD-MON-YY’. Now that we know various date formats, we can change the standard date display format using the following command.

ALTER SESSION SET NLS\_DATE\_FORMAT=’<REQUIRED FORMAT>’;

Eg: ALTER SESSION SET NLS\_DATE\_FORMAT=’Day, DD-MON-YYYY’;

SELECT SYSDATE FROM DUAL;

SELECT EMPLOYEE\_ID, FIRST\_NAME, HIRE\_DATE FROM EMPLOYEES;

We can now observe that the date values are displayed in the format that we have set. The setting that is given is just for the current session of SQL only. This setting will again set back to default (‘DD-MON-YY’)when the next session is started.

Calculations with Date values

Date value + No. of days = Date Value (Today’s date + 1 gives tomorrow’s date)

Date value - No. of days = Date Value (Today’s date - 1 gives yesterday’s date)

Date value - Date Value = No. of days (Today’s date - Yesterday’s date gives 1 )

SELECT SYSDATE + 5 FROM DUAL;

Output: 15-Sep-14

SELECT SYSDATE - 5 FROM DUAL;

Output: 05-Sep-14

SELECT TO\_DATE(’10-JUN-14’) – TO\_DATE(’15-APR-14’) FROM DUAL;

Output: 56

SELECT TO\_CHAR(SYSDATE + 5,’DAY’) FROM DUAL;

Output: Monday

**Date & Time Functions**

Add\_months() : To add specified no. of months to a given date value.

SELECT ADD\_MONTHS(’12-JUN-14’,10) FROM DUAL;

Output: 12-APR-15 (10 months added to the given date value)

SELECT ADD\_MONTHS(’29-JAN-14’,1) FROM DUAL;

Output: 28-FEB-14

(for any value such as 28, 29, 30, 31 Jan, adding 1 month will give 28-FEB-14 only)

Months\_between() : To find the difference between two date values in months.

SELECT MONTHS\_BETWEEN(’12-JUN-14’,’20-APR-13’)FROM DUAL

Output: 13.7419355 (it means 13 months & 74% of month 22 days approx.)

Last\_day() : Returns Last day’s date of the month in the given date value.

SELECT LAST\_DAY(’07-JUN-14’) FROM DUAL;

Output: 30-JUN-14 (Last date of the month given in the date value)

SELECT LAST\_DAY(’10-FEB-12’) FROM DUAL;

Output: 29-FEB-12 (2012 is a ***Leap*** year therefore 29 is returned)

Next\_day() : Returns next coming specified day’s date after a given date value.

SELECT NEXT\_DAY(’05-JUN-14’,’TUE’) FROM DUAL;

Output: 10-JUN-14 (Date of next TUESDAY after 05-JUN-14)

Round() : Rounds off the given date value to:

* Nearest first day of the week (Sunday is the first day of the week)
* Nearest first day of the month
* Nearest first day of the year

SELECT ROUND(TO\_DATE(’19-JUN-14’),’DAY’) FROM DUAL;

Output: 22-JUN-14 (nearest Sunday to 19-JUN-14 is 22-JUN-14)

SELECT ROUND(TO\_DATE(’19-JUN-14’),’MONTH’) FROM DUAL;

Output: 01-JUL-14 (nearest 1st day of month to 19-JUN-14 is 01-JUL-14)

SELECT ROUND(TO\_DATE(’19-JUN-14’),’YEAR’) FROM DUAL;

Output: 01-JAN-14 (nearest 1st day of the year to 19-JUN-14 is 01-JAN-14)

Trunc() : Returns the CUT off of the given date value to:

* Current week’s first day (Sunday is the first day of the week)
* Current Month’s first day
* Current Year’s first day

SELECT TRUNC(TO\_DATE(’19-JUN-14’),’DAY’) FROM DUAL;

Output: 15-JUN-14 (Current week is from 15 to 21)

SELECT TRUNC(TO\_DATE(’19-JUN-14’),’MONTH’) FROM DUAL;

Output: 01-JUN-14 (1st day of month to 19-JUN-14 is 01-JUN-14)

SELECT TRUNC(TO\_DATE(’19-JUN-14’),’YEAR’) FROM DUAL;

Output: 01-JAN-14 (1st day of the year to 19-JUN-14 is 01-JAN-14)

Greatest() : Returns the highest date value (latest date) from the given list of date values. As this is not just used with date values but also with nos., therefore, the date values have to be converted to the date format using *to\_date()* function.

SELECT GREATEST(

TO\_DATE(’05-JUN-14’),

TO\_DATE(’31-DEC-14’),

TO\_DATE(’25-APR-14’),

TO\_DATE(’16-DEC-14’)) FROM DUAL;

Output: 31-DEC-14 (Latest date value among the list)

Least() : Returns the smallest date value (oldest date) from the given list of date values. As this is not just used with date values but also with nos., therefore, the date values have to be converted to the date format using *to\_date()* function.

SELECT LEAST(

TO\_DATE(’05-JUN-14’),

TO\_DATE(’31-DEC-14’),

TO\_DATE(’25-APR-14’),

TO\_DATE(’16-DEC-14’)) FROM DUAL;

Output: 25-APR-14 (Oldest date value among the list)

**Miscellaneous Functions**

DECODE() : To compare value for a column and return a user given value corresponding to that value. For example if we have to update the salaries of the employees based on their *JOB\_ID* then we may have to give an update for each conditions separate as follows.

UPDATE EMPLOYEES SET SALARY=SALARY+1000 WHERE JOB\_ID=’IT\_PROG’; UPDATE EMPLOYEES SET SALARY=SALARY+800 WHERE JOB\_ID=’MK\_MAN’; UPDATE EMPLOYEES SET SALARY=SALARY+500 WHERE JOB\_ID=’PU\_CLERK’; UPDATE EMPLOYEES SET SALARY=SALARY+300 WHERE JOB\_ID=’SA\_REP’; UPDATE EMPLOYEES SET SALARY=SALARY+100 WHERE JOB\_ID NOT IN (’IT\_PROG’,’MK\_MAN’,’PU\_CLERK’,’SA\_REP);

This can be reduced into a single query by using the DECODE function.

UPDATE EMPLOYEES SET SALARY=SALARY+

DECODE(JOB\_ID,

’IT\_PROG’,1000,

’MK\_MAN’,800,

’PU\_CLERK’,500,

’SA\_REP’,300,100);

The last value (100) is substituted in all those rows where JOB\_ID does not match with any of the values in the given list.

NVL() : To substitute a user given value in place of NULL value of a column.

A NULL with any arithmetic operation will result in NULL.

SELECT FIRST\_NAME, SALARY\*NVL(COMMISSION\_PCT,0) “COMMISSION\_AMT” FROM EMPLOYEES;

Pseudo Columns

Pseudo columns are similar to the columns of a table, but are not actually stored in the table. You can select from pseudo columns, but you cannot insert, update, or delete their values. A pseudo column is also similar to a function without arguments. However, functions without arguments typically return the same value for every row in the result set, whereas pseudo columns typically return a different value for each row.

Following are some of the pseudo columns in recognized with SQL and PL/SQL.

SYSDATE, CURRENT\_DATE, SYSTIMESTAMP, ROWID, ROWNUM, UID, USER, CURRVAL, NEXTVAL, etc.

SYSDATE : It returns the date of the database server system.

SELECT SYSDATE FROM DUAL;

Output: 15-SEP-14

(If database server is in another country then date on that server system is returned)

CURRENT\_DATE : It returns the date of the client system.

SELECT CURRENT\_DATE FROM DUAL;

Output: 15-SEP-14 (Current date of the local client system)

SYSTIMESTAMP: It returns the timestamp on the oracle database server machine which includes date and time with the time zone.

SELECT SYSTIMESTAMP FROM DUAL;

Output: 15-SEP-14 03.01.31.228000 PM +05:30

ROWID : It is unique value used to identify a row of a table. This value is unique among all rows, all tables and all users of the oracle database. Generally it is used to identify the rows uniquely that have all duplicated data.

ROWNUM : It is a sequence of numbers generated for the current output of the SQL statement.

SELECT ROWID, ROWNUM, EMPLOYEE\_ID, FIRST\_NAME FROM EMPLOYEES WHERE DEPARTMENT\_ID=30;

Output:

ROWID ROWNUM EMPLOYEE\_ID FIRST\_NAME

------------------ ---------- ----------- -----------

AAAC9EAAEAAAABXAAO 1 114 Den

AAAC9EAAEAAAABXAAP 2 115 Alexander

AAAC9EAAEAAAABXAAQ 3 116 Shelli

AAAC9EAAEAAAABXAAR 4 117 Sigal

AAAC9EAAEAAAABXAAS 5 118 Guy

AAAC9EAAEAAAABXAAT 6 119 Karen

6 rows selected.

UID : It returns the current user id number; this is a unique number among all users of the database.

USER : It returns the current user name.

SELECT UID, USER FROM DUAL;

Output: 33 HR (SELECT \* FROM ALL\_USERS; also shows this info.)

CURRVAL/NEXTVAL: It returns the current value of the sequence being referred. (Discussed later with database objects - *Sequences*)

**Group/Column Functions:**

These functions are used only on columns of a table, not on individual values.

**Sum()** : Returns the total of the given numeric column of a table.

SELECT SUM(SALARY) FROM EMPLOYEES;

Output: 798900 (all rows salary value added together)

SELECT SUM(SALARY) FROM EMPLOYEES WHERE JOB\_ID=’IT\_PROG’;

Output: 38800 (all rows salary value of IT\_PROGs added together)

**Avg()** : Returns the Average value of the given numeric column of a table.

SELECT AVG(SALARY) FROM EMPLOYEES;

Output: 7536.79245

SELECT SUM(SALARY) FROM EMPLOYEES WHERE JOB\_ID=’IT\_PROG’;

Output: 7760

**Min()** : Returns the Minimum value of the given numeric/character/date column of a table. (NOTE: Character column values are based on ASCII Codes)

SELECT MIN(SALARY) FROM EMPLOYEES;

Output: 3100

SELECT MIN(HIRE\_DATE) FROM EMPLOYEES;

Output: 17-JUN-87 (oldest date is smallest, latest date is biggest)

SELECT MIN(FIRST\_NAME) FROM EMPLOYEES;

Output: Adam (ASCII Codes are considered, Not much useful)

**Max()** : Returns the Maximum value of the given numeric/character/date column of a table. (NOTE: Character column values are based on ASCII Codes)

SELECT MAX(SALARY) FROM EMPLOYEES;

Output: 24200

SELECT MAX(HIRE\_DATE) FROM EMPLOYEES;

Output: 21-APR-00 (oldest date is smallest, latest date is biggest)

SELECT MAX(FIRST\_NAME) FROM EMPLOYEES;

Output: Winston (ASCII Codes are considered, Not much used)

**Count()** : Returns the number of values of the given numeric/character/date column of a table. (NOTE: Only ***NOT NULL*** values are counted)

SELECT COUNT(\*) FROM EMPLOYEES;

Output: 107

SELECT COUNT(\*) FROM EMPLOYEES WHERE DEPARTMENT\_ID=50;

Output: 45

SELECT COUNT(COMMISSION\_PCT) FROM EMPLOYEES;

Output: 35 (NULL values of the column *commission\_pct* are not counted)

**Rank()** : Calculates the Rank of a value in a group of values based on the values of the *column/exprs* in the *order\_by* clause. Rows with equal values for the ranking criteria will have the same rank.

SELECT RANK(14000) WITHIN GROUP (ORDER BY SALARY) FROM EMPLOYEES;

Output: 101

SELECT RANK(14000) WITHIN GROUP (ORDER BY SALARY DESC) FROM EMPLOYEES;

Output: 6

Assume a sample table with following data:

PRODCODE SOLDQTY AMT

---------- ---------- ----------

1 10 125

1 3 37.5

1 7 87.5

3 2 20.5

4 3 56.25

4 7 131.25

SQL> SELECT RANK(10) WITHIN GROUP (ORDER BY SOLDQTY) FROM SALES;

Output: 6 (SOLDQTY when arranged in *ascending* order)

SQL> SELECT RANK(3) WITHIN GROUP (ORDER BY SOLDQTY) FROM SALES;

Output: 2 (SOLDQTY when arranged in *descending* order)

**ORDER BY** This is used to get the values of the table in a sequence of the given column (increasing / decreasing order). Order by must be the last clause in a select statement. The word ‘*asc*’ or ‘*desc*’ is used after the column name to indicate the order of sorting as ascending or descending. The default is ‘*asc*’ when nothing is indicated. ORDER BY ASC places **NULL values** at the end of the query results. The default placement of NULLs with ORDER BY can be changed with the addition of NULLS FIRST/NULLS LAST to the ORDER BY clause. The ORDER BY clause can contain a **maximum of 255** columns/expressions.

Syntax: Select …………… order by <ColName/Expr> [asc/desc] [NULLS FIRST/NULLS LAST];

SELECT \* FROM DEPARTMENTS ORDER BY DEPARTMENT\_NAME;

SELECT \* FROM DEPARTMENTS ORDER BY DEPARTMENT\_NAME DESC;

(shows in descending order)

If the column contains *Null* values and arranged in ascending order then, by default *Not Null* values come first and *Null* values come later.

SELECT \* FROM DEPARTMENTS ORDER BY MANAGER\_ID;

SELECT \* FROM DEPARTMENTS ORDER BY MANAGER\_ID NULLS FIRST;

SELECT \* FROM DEPARTMENTS WHERE LOCATION\_ID=1700 ORDER BY MANAGER\_ID;

Multiple columns can be used with *Order By* clause. If the first column contains same values then the next column will be considered for order.

SELECT \* FROM DEPARTMENTS ORDER BY LOCATION\_ID, MANAGER\_ID;

SELECT \* FROM EMPLOYEES ORDER BY DEPARTMENT\_ID, SALARY;

SELECT \* FROM EMPLOYEES ORDER BY MANAGER\_ID,SALARY DESC;

SELECT \* FROM EMPLOYEES ORDER BY MANAGER\_ID DESC, SALARY;

SELECT \* FROM EMPLOYEES ORDER BY MANAGER\_ID DESC, SALARY DESC;

**DISTINCT** It is used to get the unique (different) values of the column. The DISTINCT clause can only be used with Select Statement.

Syntax: Select Distinct <ColName(s)>/Expr. From <Tablename>;

SELECT DISTINCT DEPARTMENT\_ID FROM EMPLOYEES;

SELECT DISTINCT MANAGER\_ID FROM EMPLOYEES;

SELECT DISTINCT JOB\_ID FROM EMPLOYEES;

SELECT DISTINCT DEPARTMENT\_ID, JOB\_ID FROM EMPLOYEES;

SELECT DISTINCT DEPARTMENT\_ID, JOB\_ID FROM EMPLOYEES;

(Repeating values are show only once)

**GROUP BY**: It is used to get the values of the table in groups of the given column. This can be used with those columns which contain the same data for multiple rows. Also, it is used in combination with the group functions.

SELECT DEPARTMENT\_ID, SUM(SALARY) FROM EMPLOYEES GROUP BY DEPARTMENT\_ID;

SELECT JOB\_ID, COUNT(\*) FROM EMPLOYEES GROUP BY JOB\_ID;

Only those columns which are used in group by expression can be used in the select statement. We can also apply filter on the data that is being selected for grouping.

SELECT DEPARTMENT\_ID, SUM(SALARY)

FROM EMPLOYEES

WHERE HIRE\_DATE > ’01-APR-05’

GROUP BY DEPARTMENT\_ID;

**HAVING**: It is used to apply a filter on the summary column used in the select statement.

SELECT DEPARTMENT\_ID, SUM(SALARY)

FROM EMPLOYEES

WHERE HIRE\_DATE > ’01-APR-05’

GROUP BY DEPARTMENT\_ID

HAVING SUM(SALARY) > 25000;

**CONSTRAINTS**:

Constraints are limitations created on the column of a table so that, only valid values can be allowed into the table. Constraints can be created on columns of a table at the time of creating a table or constraints can be added later using ALTER command.

Following is the syntax of creating the constraints for the columns of a table while creating table.

CREATE TABLE <TABLENAME> (

<COL1> <DATATYPE>(WIDTH)**[**CONSTRAINT <Constraint\_Name>**]** <CONSTRAINT>,

<COL2> <DATATYPE>(WIDTH)**[**CONSTRAINT <Constraint\_Name>**]** <CONSTRAINT>,

. . . . . .

. . . . . . . . . . .);

In the above syntax:

<Constraint\_Name> : It is the name used to identify the constraint so that we when needed we can DISABLE/ENABLE/DROP the constraint.

<CONSTRAINT>: In this place we write a key word which represents the kind of limitation to be applied on the given column.

NOT NULL : Does not allow *NULL* values into the column

UNIQUE : Does not allow duplicate values for the column (but multiple Nulls are allowed)

PRIMARY KEY: It is combination of UNIQUE and NOT NULL

CHECK : It is constraint used to specify a condition by user. Only those values which satisfy the condition will be allowed into the column.

REFERENCES/FOREIGN KEY: When values of a column of a table are dependent on another table’s column values then we use this constraint.

CREATE TABLE COURSES (

CNO NUMBER(4) CONSTRAINT PK\_CNO\_COURSES PRIMARY KEY,

CNAME VARCHAR2(10) CONSTRAINT NN\_CNAME\_COURSES NOT NULL, FEES NUMBER(5) CONSTRAINT NN\_FEES\_COURSES NOT NULL CONSTRAINT CHK\_FEES\_COURSES CHECK (FEES>=1000));

A table can be created having some column with constraints and some without constraints. Also a column can have more than one constraint.

(In the above table CNO column does not allow null values and duplicate values, cname column does not allow null values, fees column does not allow null values and allows only fees values more than or equal to 1000.)

INSERT INTO COURSES VALUES(101,’ORACLE’,1200);(Row stored)

INSERT INTO COURSES VALUES(101,’C++’,1000);

(101 is repeating so not allowed)

INSERT INTO COURSES VALUES(102,NULL,1200);

(NOT allowed as *CNAME* column cannot contain NULL values)

INSERT INTO COURSES VALUES(102,’C++’,900);

(NOT allowed as *FEES* column cannot values less than 1000)

Following is an example to show the usage of REFERENCES constraint

CREATE TABLE STUDENTS (

RNO NUMBER(4),

STNAME VARCHAR2(10),

CID NUMBER(4) CONSTRAINT REF\_CID\_STUDENTS REFERENCES COURSES(CNO));

This table does not allow any value into *CID* column which does not exist in *CNO* column of *COURSES* table.

INSERT INTO STUDENTS VALUES(1001,’RAJESH’,101);

INSERT INTO STUDENTS VALUES(1002,’RAJESH’,105);

(NOT allowed as *CNO* column does not contain 105 course in courses table)

**Adding constraints for the columns of an existing table using *ALTER TABLE*.**

Adding the NOT NULL Constraint:

ALTER TABLE <TABLENAME> MODIFY(

<ColName> <DATATYPE>(WIDTH) **[**CONSTRAINT <Constraint\_Name>**]** NOT NULL);

ALTER TABLE STUDENTS MODIFY(STNAME VARCHAR2(10) CONSTRAINT NN\_STNAME\_STUDENTS NOT NULL);

Adding the UNIQUE Constraint:

ALTER TABLE <TABLENAME> ADD CONSTRAINT <Constraint\_Name> UNIQUE(<ColName>);

ALTER TABLE STUDENTS ADD CONSTRAINT UNQ\_RNO\_STUDENTS UNIQUE(RNO);

Adding the PRIMARY KEY Constraint:

ALTER TABLE <TABLENAME> ADD CONSTRAINT <Constraint\_Name> PRIMARY KEY(<ColName>);

ALTER TABLE STUDENTS ADD CONSTRAINT PK\_RNO\_STUDENTS PRIMARY KEY(RNO);

Adding the CHECK Constraint:

ALTER TABLE <TABLENAME> ADD CONSTRAINT <Constraint\_Name> CHECK(<Condition>);

ALTER TABLE STUDENTS ADD CONSTRAINT CHK\_RNO\_STUDENTS CHECK (RNO>=1001);

Adding the REFERENCES/FOREIGN KEY Constraint:

ALTER TABLE <TABLENAME> ADD CONSTRAINT <Constraint\_Name> FOREIGN KEY (<ChildTableColumnName>) REFERENCES <ParentTable>(<ColName>);

ALTER TABLE STUDENTS ADD CONSTRAINT FK\_CID\_STUDENTS FOREIGH KEY(CID) REFERENCES COURSES(CNO);

The information about the constraints created for various column of the table can be observed from a system defined table USER\_CONSTRINTS

SELECT \* FROM USER\_CONSTRAINTS;(This table contains many columns)

SELECT CONSTRAINT\_NAME, TABLE\_NAME, SEARCH\_CONDITION FROM USER\_CONSTRAINTS WHERE TABLE\_NAME=’STUDENTS’;

How to Enable/Disable/Drop constraints

ALTER TABLE <TABLENAME> ENABLE/DISABLE/DROP CONSTRAINT <Constraint\_Name>;

ALTER TABLE STUDENTS DISABLE CONSTRAINT PK\_RNO\_STUDENTS;

INSERT INTO STUDENTS VALUES(1001,’PRAKASH’,102);

(Although *Rno* 1001 exists this will be inserted into the table as a duplicate value)

When a constraint is disabled, the data which is not allowed by the constraint also will be allowed for the column. However, we have to correct the data according to the constraint which it is specified with to enable the constraint.

ALTER TABLE STUDENTS ENABLE CONSTRAINT PK\_RNO\_STUDENTS;

(This will NOT enable the constraint as the column contains data which is not valid. The data has to be changed to valid value so as to enable the constraint)

UPDATE STUDENTS SET RNO=1008 WHERE NAME=’PRAKASH’;

ALTER TABLE STUDENTS ENABLE CONSTRAINT PK\_RNO\_STUDENTS;

**SET OPERATORS**

In reference to general maths consider the following values for the set A and B:

SET A = {1, 3, 4, 6, 8, 9} SET B = {1, 2, 4, 5, 7, 9, 10}

Now

A Union B : {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

A Union All B : {1, 3, 4, 6, 8, 9, 1, 2, 4, 5, 7, 9, 10} (1st set followed by 2nd set)

A Intersect B : {1, 4, 9}

A Minus B : {3, 6, 8}

B Minus A : {2, 5, 7, 10}

Certain situations may arise where we may want to combine the output given by more than one queries as a single output. *SET Operators* are useful in such cases.

Oracle supports four (4) SET Operators:

UNION UNION ALL INTERSECT MINUS

The set operators are used with the following syntax:

<Select Statement> <SET Operator> <Select Statement>;

Pre requisite is that both queries should give same number of columns and same sequence of columns in the output.

UNION: Combines the output of multiple select statements and shows all values by considering duplicating values only once. The unique values of the combined data will be displayed.

SELECT SALARY FROM EMPLOYEES WHERE DEPARTMENT\_ID=60

UNION

SELECT SALARY FROM EMPLOYEES WHERE DEPARTMENT\_ID=80;

UNION ALL: Combines the output of multiple select statements including the duplicate values. The values of first query are first listed followed by the values of the second query.

SELECT SALARY FROM EMPLOYEES WHERE DEPARTMENT\_ID=60

UNION ALL

SELECT SALARY FROM EMPLOYEES WHERE DEPARTMENT\_ID=80;

INTERSECT: Shows only the common values of both queries.

SELECT SALARY FROM EMPLOYEES WHERE DEPARTMENT\_ID=60

INTERSECT

SELECT SALARY FROM EMPLOYEES WHERE DEPARTMENT\_ID=80;

MINUS: Shows only those rows that are left over after removing the common values of both queries from the first output.

SELECT SALARY FROM EMPLOYEES WHERE DEPARTMENT\_ID=60

MINUS

SELECT SALARY FROM EMPLOYEES WHERE DEPARTMENT\_ID=80;

SELECT SALARY FROM EMPLOYEES WHERE DEPARTMENT\_ID=80

MINUS

SELECT SALARY FROM EMPLOYEES WHERE DEPARTMENT\_ID=60;

Following is another example which shows the usage of the set operators:

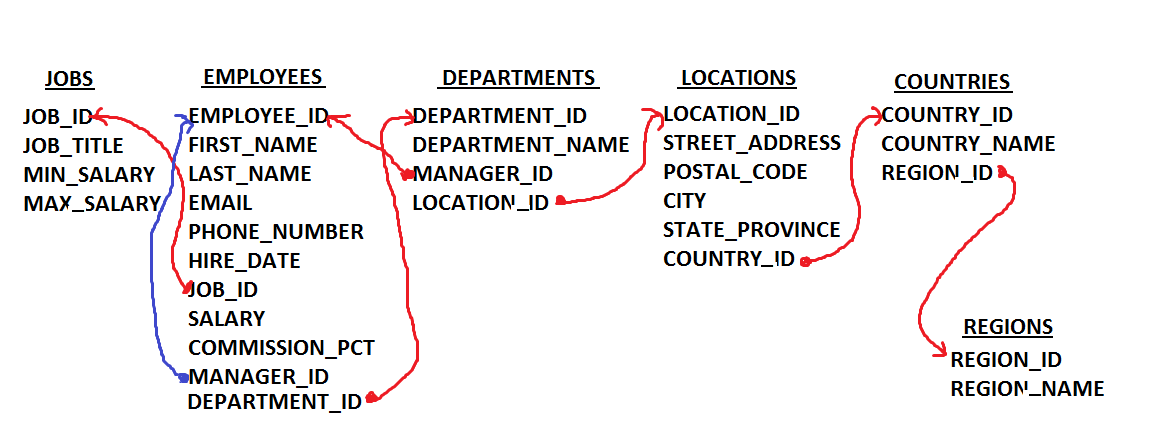
Assume that X and Y are two companies that have conducted campus interviews at a place and have short listed some of the students for their organization. Company\_X, Company\_Y are the tables which contains the list of students selected

Consider the following tables:

Company\_X Company\_Y

| **Sno** | **Name** | **Status** |
| --- | --- | --- |
| 1 | Raju | Selected |
| 2 | Suresh | Selected |
| 3 | Harish | Not Selected |
| 4 | Kiran | Not Selected |
| 5 | Vijay | Selected |
| 6 | Prasad | Selected |
| 7 | Laxman | Not Selected |
| 8 | Uday | Selected |
| 9 | Mohan | Not Selected |
| 10 | Anil | Selected |
| **Sno** | **Name** | **Status** |
| 1 | Raju | Selected |
| 2 | Suresh | Selected |
| 3 | Harish | Not Selected |
| 4 | Kiran | Not Selected |
| 5 | Vijay | Selected |
| 6 | Prasad | Selected |
| 7 | Laxman | Not Selected |
| 8 | Uday | Selected |
| 9 | Mohan | Not Selected |
| 10 | Anil | Selected |

Shows only those rows that are left over after removing the common values of both queries from the first output.

**Structure of sample tables**

**SUB QUERIES**

A query within another query is called as a Sub Query. Sub Query (or) Inner query (or) Nested query is a query in a query. SQL subquery is usually added in the WHERE Clause of the SQL statement. Commonly, a subquery is used when you know how to search for a value using a SELECT statement, but do not know the exact value in the database.

Sub queries are an alternate way of returning data from same/multiple tables.

Subqueries can be used with the following SQL statements along with the comparision operators like =, <, >, >=, <= etc.

The output of the inner query is used as an input for the outer query.

Outerquery......(innerqury)......outer query continued;

*Example 1*: To find the NAME of the employee who has the highest salary:

SELECT MAX(SALARY) FROM EMPLOYEES;

Output is 24000

SELECT FIRST\_NAME FROM EMPLOYEES WHERE SALARY=24000;

Output is Steven

Now the above two queries can be combined in one statement using a subquery.

SELECT FIRST\_NAME FROM EMPLOYEES

WHERE SALARY=(SELECT MAX(SALARY) FROM EMPLOYEES);

Output is Steven

*Example 2*: To find the number of employees in Sales department:

SELECT DEPARTMENT\_ID FROM DEPARTMENTS WHERE UPPER(DEPARTMENT\_NAME)=’SALES’;

Output is 80

SELECT COUNT(\*) FROM EMPLOYEES WHERE DEPARTMENT\_ID=80;

Output is 34

Now the above two queries can be combined in one statement using a subquery.

SELECT COUNT(\*) FROM EMPLOYEES WHERE DEPARTMENT\_ID=

(SELECT DEPARTMENT\_ID FROM DEPARTMENTS WHERE UPPER(DEPARTMENT\_NAME)=’SALES’);

Output is 34

*Example 3*: To find the No. of employees who has salary less than the average salary of Sales department employees:

SELECT DEPARTMENT\_ID FROM DEPARTMENTS WHERE UPPER(DEPARTMENT\_NAME)=’SALES’;

Output is 80

SELECT AVG(SALARY) FROM EMPLOYEES WHERE DEPARTMENT\_ID=80;

Output is 7536.79245

SELECT COUNT(\*) FROM EMPLOYEES WHERE SALARY<7536.79245;

Output is 57

Now the above three queries can be combined in one statement using a subquery.

SELECT COUNT(\*) FROM EMPLOYEES WHERE SALARY<

(SELECT AVG(SALARY) FROM EMPLOYEES WHERE DEPARTMENT\_ID=

(SELECT DEPARTMENT\_ID FROM DEPARTMENTS WHERE UPPER(DEPARTMENT\_NAME)=’SALES’));

Output is 57

Nested sub-query: A subquery within another subquery

EX: Name of the course from which highest amount is collected (courses and students tables)

select cname from courses where ccode=(

select course\_no from (select course\_no, sum(paid\_amt) tot from students group by course\_no)

where tot=(select max(tot) from (select course\_no, sum(paid\_amt) tot from students group by course\_no)))

Inner query returning multiple rows should be compared using "in" operator

select count(\*) from countries where region\_id in (

select region\_id from regions where region\_name in ('Europe','Asia'));

**JOINS**

Joins are used to get a combined output from multiple tables using a relation between them. To create a join the tables must have at least one column which contains relational data.

Joins are of 4 types:

* Simple Join
* Self / Inner join
* Outer Join
* Cartesian join

**Simple Join**

This type of join gives combined output of data from multiple tables using a relational column.

Ex: To list the employee name with his department name we use a join condition between department\_id of both tables (employees, departments)

SELECT EMPLOYEES.FIRST\_NAME,DEPARTMENTS.DEPARTMENT\_NAME

FROM EMPLOYEES, DEPARTMENTS

WHERE EMPLOYEES.DEPARTMENT\_ID=DEPARTMENTS.DEPARTMENT\_ID;

Instead of using the table name before column name every time, we can use a table alias name. Table alias is defined after the table name in the *from* clause of a select statement. Alias names are temporary names for that statement only.

SELECT E.EMPLOYEE\_ID, E.FIRST\_NAME, D.DEPARTMENT\_NAME

FROM EMPLOYEES E, DEPARTMENTS D

WHERE E.DEPARTMENT\_ID = D.DEPARTMENT\_ID;

(E is the alias name for *EMPLOYEES* and D is the alias for *DEPARTMENTS* table)

Ex: From the sample tables (already installed) observe the following:

Employees table department\_id is related to Departments table department\_id

Departments table location\_id is related to Locations table location\_id

Locations table country\_id is related to Countries table Country\_id

Countries table region\_id is related to Regions table region\_id

Now to display the First\_Name and Region\_Name the query is given as follows:

SELECT E.FIRST\_NAME, R.REGION\_NAME

FROM EMPLOYEES E, DEPARTMENTS D, LOCATIONS L, COUNTRIES C, REGIONS R

WHERE

E.DEPARTMENT\_ID = D.DEPARTMENT\_ID AND

D.LOCATION\_ID = L.LOCATION\_ID AND

L.COUNTRY\_ID = C.COUNTRY\_ID AND

C.REGION\_ID = R.REGION\_ID;

**Self/Inner Join**

This type of join is used to create a relation between the columns of same table. (Therefore called inner/self join). In certain situations we have the related data in the same table but unless a relation is created between the columns we cannot see the output. To implement a self-join we use multiple alias names for a single table and create a join between the columns by referring each column with a different alias name.

Ex: To list the employee name with their manager names.

Actually First\_Name itself contains the names of employees and managers. This information can be identified by Manager\_Id column of each row. When we compare the Manager\_id column value in Employee\_Id column, we get the manager name. (Note that all information is present in one single table)

SELECT X.FIRST\_NAME “EMP\_NAME” , Y.FIRST\_NAME “MANAGER\_NAME”

FROM EMPLOYEES X, EMPLOYEES Y

WHERE X.MANAGER\_ID = Y.EMPLOYEE\_ID;

**Outer Join**

This kind of join is an extension for the self or simple join. This join shows the result of a self or simple join along with even those rows for which no rows from the other satisfies the condition. The usage of a *plus* (+) sign when given after the column being compared (in the where condition) makes it an outer join.

Left Outer Join: If the plus sign is used on the LHS of the comparison.

Right Outer Join: If the plus sign is used on the RHS of the comparison.

SELECT E.EMPLOYEE\_ID, E.FIRST\_NAME, D.DEPARTMENT\_NAME

FROM EMPLOYEES E, DEPARTMENTS D

WHERE E.DEPARTMENT\_ID = D.DEPARTMENT\_ID(+);

(Observe in the output that one of the employee (178 Kimberely) does not have a department\_id. A Simple join did not show his record in the output, whereas here it shows)

SELECT X.FIRST\_NAME “EMP\_NAME”, Y.FIRST\_NAME “MANAGER\_NAME”

FROM EMPLOYEES X, EMPLOYEES Y

WHERE X.MANAGER\_ID = Y.EMPLOYEE\_ID(+);

(RIGHT OUTER JOIN: The above query shows the name of employee along with their manager name for all rows including the first row (100 Steven) for whom there is no manager\_id)

SELECT X.FIRST\_NAME “EMP\_NAME”, Y.FIRST\_NAME “MANAGER\_NAME”

FROM EMPLOYEES X, EMPLOYEES Y

WHERE X.MANAGER\_ID(+) = Y.EMPLOYEE\_ID;

(LEFT OUTER JOIN: This query shows the name of employee along with their manager name for all rows including the those managers who do not have employee under them)

**Cartesian Join**

This type of join gives all possible combination of the tables being joined. In this join we refer the data of more than one table but **do not** give any ‘Where’ condition.

Example:

BANKS CUSTOMERS

| **BankName** | **Intper** |
| --- | --- |
| ICICI | 9.1 |
| SBI | 8.85 |
| HDFC | 9.25 |
| **CustName** | **Amount** |
| Raju | 250000 |
| Suresh | 150000 |
| Kiran | 170000 |
| Vijay | 230000 |
| Prasad | 200000 |

Required output is to show interest amount and the net amount when each of the customer amounts are tested with each of the banks (all possible combinations, all customers with all banks for 5 Years)

SELECT C.CUSTNAME, C.AMOUNT, B.BANKNAME, B.INTPER, (C.AMOUNT\*5\*B.INTPET)/100 as “InterestAmt”, C.AMOUNT+((C.AMOUNT\*5\*B.INTPER)/100) as “NetAmount”

FROM BANKS B, CUSTOMERS C;

CUSTNAME AMOUNT BANKNAME INTPER InterestAmt NetAmount

---------- ---------- ---------- ---------- ----------- ----------

RAJU 25000 ICICI 9.1 11375 36375

RAJU 25000 SBI 8.85 11062.5 36062.5

RAJU 25000 HDFC 9.25 11562.5 36562.5

SURESH 15000 ICICI 9.1 6825 21825

SURESH 15000 SBI 8.85 6637.5 21637.5

SURESH 15000 HDFC 9.25 6937.5 21937.5

KIRAN 17000 ICICI 9.1 7735 24735

KIRAN 17000 SBI 8.85 7522.5 24522.5

KIRAN 17000 HDFC 9.25 7862.5 24862.5

VIJAY 23000 ICICI 9.1 10465 33465

VIJAY 23000 SBI 8.85 10177.5 33177.5

VIJAY 23000 HDFC 9.25 10637.5 33637.5

PRASAD 20000 ICICI 9.1 9100 29100

PRASAD 20000 SBI 8.85 8850 28850

PRASAD 20000 HDFC 9.25 9250 29250

**DATABASE OBJECTS**

A Database object is an entity created at the database server permanently. These objects are generally created using CREATE command. A table is a basic Database Object. Most of the other database objects are based on Tables. Following are some of the examples of Database Objects.

Table

Synonym

View

Sequence

Index

User

Partition

Type

Procedure

Function

Package

Trigger

The last mentioned 4 are created from PL/SQL others can be created from both SQL or PL/SQL.

**SYNONYM**:

A synonym is a Database object which is like a permanent alias name for the table. Once a synonym is created, we can use it to perform any DML operation (Insert/Update/Delete) through it. Generally synonym is for the entire table. Such synonyms are mainly used to reduce long names of tables to shorter names. A synonym can be granted to other users as well. When a synonym s granted, the grantee can use the synonym to perform the given privileges on the table using the synonym. TAB is one such type of synonym created by system for every user.

CREATE SYNONYM <SYNONYM\_NAME> FOR <TABLENAME>;

Eg: CREATE SYNONYM EMP FOR EMPLOYEES;

Select \* from employees;

Select \* from emp;

Insert into employees values(. . . . .);

Insert into emp values(. . . . . );

Delete from employees where . . . . . ;

Delete from emp where . . . . . .;

Update employees set . . . . . . .;

Update emp set . . . . . . . . .;

Create synonym granted for user\_tab\_privs\_made;

Select \* from user\_tab\_privs\_made;

Select \* from granted;

Create synonym grants\_got for user\_tab\_privs\_recd;

Select \* from user\_tab\_privs\_recd;

Select \* from grants\_got;

USER A> Grant select, insert, delete on emp to B;

USER B> select \* from A.emp;

USER B> delete from A.emp where . . . . . . ;

A Synonym cannot be altered. We can only drop a synonym and recreate it. The information about the synonyms can be obtained by using the table USER\_SYNONYMS

SELECT \* FROM USER\_SYNONYMS;

A Synonym cab be removed if not needed by using

DROP SYNONYM <SYNONYM\_NAME>;

**VIEW**

A View is another database object similar to synonym but with much more features than it. A view can be created on entire table or specific portion of a table or for columns of multiple tables. A view is a representation for a select statement. Whenever we refer the view, the select statement corresponding to it will be executed and the output is produced.

A View can represent :

* a simple select statement with all or specific columns or expressions based on a table
* a statement with a where clause, order by clause, group by clause
* a statement having set operators
* a statements with join conditions
* a statement with sub queries etc.

Such view if refers data from one table then we can perform the DML operations also using the view. A View can be granted to other users so that the other user can read specific data which is required by them and restrict access to other columns/rows of data.

CREATE VIEW <VIEW\_NAME> AS <SELECT STATEMENT>;

**SEQUENCE**

A Sequence is a database object used to create a series of values at the database server so that this sequence of values can be inserted into the table from the sequence. It is an object from which multiple users may generate unique integers. You can use sequences to automatically generate primary key values. One user can never acquire the sequence number generated by another user. Once a sequence value is generated by one user, that user can continue to access that value regardless of whether the sequence is incremented by another user. Once a sequence is created, you can access its values in SQL statements with the pseudo column, CURRVAL and NEXTVAL.

CURRVAL: It is a pseudo column, which returns the current value of the sequence.

NEXTVAL: It is a pseudo column, which increments the sequence and returns the new value.

A sequence is created with the following syntax:

CREATE SEQUENCE <Sequence\_Name>

INCREMENT BY <VALUE>

START WITH <VALUE>

MAXVALUE <VALUE>

MINVALUE <VALUE>

CYCLE/NOCYCLE

CACHE <No.>/NOCACHE;

Eg: Consider a table ***STUDENTS*** as follows:

| **Rno** | **Name** | **Course** | **Fees** | **Contact** |
| --- | --- | --- | --- | --- |
| 101 | Kiran | C++ | 1000 | 9836363535 |
| 102 | Vijay | Java | 5000 | 7386373636 |

CREATE SEQUENCE SQ1

INCREMENT BY 1

START WITH 103

MINVALUE 101

MAXVALUE 110

CYCLE

CACHE 4;

The above statement creates a sequence which begins for the first time with 103 and for every cycle of repetition it begins with 101. The sequence is incremented by 1 (as we have indicated) for every reference of NEXTVAL. It ends at 110 from where the cycle restarts. CACHE 4 indicates that the server responds to client with 4 value of the sequence so that for next 3 times of reference, the server need not respond. The client system has the next 3 value in its CACHE memory. CACHE reduces the network traffic between the client and server and improves the performance of both client and server system.

Now, while inserting the records the command may be given as:

INSERT INTO STUDENTS VALUES (SQ1.NEXTVAL, ‘&NAME’, ‘&COURSE’, &FEES, &CONTACT);

To know the current value of the sequence the pseudo column CURRVAL can be referred.

SELECT SQ1.CURRVAL FROM DUAL;

SELECT \* FROM USER\_SEQUENCES;

This table shows list of sequences created by the user along with the details of each sequence.

A sequence can be dropped if it is not needed any more.

DROP SEQUENCE <SEQUENCE\_NAME>;

DROP SEQUENCE SQ1;

**INDEX**

This is a database object used for fast accessing of data of the tables. An index is very much useful in tables containing large volumes of data. An index creates an entry for each value that appears in the indexed columns. By default, Oracle creates B-tree indexes. An index can be created on any of the columns of the table which is not created with a *UNIQUE* or *PRIMARY KEY* constraint. Unlike any other database objects, we do not refer the index name in any of the queries. It is automatically activated whenever a query refers a filter (where <condition>) based on the indexed column.

CREATE INDEX <INDEX\_NAME> ON <TABLENAME>(<COLUMN\_NAME>);

Eg: CREATE INDEX ID1 ON EMPLOYEES(SALARY);

SELECT \* FROM EMPLOYEES WHERE SALARY=10000;

Now that the above query refers the indexed column (salary), the index ID1 is activated to search for the value 10000. Searching data with an indexed column will be very much faster when compared to the searching on a column on which index is not created.

Index can be created on multiple columns also.

CREATE INDEX ID1 ON EMPLOYEES(DEPARTMENT\_ID,SALARY);

SELECT \* FROM EMPLOYEES WHERE SALARY=10000;

The information about the index names and their corresponding columns is shown in USER\_INDEXES.

SELECT \* FROM USER\_INDEXES; (Note that this is a table with lot of columns in it)

SELECT INDEX\_NAME, TABLE\_NAME FROM USER\_INDEXES;

An index cannot be created on such a column which has been created with a UNIQUE or PRIMARY KEY constraint. When the constraint is added to the column, an index with the same name as constraint name is created.

CREATE TABLE SAMPLE(

IDNO NUMBER(5) CONSTRAINT PK\_ID\_SAMPLE PRIMARY KEY,

NAME VARCHAR2(10));

SELECT INDEX\_NAME, TABLE\_NAME FROM USER\_INDEXES WHERE TABLE\_NAME = ‘SAMPLE’;

An index can be dropped if it is not needed any more

DROP INDEX <INDEX\_NAME>;

**PL/SQL**

**Procedural Language using Structured Query Language**

PL/SQL is a combination of SQL along with the procedural features of programming languages. It was developed by Oracle Corporation in the early 90’s to enhance the capabilities of SQL. PL/SQL includes procedural language elements such as conditions and loops. You can declare and handle PL/SQL units such as constants and variables, procedures and functions, types and database triggers that are stored in the database. These elements are for reuse by applications that use any of the Oracle Database programmatic interfaces. You can also handle exceptions (runtime errors).

The PL/SQL Engine:

Oracle uses a PL/SQL engine to processes the PL/SQL statements. A PL/SQL code can be stored in the client system (client-side) or in the database (server-side). A PL/SQL program contains SQL and PL/SQL statements. The SQL statements are executed by the database server. The PL/SQL statements are executed by the PL/SQL engine (PL/SQL compiler).

Structure of a PL/SQL program

/\* Comment line \*/

Declare

..........

Variable Declarations...

..........

Begin

..........

..........

.Actual program.

..........

..........

**[**Exception**]**

..........

..........

End;

**/**

A program generally contains 4 sections.

The “*Declare*” section contains the variable declarations.

The “*Begin*” section contains the actual statements and the logic of the PL/SQL program.

The “*Exception*” section is optional. It contains the user defined or system defined exceptions.

The “*End*” is last part of a PL/SQL program and it end with a forward slash ‘/’symbol. This symbol will execute the program when it is loaded into the memory.

/\* Comment \*/ Any text enclosed within these symbols is treated as a comment. The comment symbols are used to write the description for the reader of the program. The comment can occur at any part of the program. PL/SQL engine omits the contents of a comment.

PL/SQL is also **NOT** case sensitive language. The statements of the program may be typed in capitals or small. However the character data is always case sensitive.

**Declaring Variables**

A variable is declared along with a data type and width. All SQL data types are supported as it is along with a new data type called BOOLEAN. This data type is used to store TRUE or FALSE. It occupies 1 Bit of memory.

Each variable is to be declared in a different line. Even if the variables are of the same data type or width they cannot be declared in the same line.

<Variable> <Datatype>(Width);

Ex: N Number(4);

Avrg number(6,2); (6 is precision and 2 is the scale)

Name varchar2(10);

ManfDt Date;

Status Boolean;

**Some special operators in PL/SQL.**

:= This symbol is used to assign values to variables.

= This symbol is used to check a condition.

|| This symbol is used to concatenate (join) two or more values into a single line output.

**How to perform input and output?**

To accept the input from the user we use & symbol (same as in SQL). This symbol will ask value for a variable and store in it.

Eg: N:=&N;

Name:=’&Student\_Name’;

ManfDt:=’&Manufacture\_Date’;

To display the calculated values of variables or messages on the output screen we use:

DBMS\_OUTPUT.PUT\_LINE(‘MESSAGE’/VARIABLE);

Eg: DBMS\_OUTPUT.PUT\_LINE(‘HELLO WELCOME TO PL/SQL PROGRAMMING’);

DBMS\_OUTPUT.PUT\_LINE(NAME);

DBMS\_OUTPUT.PUT\_LINE(‘THE TOTAL IS’||TOT);

**Performing calculations:**

Calculations can be performed by a direct expression or by using an SQL statement.

<Variable>**:=**<value>/<expression>;

(or)

Select <value>/<expression> into <variable> from dual;

Eg: Amount := Cost \* Qty;

(or)

Select Cost \* Qty into Amount from dual;

**How to type and execute a PL/SQL program?**

SQL> ED filename

This will open a new file in the notepad (the default editor).

Type the program, save it (File🡪Save) and close it (File🡪Exit)

SQL> @filename

This will load the PL/SQL program into the memory. The last line of the program / will execute the program.

SQL> SET SERVEROUTPUT ON

This statement will set an environment value to display the output given from the server. This command is given only once for an SQL session.

SQL> SET VERIFY OFF

After accepting an input it is shown to user as

Old: <user given statement>

New: <the substituted value statement>

This command (SET VERIFY OFF) will not display the verification after accepting the values from the user.

As these set commands are only for the current session of SQL, therefore these statements may be written in the program before the “*Declare*” section directly.

/\* PROGRAM TO PERFORM ARITHMETIC CALCULATIONS \*/

DECLARE

X NUMBER(4);

Y NUMBER(4);

Z NUMBER(5);

BEGIN

X:=&X;

Y:=&Y;

Z:=X+Y;

DBMS\_OUTPUT.PUT\_LINE('THE SUM IS '||Z);

END;

/

/\* PROGRAM TO CALCULATE AREA AND CIRCUMFERENCE OF A CIRCLE \*/

DECLARE

RAD NUMBER(5,1);

AREA NUMBER(7,1);

CIRC NUMBER(7,1);

BEGIN

RAD:=&RADIUS;

AREA:=3.14\*RAD\*RAD; /\* AREA:=3.14\*POWER(RAD,2); \*/

SELECT 3.14\*RAD\*RAD INTO AREA FROM DUAL;

CIRC:=2\*3.14\*RAD;

DBMS\_OUTPUT.PUT\_LINE('THE AREA OF CIRCLE IS'||AREA);

DBMS\_OUTPUT.PUT\_LINE('CIRCUMFERENCE OF CIRCLE IS'||CIRC);

END;

/

PL/SQL program extracting data from the database is the actual implementation of a program. A PL/SQL program cannot display the output of a “select” statement. So the values extracted from the database are stored in the temporary variables (as declared in the declare section) and then displayed/used for the process.

/\* PROGRAM TO ASK EMPLOYEE NUMBER AND SHOW OTHER DETAILS FROM THE EMPLOYEES TABLE\*/

DECLARE

ENO NUMBER(5);

ENA VARCHAR2(15);

JOB VARCHAR2(10);

SAL NUMBER(7,1);

BEGIN

ENO:=&EMPLOYEE\_NUMBER;

SELECT FIRST\_NAME, JOB\_ID, SALARY INTO ENA, JOB, SAL FROM EMPLOYEES WHERE EMPLOYEE\_ID=ENO;

DBMS\_OUTPUT.PUT\_LINE('EMPLOYEE NAME IS '||ENA);

DBMS\_OUTPUT.PUT\_LINE('EMPLOYEE JOB ID IS '||JOB);

DBMS\_OUTPUT.PUT\_LINE('EMPLOYEE SALARY IS '||SAL);

END;

/

**ROWTYPE and COLUMN TYPE Variables**

The variables declared with a specific data type and width in a PL/SQL program is to store the data that is fetched from the database. If the data type or width of the columns of the table is changed, then all those programs which refer the data from that table have to be modified to suit the same data type or width as declared in the table.

Declaring a COLUMN TYPE or ROWTYPE variable is a solution to such problem.

A **Column Type** variable is one such variable which is declared with the same data type and width as in the table structure description. So, whenever the program is executed, the variable is created with the same data type and width as in the table. This provides the optimal utilization of the memory. A column type variable is declared as follows;

<Variable> <TableName>.<ColName>%TYPE;

Ex: Empno EMPLOYEES.EMPLOYEE\_ID%TYPE;

This will create a variable “Empno” with the same data type and width as “Employee\_id” column of “Employees” table.

A **Row Type** variable is a *group variable* which contains variables as similar to all the columns of the table with the same data type and width as in the table. Such a “Row Type” variable avoids the explicit declaration of many temporary variables in the program. Also while fetching data from the table it become a very simple to store them in the temporary variables. A Row type variable is declared as follows:

<Variable> <TableName>%ROWTYPE; (Note that ROWTYPE is a single word)

Ex: X EMPLOYEES%ROWTYPE;

X

| X.Employee\_id | X.First\_Name | X.Last\_Name | X.Email | X.Phone\_Number | X.Hire\_date | X.Job\_id | X.Salary | X.Commission\_pct | X.Manager\_id | X.Department\_id |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |

The reference of any of the variables is given as X.<ElementName>

Following examples show the usage of the Rowtype and Column type variables.

/\* PROGRAM TO ASK EMPLOYEE NUMBER AND SHOW OTHER DETAILS FROM THE EMPLOYEES TABLE **USING COLUMN TYPE VARIABLES** \*/

DECLARE

ENO EMPLOYEES.EMPLOYEE\_ID%TYPE;

ENA EMPLOYEES.FIRST\_NAME%TYPE;

JOB EMPLOYEES.JOB\_ID%TYPE;

SAL EMPLOYEES.SALARY%TYPE;

BEGIN

ENO:=&EMPLOYEE\_ID;

SELECT FIRST\_NAME, JOB\_ID, SALARY INTO ENA,JOB,SAL FROM EMPLOYEES WHERE EMPLOYEE\_ID=ENO;

DBMS\_OUTPUT.PUT\_LINE('EMPLOYEE NAME IS '||ENA);

DBMS\_OUTPUT.PUT\_LINE('EMPLOYEE JOB ID IS '||JOB);

DBMS\_OUTPUT.PUT\_LINE('EMPLOYEE SALARY IS '||SAL);

END;

/

/\* PROGRAM TO ASK DEPARTMENT\_ID AND SHOW THE DEPARTMENT NAME, LOCATION ID FROM DEPARTMENTS TABLE **USING ROWTYPE VARIABLE** \*/

DECLARE

D DEPARTMENTS.DEPARTMENT\_ID%TYPE;

X DEPARTMENTS%ROWTYPE;

BEGIN

D:=&DEPARTMENT\_NUMBER;

SELECT \* INTO X FROM DEPARTMENTS WHERE DEPARTMENT\_ID=D;

DBMS\_OUTPUT.PUT\_LINE('DEPARTMENT NAME IS '||X.DEPARTMENT\_NAME);

DBMS\_OUTPUT.PUT\_LINE('LOCATION ID IS '||X.LOCATION\_ID);

END;

/

**Control Statements / Constructs**

The flow of execution of a program is controlled by a control statement or constructs. The following control statements are supported by oracle PL/SQL.

Conditional Statements

* If – Then – Else
* Case

Looping Statements

* For Loop
* While Loop
* Simple Loop

**Conditional Statements**

IF – THEN – ELSE

This construct can be given in 2 forms:

* + Simple If – Then – Else
  + Nested If – Then – Else

**Simple If – Then – Else**

The usage of a simple IF – THEN – ELSE is given below.

IF <condition> THEN

One or more Statements

ELSE

One or more statements

END IF;

This construct is used to check a condition and execute a block of statement when the condition is TRUE or if the condition is FALSE then another block of statements can be executed. However “ELSE” is an optional block and every “IF” should have an “END IF”.

**Nested If – Then – Else**

The usage of a *nested* IF – THEN – ELSE is given below.

IF <condition> THEN

One or more Statements

ELSIF <condition> THEN

One or more statements

ELSE <condition> THEN

One or more statements

END IF;

In this construct we check a condition and execute a block of statement when the condition is TRUE or if the condition is FALSE then another condition is checked. The same rule is again applied for this condition. “ELSE” is an optional block in this method also and every “IF” should have an “END IF”.

/\* PROGRAM USING SIMPLE IF-ELSE.

PROGRAM TO CHECK IF A GIVEN VALUE IS EVEN OR ODD NUMBER \*/

DECLARE

N NUMBER(6);

R NUMBER(1);

BEGIN

N:=&N;

R:=MOD(N,2);

/\* MOD() Calculates the remainder of a division.

We can also write it as SELECT MOD(N,2) INTO R FROM DUAL; \*/

IF R=0 THEN

DBMS\_OUTPUT.PUT\_LINE('EVEN NUMBER');

ELSE

DBMS\_OUTPUT.PUT\_LINE('ODD NUMBER');

END IF;

END;

/

/\* PROGRAM USING NESTED IF ELSE (**ELSIF**)\*/

DECLARE

AGE1 NUMBER(3);

AGE2 NUMBER(3);

BEGIN

AGE1:=&FIRST\_PERSON\_AGE;

AGE2:=&SECOND\_PERSON\_AGE;

IF AGE1>AGE2 THEN

DBMS\_OUTPUT.PUT\_LINE('FIRST PERSON IS ELDER THAN SECOND');

**ELSIF** AGE1<AGE2 THEN

DBMS\_OUTPUT.PUT\_LINE('SECOND PERSON IS ELDER THAN FIRST');

ELSE

DBMS\_OUTPUT.PUT\_LINE('BOTH ARE OF THE SAME AGE');

END IF;

END;

/

/\* IF ELSE WITH COMPOUND CONDITIONS USING **AND**, **OR** OPERATORS\*/

DECLARE

M1 NUMBER(3);

M2 NUMBER(3);

M3 NUMBER(3);

BEGIN

M1:=&SUBJECT1\_MARKS;

M2:=&SUBJECT2\_MARKS;

M3:=&SUBJECT3\_MARKS;

IF M1>=35 AND M2>=35 AND M3>=35 THEN

DBMS\_OUTPUT.PUT\_LINE('RESULT PASS');

ELSE

DBMS\_OUTPUT.PUT\_LINE('RESULT FAIL');

END IF;

/\* USING **OR** OPERATOR \*/

IF M1<35 OR M2<35 OR M3<35 THEN

DBMS\_OUTPUT.PUT\_LINE('RESULT FAIL');

ELSE

DBMS\_OUTPUT.PUT\_LINE('RESULT PASS');

END IF;

END;

/

**Case**

This construct is used when multiple individual values are to be compared for a single variable. Each “When” indicates a value to be compared for the variable. If the variable value matches then that block statements are executed. When none of the values are true then the statements of the “ELSE” block are executed. “ELSE” is an optional block. Every “CASE” block ends with an “END CASE”.

CASE <variable>

WHEN value1 THEN

statements

WHEN value2 THEN

statements

.....

.....

ELSE

else\_statements

END CASE;

An example program is given below

/\* USAGE OF **CASE – END CASE** \*/

DECLARE

GRADE CHAR(1);

BEGIN

GRADE:='&GRADE';

CASE GRADE

WHEN 'A' THEN DBMS\_OUTPUT.PUT\_LINE('EXCELLENT');

WHEN 'B' THEN DBMS\_OUTPUT.PUT\_LINE('VERY GOOD');

WHEN 'C' THEN DBMS\_OUTPUT.PUT\_LINE('GOOD');

WHEN 'D' THEN DBMS\_OUTPUT.PUT\_LINE('FAIR');

WHEN 'F' THEN DBMS\_OUTPUT.PUT\_LINE('POOR');

ELSE DBMS\_OUTPUT.PUT\_LINE('NO SUCH GRADE');

END CASE;

END;

/

The above program can also be written as follows.

DECLARE

GRADE CHAR(1);

BEGIN

GRADE:='&GRADE';

CASE

WHEN **GRADE = 'A'** THEN DBMS\_OUTPUT.PUT\_LINE('EXCELLENT');

WHEN GRADE = 'B' THEN DBMS\_OUTPUT.PUT\_LINE('VERY GOOD');

WHEN GRADE = 'C' THEN DBMS\_OUTPUT.PUT\_LINE('GOOD');

WHEN GRADE = 'D' THEN DBMS\_OUTPUT.PUT\_LINE('FAIR');

WHEN GRADE = 'F' THEN DBMS\_OUTPUT.PUT\_LINE('POOR');

ELSE DBMS\_OUTPUT.PUT\_LINE('NO SUCH GRADE');

END CASE;

END;

/

**Looping Statements**

Loops are used to repeat a group of statements for number of times.

**For Loop**

This loop is used when the starting and ending values of the counter are known.

FOR <variable> IN [REVERSE] <LOWER\_BOUND>..<UPPER\_BOUND>

LOOP

STATEMENTS

END LOOP;

* The variable used as counter in the “FOR” loop should NOT be declared.
* Variable is automatically created and initialized with the lower bound value. If “REVERSE” is used then it is initialized with upper bound value.
* Checking the value of the variable if it is within the range is done automatically.
* After executing the statements of the block the variable is incremented by 1 only. It is decremented by 1 when the word “REVERSE” is used.
* When the variable value exceeds the range, the loop is terminated and the counter variable is removed from the memory automatically. Its value cannot be referred any more.

/\* USAGE OF FOR LOOP \*/

DECLARE

BEGIN

FOR X IN 1..10

LOOP

DBMS\_OUTPUT.PUT\_LINE(X);

END LOOP;

DBMS\_OUTPUT.PUT\_LINE('-------------------');

FOR X IN REVERSE 1..10

LOOP

DBMS\_OUTPUT.PUT\_LINE(X);

END LOOP;

END;

/

**While Loop**

This loop is used in a situation when the number of repetitions is unknown. We give a condition with “WHILE” and as long as the condition is TRUE the statements in loop will be executed. When the condition becomes FALSE the loop is terminated.

WHILE <Condition>

LOOP

. . . . . . . . .

. STATEMENTS. . .

. . . . . . . . .

END LOOP;

An example program is given below

/\* USAGE OF WHILE LOOP \*/

DECLARE

X NUMBER(5):=1;

BEGIN

WHILE X<=10

LOOP

DBMS\_OUTPUT.PUT\_LINE(X);

X:=X+1;

END LOOP;

DBMS\_OUTPUT.PUT\_LINE('-------------------');

X:=10;

WHILE X>=1

LOOP

DBMS\_OUTPUT.PUT\_LINE(X);

X:=X-1;

END LOOP;

END;

/

**Simple Loop**

This loop directly begins with the term LOOP. All the statements given with in the LOOP and END LOOP are executed. “Exit” statement is given within the loop that contains a condition. When this condition is TRUE the loop is terminated and if the condition is FALSE, then the statements inside the loop block are executed.

LOOP

. . . . . . . . .

. STATEMENTS. . .

EXIT WHEN <Condition>;

. . . . . . . . .

END LOOP;

An example program is given below.

/\* USAGE OF SIMPLE LOOP \*/

DECLARE

X NUMBER(5):=1;

BEGIN

LOOP

DBMS\_OUTPUT.PUT\_LINE(X);

X:=X+1;

EXIT WHEN X>10;

END LOOP;

DBMS\_OUTPUT.PUT\_LINE('-------------------');

X:=10;

LOOP

DBMS\_OUTPUT.PUT\_LINE(X);

X:=X-1;

EXIT WHEN X=0;

END LOOP;

END;

/

Following are some of the example programs using the control statements.

/\* PROGRAM TO ASK A NUMBER AND PRINT ITS MULTIPLICATION TABLE \*/

DECLARE

X NUMBER(6);

P NUMBER(15);

BEGIN

X:=&X;

DBMS\_OUTPUT.PUT\_LINE('MULITIPLICATION TABLE OF '||X);

FOR C IN 1..10

LOOP

P:=X\*C;

DBMS\_OUTPUT.PUT\_LINE(X||' \* '||C||' = '||P);

END LOOP;

END;

/

/\* PROGRAM TO CALCULATE SUM OF DIGITS IN A NUMBER \*/

DECLARE

N NUMBER(10);

R NUMBER(4);

S NUMBER(5):=0;

BEGIN

N:=&N;

LOOP

R:=MOD(N,10);

S:=S+R;

N:=FLOOR(N/10);

EXIT WHEN N=0;

END LOOP;

DBMS\_OUTPUT.PUT\_LINE('THE SUM OF DIGITST IS '||S);

END;

/

**EXCEPTIONS**

Exception is a runtime error. It occurs when the program is in execution. When this type of exception (Runtime Error) occurs, the program execution is terminated. All the variables that contained values till then will be lost from the memory. The error message is displayed when the program terminates. In PL/SQL exceptions are used to handle the runtime errors. Exceptions are of 2 types.

* System defined (these are pre-defined names in oracle)
* User defined (we create them as per our requirement in the program)

**SYSTEM DEFINED EXCEPTIONS**

Most of the runtime errors are already provided with exception names. These exceptions are automatically raised as per the situation in the program. We only have to handle the exception by using the exception name. Some of them are given below:

| **Oracle Exception Name** | **Oracle Error Code** | **When the exception is raised** |
| --- | --- | --- |
| DUP\_VAL\_ON\_INDEX | ORA-00001 | You tried to execute an INSERT or UPDATE statement that has created a duplicate value in a field restricted by a unique index. |
| TIMEOUT\_ON\_RESOURCE | ORA-00051 | You were waiting for a resource and you timed out. |
| TRANSACTION\_BACKED\_OUT | ORA-00061 | The remote portion of a transaction has rolled back. |
| INVALID\_CURSOR | ORA-01001 | You tried to reference a cursor that does not yet exist. This may have happened because you've executed a FETCH cursor or CLOSE cursor before OPENing the cursor. |
| NOT\_LOGGED\_ON | ORA-01012 | You tried to execute a call to Oracle before logging in. |
| LOGIN\_DENIED | ORA-01017 | You tried to log into Oracle with an invalid username/password combination. |
| NO\_DATA\_FOUND | ORA-01403 | You tried one of the following:   1. You executed a SELECT INTO statement and no rows were returned. 2. You referenced an uninitialized row in a table. |
| TOO\_MANY\_ROWS | ORA-01422 | You tried to execute a SELECT INTO statement and more than one row was returned. |
| ZERO\_DIVIDE | ORA-01476 | You tried to divide a number by zero. |
| INVALID\_NUMBER | ORA-01722 | You tried to execute a SQL statement that tried to convert a string to a number, but it was unsuccessful. |
| STORAGE\_ERROR | ORA-06500 | You ran out of memory or memory was corrupted. |
| PROGRAM\_ERROR | ORA-06501 | This is a generic "Contact Oracle support" message because an internal problem was encountered. |
| VALUE\_ERROR | ORA-06502 | You tried to perform an operation and there was a error on a conversion, truncation, or invalid constraining of numeric or character data. |
| CURSOR\_ALREADY\_OPEN | ORA-06511 | You tried to open a cursor that is already open. |

We use the “Exception” block of the PL/SQL program to handle the exception and provide the alternative statements to execute when the exception occurs.

/\* Comment line \*/

Declare

..........

Variable Declarations...

..........

Begin

..........

..........

.Actual program.

..........

..........

**[Exception]**

**When <ExceptionName> then**

**..........**

**..........**

**When <ExceptionName> then**

**..........**

**..........**

**When OTHERS then**

**..........**

**..........**

End;

**/**

There can be many exceptions defined in a single program. Exception should be the last block of a PL/SQL program. “OTHERS” is an added exception name to the above list which can be used in some cases when the error is something other than the handled exceptions.

/\* PROGRAM TO IMPLIMENT EXCEPTION - ZERO\_DIVIDE \*/

DECLARE

X NUMBER(4);

Y NUMBER(4);

Z NUMBER(5,1);

BEGIN

X:=&X;

Y:=&Y;

Z:=X/Y;

DBMS\_OUTPUT.PUT\_LINE('THE QUOTIENT IS '||Z);

EXCEPTION

WHEN ZERO\_DIVIDE THEN

DBMS\_OUTPUT.PUT\_LINE('RESULT IS INFINITY');

END;

/

/\* PROGRAM TO SHOW NO\_DATA\_FOUND EXCEPTION \*/

DECLARE

ENO NUMBER(5);

ENAME VARCHAR2(10);

SAL NUMBER(5);

DEPT\_ID NUMBER(2);

BEGIN

ENO:=&EMPLOYEE\_NUMBER;

SELECT FIRST\_NAME, SALARY, DEPARTMENT\_ID INTO ENAME,SAL, DEPT\_ID FROM EMPLOYEES WHERE EMPLOYEE\_ID=ENO;

DBMS\_OUTPUT.PUT\_LINE('EMPLOYEE NAME '||ENAME);

DBMS\_OUTPUT.PUT\_LINE('EMPLOYEE SALARY '||SAL);

DBMS\_OUTPUT.PUT\_LINE('DEPARTMENT NO. '||DEPT\_ID);

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

DBMS\_OUTPUT.PUT\_LINE('EMPLOYEE\_ID NOT FOUND');

END;

/

/\* NO\_DATA\_FOUND AND TOO\_MANY\_ROWS EXCEPTION \*/

DECLARE

ENO NUMBER(5);

ENAME VARCHAR2(10);

SAL NUMBER(5);

DEPT\_ID NUMBER(3);

BEGIN

ENAME:=UPPER('&EMPLOYEE\_NAME');

SELECT EMPLOYEE\_ID, SALARY, DEPARTMENT\_ID INTO ENO, SAL, DEPT\_ID

FROM EMPLOYEES WHERE UPPER(FIRST\_NAME)=ENAME;

DBMS\_OUTPUT.PUT\_LINE('EMPLOYEE ID '||ENO);

DBMS\_OUTPUT.PUT\_LINE('EMPLOYEE SALARY '||SAL);

DBMS\_OUTPUT.PUT\_LINE('DEPARTMENT NO. '||DEPT\_ID);

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

DBMS\_OUTPUT.PUT\_LINE('EMPLOYEE NAME NOT FOUND');

WHEN TOO\_MANY\_ROWS THEN

DBMS\_OUTPUT.PUT\_LINE('THERE ARE MANY EMPLOYEES WITH SAME NAME');

END;

/

**USER DEFINED EXCEPTION**

The user defined exceptions are those which are created by the user as per the requirement. In order to create a user defined exception we have to

* Declare an exception variable (using the data type “EXCEPTION”)
* Raise the exception as per requirement of the logic (Raise <ExceptionName>)
* Provide alternative statements in the exception block

A user defined exception program will have the following syntax.

/\* Comment line \*/

Declare

..........

Variable Declarations...

**<ExceptionName> Exception;**

..........

Begin

..........

..........

**Raise <ExceptionName>;**

..........

..........

**Exception**

**When <ExceptionName> then**

**..........**

**..........**

**When <ExceptionName> then**

**..........**

**..........**

**When OTHERS then**

**..........**

**..........**

End;

**/**

Consider the following example program.

/\* EXAMPLE ON USER DEFINED EXCEPTION \*/

DECLARE

INVALID\_READINGS EXCEPTION;

R1 NUMBER(6);

R2 NUMBER(6);

UNITS NUMBER(6);

BILL NUMBER(8,2);

BEGIN

R1:=&OLD\_READING;

R2:=&NEW\_READING;

IF R1>R2 THEN

RAISE INVALID\_READINGS;

END IF;

UNITS:=R2-R1;

BILL:=UNITS\*2.5;

DBMS\_OUTPUT.PUT\_LINE('UNITS USED IS '||UNITS);

DBMS\_OUTPUT.PUT\_LINE('BILL AMOUNT IS '||BILL);

EXCEPTION

WHEN INVALID\_READINGS THEN

DBMS\_OUTPUT.PUT\_LINE('OLD READING CANNOT BE MORE THAN THE NEW READING');

END;

/

**CURSORS**

When a SQL statement is executed, a temporary work area is created in the system memory called as CURSOR. A cursor contains information of the rows accessed from a table when a select, insert, update or delete statement is executed. This temporary work area is used to store the data retrieved from the database, and manipulate this data. A cursor can hold more than one row of data, but can process only one row at a time.

Cursors are of two types:

* Explicit cursor (we have to create them)
* Implicit cursor (these are created by the system)

**Explicit Cursors**

These cursors are created by the user to manipulate the data of a table through a PL/SQL program. To process the set of rows that are held by a cursor one by one in a program we have to:

* Declare the cursor
* Open the cursor
* Fetch one record into temporary variables
* Check for *End of Cursor*
* If it is NOT end of cursor then process the data and fetch next record
* If it is end of cursor then close the cursor

Every cursor will have 4 attributes. These attributes are used along with the cursor name.

These attributes are used as **<CursorName>%<Attribute>**

%FOUND to check if *Fetch* statement is successful or not

%NOTFOUND to check if *Fetch* statement is NOT successful or not

%ISOPEN to check if the cursor is already open or not

%ROWCOUNT to get the number of rows fetched from the cursor

Declaring the cursor CURSOR <CursorName> IS/AS <Select Statement>;

Opening the cursor OPEN <CursorName>;

Fetching record from cursor FETCH <CursorName> INTO <TemporaryVariables>;

Checking for *End of Cursor* <CursorName>%NOTFOUND; **(OR)** <CursorName>%FOUND;

Closing the cursor CLOSE <CursorName>;

/\* PROGRAM USING SIMPLE CURSOR (**COLUMN TYPE VARIABLES**) \*/

DECLARE

CURSOR C1 IS SELECT EMPLOYEE\_ID, FIRST\_NAME, SALARY, DEPARTMENT\_ID FROM EMPLOYEES WHERE EMPLOYEE\_ID<=120;

EMPID EMPLOYEES.EMPLOYEE\_ID%TYPE;

ENAME EMPLOYEES.FIRST\_NAME%TYPE;

SAL EMPLOYEES.SALARY%TYPE;

DEPT\_ID EMPLOYEES.DEPARTMENT\_ID%TYPE;

BEGIN

OPEN C1;

LOOP

FETCH C1 INTO EMPID, ENAME, SAL, DEPT\_ID;

EXIT WHEN C1%NOTFOUND;

DBMS\_OUTPUT.PUT\_LINE('EMPLOYEE NUMBER '||EMPID);

DBMS\_OUTPUT.PUT\_LINE('EMPLOYEE NAME '||ENAME);

DBMS\_OUTPUT.PUT\_LINE('EMPLOYEE SALARY '||SAL);

DBMS\_OUTPUT.PUT\_LINE('DEPT NUMBER '||DEPT\_ID);

DBMS\_OUTPUT.PUT\_LINE('------------------------');

END LOOP;

DBMS\_OUTPUT.PUT\_LINE('THE NUMBER OF RECORDS ARE '||C1%ROWCOUNT);

CLOSE C1;

END;

/

/\* ABOVE PROGRAM USING SIMPLE CURSOR (**ROWTYPE VARS**) \*/

DECLARE

CURSOR C1 IS SELECT \* FROM EMPLOYEES WHERE EMPLOYEE\_ID<=120;

X EMPLOYEES%ROWTYPE;

BONUS EMPLOYEES.SALARY%TYPE;

TOTAL EMPLOYEES.SALARY%TYPE:=0;

BEGIN

OPEN C1;

LOOP

FETCH C1 INTO X;

EXIT WHEN C1%NOTFOUND;

IF X.JOB\_ID='IT\_PROG' THEN

BONUS:=500;

ELSIF X.JOB\_ID='PU\_CLERK' THEN

BONUS:=400;

ELSE

BONUS:=100;

END IF;

TOTAL:=TOTAL+BONUS;

DBMS\_OUTPUT.PUT\_LINE('EMPLOYEE NUMBER '||X.EMPLOYEE\_ID);

DBMS\_OUTPUT.PUT\_LINE('EMPLOYEE NAME '||X.FIRST\_NAME);

DBMS\_OUTPUT.PUT\_LINE('EMPLOYEE SALARY '||X.SALARY);

DBMS\_OUTPUT.PUT\_LINE('DEPT NUMBER '||X.DEPARTMENT\_ID);

DBMS\_OUTPUT.PUT\_LINE('BONUS AMOUNT'||BONUS);

DBMS\_OUTPUT.PUT\_LINE('------------------------');

UPDATE EMPLOYEES SET SALARY=SALARY+BONUS WHERE

EMPLOYEE\_ID=X.EMPLOYEE\_ID;

END LOOP;

DBMS\_OUTPUT.PUT\_LINE('TOTAL OF BONUS AMOUNT'||TOTAL);

DBMS\_OUTPUT.PUT\_LINE(C1%ROWCOUNT||' ROWS PROCESSED');

DBMS\_OUTPUT.PUT\_LINE('------------------------');

CLOSE C1;

END;

/

**Using PARAMETERIZED cursor:**

Generally a cursor is created in the declare section itself, but, in a prameterized cursor it is created when we pass the required values as arguments to the cursor while opening the cursor. This allows us to recall the same cursor many times just by passing the required value as input. Note that the parameter does not have the width. We give only the datatype.

/\* program to display employee details of given department\_id

using PARAMETERIZED CURSOR \*/

declare

dno employees.department\_id%type;

cursor c1(d number) is select employee\_id, first\_name, salary from employees

where department\_id=d;

x c1%rowtype;

begin

dno:=&department\_id;

open c1(dno); /\* now the cursor is created \*/

loop

fetch c1 into x;

exit when c1%notfound;

Dbms\_output.put\_line('emp id number '||x.employee\_id);

Dbms\_output.put\_line('emp name is '||x.first\_name);

Dbms\_output.put\_line('emp salary is '||x.salary);

Dbms\_output.put\_line('---------------------------');

end loop;

Dbms\_output.put\_line('number of rows listed '||c1%rowcount);

close c1;

end;

/

**Using FOR Loop with cursor programs:**

By using FOR LOOP with cursor programs we get the following advantages.

1. We do not need to declare the ROWTYPE variable because this variable is automatically created. (it is automatically assumed as a CursorRowtype variable)
2. FOR Loop variable is automatically created and initialized with first row of the cursor. Cursor is automatically opened. (We don’t give OPEN <CursorName> statement and FETCH statement)
3. After the loop statements are completed, with END LOOP it automatically moves to the next record and checks for END of Cursor. (we don’t need to check for C1%NOTFOUND)
4. When the last row of the cursor is processed then the cursor is closed automatically.

/\* PROGRAM USING **CURSOR WITH FOR LOOP** \*/

DECLARE

CURSOR C1 IS SELECT \* FROM EMPLOYEES WHERE EMPLOYEE\_ID>=175 AND EMPLOYEE\_ID<=180;

BEGIN

FOR X IN C1 /\* X IS ASSUMED AS ROWTYPE VAR OF C1 CURSOR \*/

LOOP

DBMS\_OUTPUT.PUT\_LINE('EMPLOYEE NUMBER '||X.EMPLOYEE\_ID);

DBMS\_OUTPUT.PUT\_LINE('EMPLOYEE NAME '||X.FIRST\_NAME);

DBMS\_OUTPUT.PUT\_LINE('EMPLOYEE SALARY '||X.SALARY);

DBMS\_OUTPUT.PUT\_LINE('DEPT NUMBER '||X.DEPARTMENT\_ID);

DBMS\_OUTPUT.PUT\_LINE('------------------------');

END LOOP;

END;

/

**Implicit Cursors**

These cursors are created by the system automatically when user gives Insert, Update, Delete or Select statement. User can create any name for an explicit cursor but the name of the implicit cursor is “SQL”. All the cursor attributes can be now used with this cursor name (SQL%FOUND, SQL%NOTFOUND, SQL%ISOPEN, SQL%ROWCOUNT).

/\* EXAMPLE ON IMPLICIT CURSORS \*/

DECLARE

EMPNO EMPLOYEES.EMPLOYEE\_ID%TYPE;

BEGIN

DELETE FROM EMPLOYEES WHERE EMPLOYEE\_ID=&EMPNO;

IF SQL%NOTFOUND THEN

DBMS\_OUTPUT.PUT\_LINE('NO EMPLOYEES FOUND');

ELSE

DBMS\_OUTPUT.PUT\_LINE(SQL%ROWCOUNT||' EMPLOYEE RECORD(S) DELETED');

END IF;

END;

/

**SUB-PROGRAMS**

A program used within another program is called sub program. Following are the sub programs.

* Procedures
* Functions
* Packages

These are called as sub programs because these programs are used from another program. All sub programs are database objects created at the database server. (CREATE command is used to create them)

**PROCEDUREs/STORED PROCEDUREs**

A procedure is database object which represents a group of statements. Whenever it is called with the required arguments it will execute the statements that it represents. A procedure can be called any number of times and each time with a different argument. A procedure is created with the following syntax:

CREATE OR REPLACE PROCEDURE <ProcedureName>**[**(Parameters)**]** IS/AS

. varaibles .

BEGIN

. . . . . . . .

Statements

. . . . . . . .

END;

/

The variables used as parameters should not be declared with the width, only the data type is given. Once a procedure is created it can be called from SQL Prompt or from any PL/SQL program.

To call a procedure from SQL prompt we use EXECUTE/EXEC command.

SQL> EXECUTE <ProcedureName>(ParameterValues)

(OR)

SQL> EXEC <ProcedureName>(ParameterValues)

To call a procedure from PL/SQL program EXECUTE / EXEC is NOT used, it is directly called.

DECLARE

. . . . . . . .

BEGIN

. . . . . . . .

<ProcedureName>(ParameterValues);

. . . . . . . .

END;

/

/\* PROGRAM TO CREATE A PROCEDURE - PRINT \*/

CREATE OR REPLACE PROCEDURE PRINT(X VARCHAR2) IS

BEGIN

DBMS\_OUTPUT.PUT\_LINE('------------------------');

DBMS\_OUTPUT.PUT\_LINE(X);

DBMS\_OUTPUT.PUT\_LINE('------------------------');

END;

/

/\* USING THE PROCEDURE (PRINT) FROM A PL/SQL PROGRAM \*/

DECLARE

X NUMBER(4);

Y NUMBER(4);

Z NUMBER(5);

BEGIN

X:=&X;

Y:=&Y;

Z:=X+Y;

DBMS\_OUTPUT.PUT\_LINE('FIRST VALUE IS '||X);

DBMS\_OUTPUT.PUT\_LINE('SECOND VALUE IS '||Y);

PRINT('THE SUM OF THE VALUES IS '||Z);

END;

/

/\* PROCEDURE TO SHOW EMPLOYEES DATA OF GIVEN DEPARTMENT NUMBER \*/

CREATE OR REPLACE PROCEDURE SHOW\_EMP\_DATA(DNO NUMBER) IS

CURSOR C1 IS SELECT \* FROM EMPLOYEES WHERE DEPARTMENT\_ID = DNO;

N NUMBER(4);

BEGIN

FOR X IN C1 /\* X IS ASSUMED AS ROWTYPE VAR \*/

LOOP

DBMS\_OUTPUT.PUT\_LINE('EMPLOYEE NUMBER '||X.EMPLOYEE\_ID);

DBMS\_OUTPUT.PUT\_LINE('EMPLOYEE NAME '||X.FIRST\_NAME);

DBMS\_OUTPUT.PUT\_LINE('EMPLOYEE SALARY '||X.SALARY);

DBMS\_OUTPUT.PUT\_LINE('DEPT NUMBER '||X.DEPARTMENT\_ID);

DBMS\_OUTPUT.PUT\_LINE('------------------------');

N:=C1%ROWCOUNT;

END LOOP;

PRINT('NUMBER OF RECORDS '||NVL(N,0));

END;

/

/\* PROGRAM USING THE PROCEDURE SHOW\_EMP\_DATA \*/

DECLARE

DNO EMPLOYEES.DEPARTMENT\_ID%TYPE;

BEGIN

DNO:=&DEPARTMENT\_NUMBER;

SHOW\_EMP\_DATA(DNO);

END;

/

/\* PROCEDURE TO ADD A RECORD INTO A TABLE “FRIENDS” BY TAKING REQUIRED ARGUMENTS \*/

CREATE OR REPLACE PROCEDURE ADD\_FRIEND\_DATA(SN NUMBER, NA VARCHAR2, DB DATE, PH NUMBER) IS

BEGIN

INSERT INTO FRIENDS VALUES(SN,INITCAP(NA),DB,PH);

PRINT('ONE ROW IS INSERTED INTO FRIENDS TABLE');

END;

/

/\* PROCEDURE TO DELETE A RECORD FROM A TABLE “FRIENDS” BY TAKING REQUIRED ARGUMENTS \*/

CREATE OR REPLACE PROCEDURE DEL\_FRIEND\_DATA(SN NUMBER) IS

N NUMBER(4);

BEGIN

SELECT NVL(COUNT(SNO),0) INTO N FROM FRIENDS WHERE SNO=SN;

IF N=0 THEN

PRINT('SORRY THIS RECORD DOES NOT EXITS');

ELSE

DELETE FROM FRIENDS WHERE SNO=SN;

PRINT(N||' ROW(S) DELETED');

END IF;

END;

/

**FUNCTIONS**

Oracle provides a feature of creating a function of our own. Unlike built-in functions these are programs designed by the user which calculates and returns a value. A function is a database object which represents a group of statements. Whenever a function is called with the required arguments it will execute the statements that it represents and return a value. This returned value can be used in the program at various places. A function can be called any number of times and each time with a different argument. A function is created with the following syntax:

CREATE OR REPLACE FUNCTION <FunctionName>(Parameters) RETURN <ReturnDatatype> IS/AS

. . . . . . .

. varaibles .

. . . . . . .

BEGIN

. . . . . . . .

Statements

. . . . . . . .

**RETURN <Value>; (this statement is compulsory)**

END;

/

The variables used as parameters should not be declared with the width, only the data type is given. Once a function is created it can be called from SQL Prompt or from any PL/SQL program.

To call a function from SQL prompt we use the normal select statement.

SQL> SELECT <FunctionName>(Parameters) FROM DUAL;

The user defined function must be refered only from DUAL table.

To call a function from PL/SQL program we have 2 ways.

DECLARE

. . . . . . . .

BEGIN

. . . . . . . .

<Variable>:=<FunctionName>(ParameterValues);

(OR)

SELECT <FunctionName>(Parameters) INTO <Variable> FROM DUAL;

. . . . . . . .

END;

/

Example 1

/\* CREATING A FUNCTION GET\_CUBE TO CALCULATE CUBE OF A NUMBER \*/

CREATE OR REPLACE FUNCTION GET\_CUBE(N NUMBER) RETURN NUMBER IS

RES NUMBER(20);

BEGIN

RES:=N\*N\*N;

RETURN RES;

END;

/

The above function can be called from SQL Prompt or from PL/SQL program

* Calling from SQL Prompt:

SQL> Select GET\_CUBE(5) From Dual;

Output: 125

* The above created function can be used in another program as follows.

/\* USING THE FUNCTION GET\_CUBE IN A PL/SQL PROGRAM \*/

DECLARE

X NUMBER(5);

RES NUMBER(20);

BEGIN

X:=&NUMBER;

RES:=GET\_CUBE(X);

**(OR)**

SELECT GET\_CUBE(X) INTO RES FROM DUAL;

DBMS\_OUTPUT.PUT\_LINE('THE CUBE IS '||RES);

END;

/

Example 2

/\* CREATING A FUNCTION SUM\_DIGITS TO CALCULATE SUM OF DIGITS IN A NUMBER \*/

CREATE OR REPLACE FUNCTION SUM\_DIGITS(N NUMBER) RETURN NUMBER IS

X NUMBER(15):=N;

R NUMBER(2);

S NUMBER(6);

BEGIN

S:=0;

LOOP

R:=MOD(X,10);

S:=S+R;

X:=FLOOR(X/10);

EXIT WHEN X=0;

END LOOP;

RETURN S;

END;

/

* Calling from SQL Prompt:

SQL> Select SUM\_DIGITS(53621) From Dual;

Output: 17

* The following program is to call a function created above.

/\* USING THE FUNCTION SUM\_DIGITS IN A PL/SQL PROGRAM \*/

DECLARE

X NUMBER(20);

RES NUMBER(5);

BEGIN

X:=&NUMBER;

RES:=SUM\_DIGITS(X);

**(OR)**

SELECT SUM\_DIGITS(X) INTO RES FROM DUAL;

DBMS\_OUTPUT.PUT\_LINE('SUM OF DIGITS IS '||RES);

END;

/

Example 3

/\* CREATING A FUNCTION “GET\_EMP\_COMMISSION” TO CALCULATE SUM OF SALARIES OF THE GIVEN DEPARTMENT ID \*/

CREATE OR REPLACE FUNCTION GET\_EMP\_COMMISSION(DNO NUMBER) RETURN NUMBER IS

CURSOR C1 IS SELECT SALARY,COMMISSION\_PCT FROM EMPLOYEES WHERE DEPARTMENT\_ID=DNO;

TOTAL NUMBER(10,2):=0;

BEGIN

FOR X IN C1

LOOP

TOTAL:=TOTAL+(X.SALARY\*NVL(X.COMMISSION\_PCT,0));

END LOOP;

RETURN TOTAL;

END;

/

The above function can be called from SQL Prompt or from PL/SQL program

* From SQL Prompt:

SQL> Select GET\_EMP\_COMMISSION(80) From Dual;

Note that even though the function refers the data from EMPLOYEES table, we use DUAL.

**PACKAGES**

PL/SQL packages are Database objects that groups logically related PL/SQL types, variables and subprograms. In simple words a package is a container object which binds types, procedures, functions, variables together under one name. Package does not contain any code of its own.

A package is created in two steps:

* Package specification
* Package body or definition

**Package Specification**

The specification is the interface to the package. It contains just DECLARATIONS of the types, variables, constants, exceptions, cursors, procedures and functions that can be referenced from outside the package. In other words, it contains all information about the content of a package, but excludes the code for the subprograms.

Following is the syntax for creating package specification.

CREATE OR REPLACE PACKAGE <PackageName> IS

PROCEDURE <ProcedureName1>(Parameters);

PROCEDURE <ProcedureName2>(Parameters);

. . . . .

. . . . .

FUNCTION <FunctionName1>(Parameters) RETURN <DataType>;

FUNCTION <FunctionName2>(Parameters) RETURN <DataType>;

. . . . .

. . . . .

END;

/

**Package Body**

The package body has the descriptive code for various methods declared in the package specification and other private declarations, which are hidden from code outside the package.

The CREATE PACKAGE BODY Statement is used for creating the package body.

CREATE OR REPLACE **PACKAGE BODY** <PackageName> IS

PROCEDURE <ProcedureName1>(Parameters) IS

Variables

BEGIN

. . . . .

Statements

. . . . .

END;

PROCEDURE <ProcedureName2>(Parameters) IS

Variables

BEGIN

. . . . .

Statements

. . . . .

END;

. . . . .

. (description of all procedures). .

. . . . .

FUNCTION <FunctionName1>(Parameters) RETURN <DataType> IS

Variables

BEGIN

. . . . .

Statements

RETURN <Value>;

END;

FUNCTION <FunctionName2>(Parameters) RETURN <DataType> IS

Variables

BEGIN

. . . . .

Statements

RETURN <Value>;

END;

. . . . .

. (description of all functions). .

. . . . .

END;

/

Once the package body is created the reference of any of its elements is given as

<PackageName> . <Procedure>(Parameters)

(OR)

<PackageName> . <FunctionName>(Parameters)

They may be called from SQL prompt or from another PL/SQL program.

/\* PACKAGE SPECIFICATION APEC \*/

CREATE OR REPLACE PACKAGE APEC IS

PROCEDURE EVEN\_ODD(N NUMBER);

PROCEDURE SHOW\_TABLE(N NUMBER);

FUNCTION REVERSE\_NUMBER(N NUMBER) RETURN NUMBER;

FUNCTION AVERAGE(X NUMBER, Y NUMBER) RETURN NUMBER;

END;

/

/\* PACKAGE BODY OF APEC \*/

CREATE OR REPLACE PACKAGE BODY APEC IS

PROCEDURE EVEN\_ODD(N NUMBER) IS

R NUMBER(1);

BEGIN

R:=MOD(N,2);

IF R=0 THEN

DBMS\_OUTPUT.PUT\_LINE('EVEN NUMBER');

ELSE

DBMS\_OUTPUT.PUT\_LINE('ODD NUMBER');

END IF;

END;

PROCEDURE SHOW\_TABLE(N NUMBER) IS

P NUMBER(10);

BEGIN

FOR C IN 1..10

LOOP

P:=N\*C;

DBMS\_OUTPUT.PUT\_LINE(N||' \* '||C||' = '||P);

END LOOP;

END;

FUNCTION REVERSE\_NUMBER(N NUMBER) RETURN NUMBER IS

X NUMBER(20):=N;

REV NUMBER(20):=0;

R NUMBER(1);

BEGIN

LOOP

R:=MOD(X,10);

REV:=REV\*10+R;

X:=FLOOR(X/10);

EXIT WHEN X=0;

END LOOP;

RETURN REV;

END;

FUNCTION AVERAGE(X NUMBER, Y NUMBER) RETURN NUMBER IS

RES NUMBER(9,2);

BEGIN

RES:=(X+Y)/2;

RETURN RES;

END;

END;

/

**TRIGGERS**

Database Triggers is one of the good features of Oracle Database which provides automated functionalities on tables. These are programs created at the database as objects and are stored permanently. These are similar to stored procedures. A trigger can contain SQL or PL/SQL statements. Triggers cannot be called explicitly. They fire (execute) automatically whenever a said DML operation occurs on a given table. A general syntax of creating a trigger is given as follows.

CREATE OR REPLACE TRIGGER <TriggerName> BEFORE/AFTER <DML Operations> ON <TableName> [FOR EACH ROW]

[DECLARE]

..Variables..

BEGIN

. . . . . . . .

Statements

. . . . . . . .

END;

/

Triggers can be created to execute the body “Before” or “After” the given DML operation is done. The optional clause “FOR EACH ROW” indicates a trigger body to execute once for every row that was affected for the given DML operation. Such trigger is a *Row Level* *trigger*. If the “FOR EACH ROW” clause is not given then the trigger body will execute only once when the trigger fires. Such trigger is called *Table Level Trigger*.

Information about list of triggers can be obtained from USER\_OBJECTS

SQL> SELECT \* FROM USER\_OBJECTS WHERE OBJECT\_TYPE=’TRIGGER’;

If the trigger is not needed any more then it can be dropped.

SQL> DROP TRIGGER <TriggerName>;

Example1: Following is an example that shows how to restrict all users other than HR user from performing any DML operations on the employees table even though privileges are given.

/\* TRIGGER EXAMPLE \*/

CREATE OR REPLACE TRIGGER STOP\_TRANSACTION BEFORE INSERT OR UPDATE OR DELETE ON EMPLOYEES

DECLARE

UNAME VARCHAR2(30);

BEGIN

SELECT UPPER(USER) INTO UNAME FROM DUAL;

IF UNAME!='HR' THEN

RAISE\_APPLICATION\_ERROR(-20001,'SORRY TABLE IS LOCKED .....

PLEASE CONTACT HR');

END IF;

END;

/

In the above program “Raise\_Application\_Error” is a pre-defined procedure (stored procedure) which replaces generic Oracle exception messages with our own, more meaningful messages. The user defined error codes are allowed within a range of -20000 to -20999.

After the trigger is created try executing any of the DML operations on the employees table from another user login. You get an error message as said in the trigger body. Try the same from HR user, you do not get any error.

**OLD and NEW Pseudo records**

When a row-level trigger fires, the PL/SQL run-time system creates and populates the two pseudo records **:OLD** and **:NEW**. They are called pseudo records because they have some, but not all, of the properties of records.

:OLD.<ColName> applies for update and delete commands

:NEW.<ColName> applies for insert and update commands

Example2: Consider the following table “FRIENDS”

| Sno | Name | Dob | Phone |
| --- | --- | --- | --- |
| 1 | David | 12-Jun-85 | 9347473633 |
| 2 | John | 16-Apr-84 | 48476254 |

Now a trigger is to be created which records the current phone no and the changed phone no. along with the user who is updating when the user gives an update command.

Create another table with name FRIENDS\_BACKUP with following fields.

| Sno | Name | Dob | Prev\_Phone | New\_Phone | Uname |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

Now type the following trigger program

/\* TRIGGER EXAMPLE TO STORE THE MODIFIED DATA ALONG WITH OLD DATA \*/

CREATE OR REPLACE TRIGGER TRACK\_PHNO\_FRIENDS AFTER UPDATE ON FRIENDS FOR EACH ROW

BEGIN

INSERT INTO FRIENDS\_BACKUP VALUES

(:OLD.SNO,:OLD.NAME,:OLD.DOB,:OLD.PHONE,:NEW.PHONE, USER);

END;

/

After the trigger is created, try changing the phone no. of any of the records. Then check back to the data in FRIENDS\_BACKUP table. (SELECT \* FROM FRIENDS\_BACKUP;)

Example 3: Assume table “EMPLOYEES” has been granted to a number of users. Many of those who have privileges on the table are performing transactions (Insert/Update/Delete). We now would like to make a note of all the transactions made by the users in another table called “EMP\_AUDIT”.

* Create a table EMP\_AUDIT with following columns

ColName Datatype

-----------------------------

UNAME VARCHAR2(15)

TRANS\_DATE VARCHAR2(30)

TRANS VARCHAR2(15)

RO\_ID VARCHAR2(35)

* Now type the following program and execute it

CREATE OR REPLACE TRIGGER RECORD\_TRANS AFTER INSERT OR UPDATE OF

SALARY, DEPARTMENT\_ID OR DELETE ON EMPLOYEES FOR EACH ROW

DECLARE

X EMP\_AUDIT%ROWTYPE;

BEGIN

SELECT USER, TO\_CHAR (SYSDATE,'DY, DD-MON-YYYY, HH24:MI:SS') INTO

X.UNAME, X.TRANS\_DATE FROM DUAL;

CASE

WHEN INSERTING THEN

X.TRANS:='INSERT';

WHEN UPDATING('SALARY') THEN

X.TRANS:='SAL\_UPDATE';

WHEN UPDATING('DEPARTMENT\_ID') THEN

X.TRANS:='DEPTID\_UPDATE';

WHEN DELETING THEN

X.TRANS:='DELETE';

END CASE;

SELECT :OLD.ROWID INTO X.RO\_ID FROM DUAL;

INSERT INTO EMP\_AUDIT VALUES(X.UNAME,X.TRANS\_DATE,X.TRANS,X.RO\_ID);

END;

/

After the trigger is created (SQL> @filename), perform some DML operations from any of the users. All the transactions are now recorded in the table “EMP\_AUDIT”

SQL> SELECT \* FROM EMP\_AUDIT;

/\* PROGRAM TO INITIALIZE AN ARRAY \*/

DECLARE

TYPE TEST\_TAB IS TABLE OF NUMBER;

VAR TEST\_TAB:=TEST\_TAB(450,620,430,340,560);

BEGIN

FOR C IN 1..VAR.COUNT

LOOP

DBMS\_OUTPUT.PUT\_LINE(VAR(C));

END LOOP;

END;

/

/\* PROGRAM TO INITIALIZE AN ARRAY \*/

DECLARE

TYPE TEST\_TAB IS TABLE OF NUMBER;

VAR TEST\_TAB:=TEST\_TAB(450,620,430,340,560);

BEGIN

FOR C IN 1..VAR.COUNT

LOOP

DBMS\_OUTPUT.PUT\_LINE(VAR(C));

END LOOP;

END;

/