Rs: 120/-

S.T.Y

ORACLE 11g

COURSE MATERIAL

NARESH i TM
technologies
SOFTWARE TRAINING & DEVELOPMENT

An ISO 9001 : 2000 Certified Company

Opp. Satyam Theatre, Ameerpet, Hyderabad - 500 016.
Ph : 23746666, 23734842

First Steps Towards Oracle 10g

Let us Make Database Practical

What is Data Actually?

Data:

It is Stored Representation of OBJECTS and EVENTS That Have Meaning and Importance in the User's Environment. Data Can be Structure OR Unstructured.

- Structured → Student Name, Address..
- Unstructured → StudPhoto, AddrMap..

Representation of Data :

King	10	5500	10-02-06
Blake	10	4500	02-03-05
Clark	20	3500	15-05-04
Smith	30	1800	06-10-00
Miller	20	1100	22-12-03
Taylor	30	3500	24-09-98

What is Information Actually?

Information:

It is Data that is in Processed Form, Such That It Increases the Knowledge of the Person Who Uses the Data.

Representation of Information:

Employees Information

Organization : Naresh *i* Technologies.

Date : 29 October 2010.

Place : Hyderabad.

Name	Department	Salary	DOJ
King	10	5500	10-02-06
Blake	10	4500	02-03-05
Clark	20	3500	15-05-04
Smith	30	1800	06-10-00
Miller	20	1100	22-12-03
Taylor	30	3500	24-09-98

What is Metadata Actually?

Metadata:

It is the Data Which Describes the Properties OR Characteristics of End Users Data and the Context of the Data.

Metadata Properties Can Include Information Such as:

- Data Name.
- Definitions.
- Length OR Size.
- Values Allowed.
- Source of Data.
- Ownership.

Metadata and Data are always separate.

Metadata Enables The Database Designers and Programmers to Understand Exactly in Which Form The Data Should Exist Within The System.

Metadata for Employees Information

Data Item Specification			Value	
Field Name	Type	Length	Min	Max
Empname	Alpha Numeric	30	10	90
DeptNo	Integer	2	1500	50000
EmpSal	Decimal	7		
HireDate	Date			

Database Management Systems

Database Management Systems is Software that is Used to Create, Maintain, and Provide Controlled Access to User Databases.

Database Management Systems Should Provide Systematic Method of

- Creating the Database.
- Updating the Database.
- Storing the Database.
- Retrieving of Data from Database.

Expected Features of DBMS Software

- Enable End Users and Application Programmers to Share the Data.
- Enable Data Shared Among Multiple Applications.
- Should not Propagate and Store Data in New Files for Every New Application.
- Should Provide Facility for…
  - Controlling Data Access.
  - Enforce Data Integrity.
  - Manage Concurrency Control.
  - Restoring the Data in System Failures.

Database Management Systems Evolution

- First Time Introduced During 1960's.
- Relational Model First Defined By E.F. Codd (IBM) in 1970.

Objectives Behind Evolution…

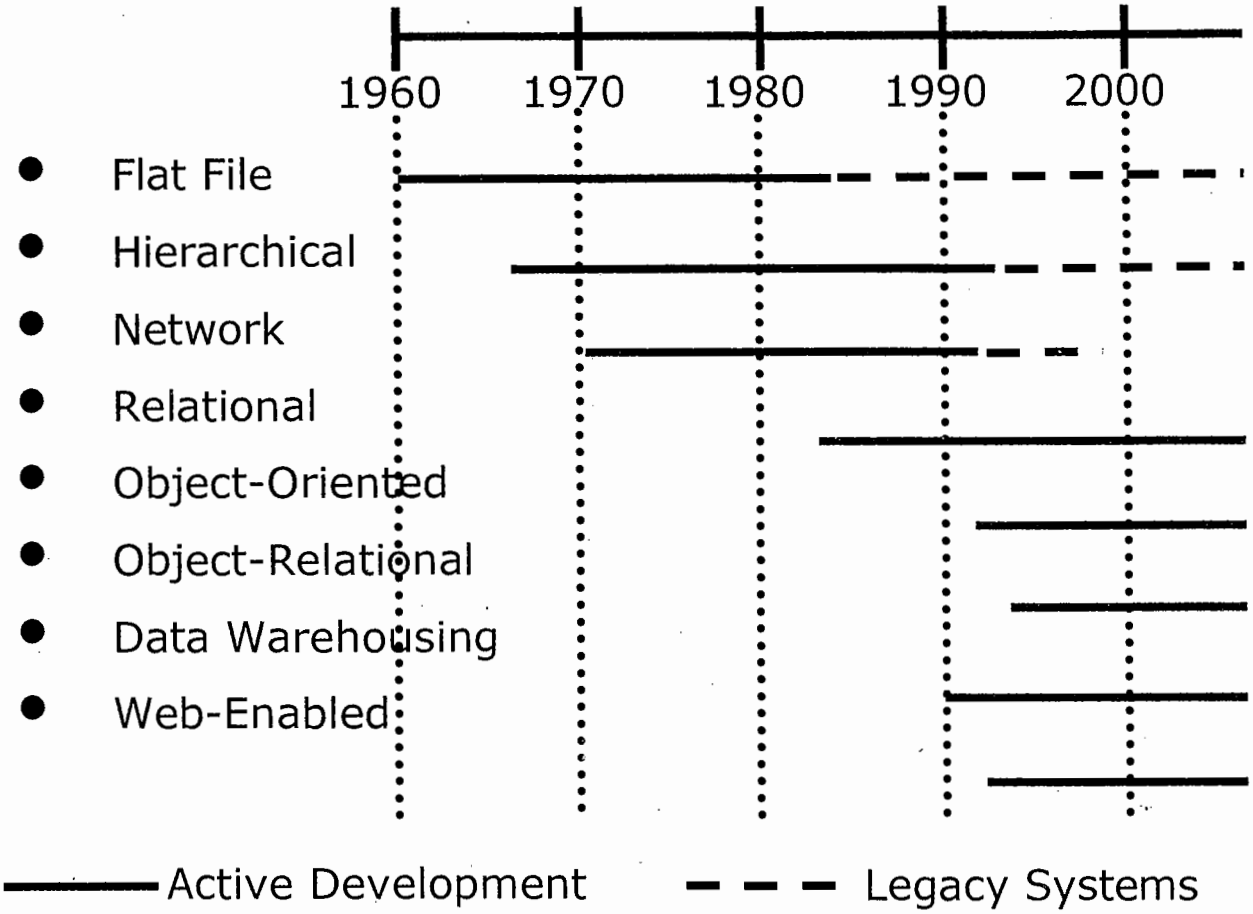
- Provide Greater Independence Between Programs and Data.
- Reduce The Maintenance Cost of Software.
- Manage Complex Data Types and Structures.
- Provide Easier and Faster Access to Data Even for Novice Users.
- Provide Future Sensation for Powerful Platforms for Decision Support Systems and Expert Systems.

Different Database Models

- Flat File DBMS.
- Hierarchical DBMS.
- Network DBMS.

- Relational DBMS.
- Object-Oriented DBMS.
- Object-Relational DBMS.
- Data Warehousing.
- Web-Enabled DBMS.

Database Models and Their Timelines



What is Meant By Database Application

- It is an Application Program OR Set of Related Programs That is Used to Perform a Series of Activities on Behalf of Database Users.
- The Database Application Activities Include ...
  - Create → Add New Data to Database.
  - Read → Read Current Database data.
  - Update → Update OR Modify Current Database Data.
  - Delete → Delete Current Data From The Database.

Database as Per Oracle

- As Per Oracle Database is a Collection of Data in One OR More Number of Files.
- The Database is a Collection of Logical Structures And Physical Structures Which are Integrated and Configured FOR The Integrity of The System.

Logical Structures :

- In Design State The System is Represented in the Form of Entity Relationship Model, OR DFD's OR UML Diagrams.
- In the Database State it is Representation of the Actual Metadata of the Database Management System Software Which Will be in the form of Tablespaces, Data Dictionary Objects Etc.

Physical Structures :

- In This the System is Represented in the Form of Tables, Indexes, Synonyms, Views etc. as Per the Database Software.

Database Should Have The Ability To Provide Access to External Tables for Files Outside the Database, as if The Rows in the Files Were Rows in the Table.

The Course of Developing an Application Consists of...

- Creating Structures (Tables & Indexes)
- Creating Synonyms for the Object Names.
- View Objects in Different Databases.
- Restricting Access to the Objects.
- With in The Oracle Database, the Basic Structure is a Table Used to Store Data.

The Different Tables Supported by Oracle Database are...

- Relational Tables
- Object Relational Tables
- Index Organized Tables
- External Tables
- Partitioned Tables
- Materialized Views
- Temporary Tables
- Clustered Tables
- Dropped Tables

Note: A Database Application may Contain all the Above Categories of Tables OR May Contain Few Depending on the Application Type.

As the Data in the Database Grows, The Speed and Performance of Database Decreases.

The Following Indexes are Provided by Oracle Products for Speed of Database...

- B+ Tree Indexes.
- Bitmap Indexes.
- Reverse Key Indexes.
- Function Based Indexes.
- Partitioned Indexes.
- Text Indexes.
- Domain Indexes.

Let us understand The Oracle Style for Data Storage

All the Logical Structures in the Database must be stored with in the Database Only.

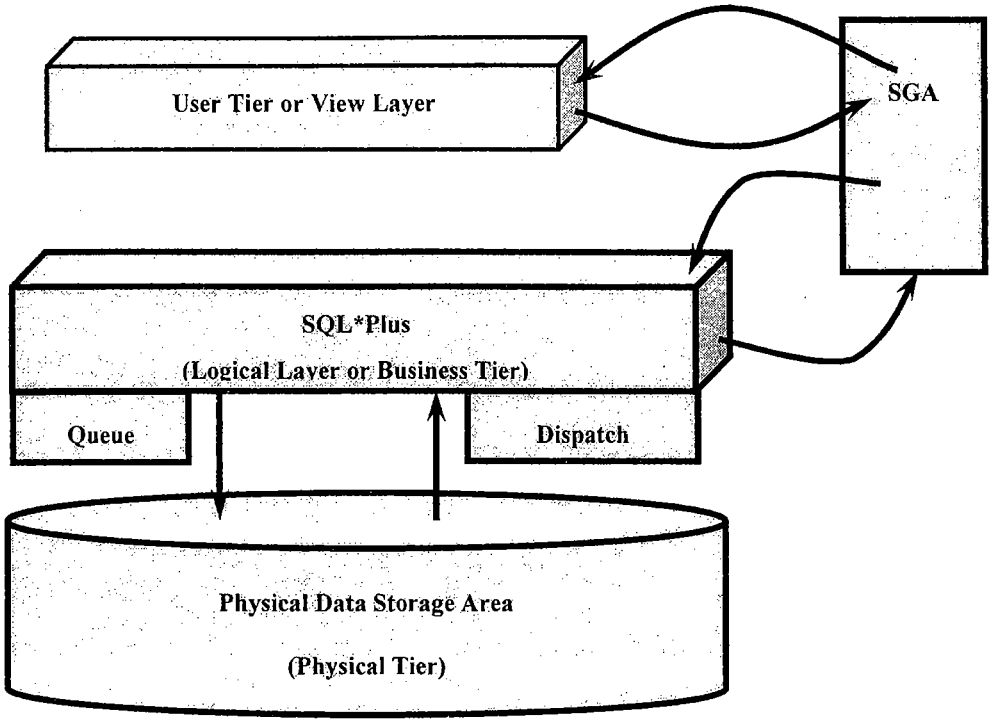
Oracle Maintains a Data Dictionary, Which Records Metadata About all the Database Objects.

The Database Objects Which Need Physical Storage Space on the Computer System, are Allocated Space Within a Table Space.

Table Space :

- It is a Logical Storage Unit within Oracle Database and is not Visible in the File System of the Machine on Which the Database Resides.
- The Table Space Builds the Bridge Between the Oracle Database and the File System in which the Table's OR Index Data is Stored.
- It Consists of One OR More Number of Data Files of the Database.
- A Data File can be Part of One and Only One Table Space.
- There are Three Types of Table Spaces in Oracle:
  - Permanent Table Spaces.
  - Undo Table Spaces.
  - Temporary Tables Spaces.
- Each Table OR Index stored in an Oracle Database Belongs to a Table Space.
- As Per Oracle 10g Minimum Two Table Spaces are Compulsory They are System and SysAux Table Space, Both Used to Support Oracle Internal Management.

- The System Table Space is always Available When a Database is Open and cannot be Taken Offline.
- The System Table Space Stores the Data Dictionary OR Their Base Tables, Respectively.
- In 10g a Big File Table Space can be Created, Which can grow to The Size of Tera Bytes of Space on Disk.



General Facts about Relational Model

The Model was first outlined by Dr. E. F. Codd in 1970.

The Components of the Model are...

- Collection of Objects OR Relations that Store the Data.
- A Set of Operators that can Act on the Relations to Produce Other Relations.
- Data Integrity for Accuracy and Consistency
- A Relational Database Should Use Relations OR Two Dimensional Tables to Store Information.

System and Data Modeling Importance

Data Models:

Data Models Help in Exploring Ideas and Improve the Understanding of the Database Design FOR Both Developers And System Designers.

Purpose of Data Models:

- Communicate
- Categorize
- Describe
- Specify
- Investigate
- Evolve
- Analyze
- Imitate

Objective of The Models:

- To Produce a Model That Fits a Multitude of Users.
- Should be understood by An End User.
- Should Contain Sufficient Details for a Developer to Build the Database System.

Relational Database Properties:

- Should be Accessed and Modified by Executing STRUCTURED QUERY LANGUAGE (SQL) Statements Only.
- Should Contain a Collection of Tables With no Physical Pointers.
- Should Use a Set of Operators.
- Need Not Specify the Access Route to The Tables and Data.
- There is no Need to Identify How the Data is arranged physically.

How to Communicate With RDBMS?

- The Structured Query Language is used to Communicate With RDBMS.

Generic Features of Structured Query Language

- It Allows the User to Communicate With the Server.
- It is highly efficient.
- It is Easy to Learn and Use.
- Functionally Complete, By Allowing the User to
    - Define The System.
    - Retrieve and Manipulate the Data From The System.
- The General Dialect is Similar to English Language.
- The Entire Structure is built on Collection of Statements.
- It is Platform Independent and Architecture Independent.

About Oracle 8

- It is the First Object Capable Database.
- It Provides a New Engine Which Supports
    - Object Oriented Programming.
    - Complex Data Types.
    - Complex Business Objects.
- Full Compatibility with The Relational Concepts.
- It Provides OLTP Applications, With
- Sharing of Runtime Data Structures.
- Large Buffer Caches
- Deferrable Constrains.
- It Supports Client Server and Web Based Application which are Distributed and Multi Tired.
- It Can Scale Tens and Thousands of Concurrent Users.

About Oracle 8i

- It is The Database of Internet Computing.
- It Provides Advanced Tools to Manage All Types of Data in Web Sites.
- The Internet File System (IFS) Combines The Power FOR Ease of Use a File System.
- End Users Can Easily Access Files and Folders in Oracle IFS via a Variety of Protocols.
- It Enables the Users to Web - Enable Their Multimedia Data.
- It Provides Full, Native Integration With Microsoft Transaction Server (MTS).
- It Provides High Performance Internet Platform for e-Commerce and Data Warehousing.
- The core pieces of Oracle Internet Platform are
    - Browser Based Clients to Process Presentation.
    - Application Servers to Execute Business Logic and Serve Presentation Logic to Browser Based Clients.
    - Databases to Execute Database Intensive Business Logic and Serve Data.

Entity Relationship Model

- In an Effective System Data should be divided into Discrete Categories OR Entities.
- An ER – Model is an Illustration of Various Entities in a Business and the Relationships between Them.
- It is built During the Analysis Phase of the System Developing Life Cycle.
- ER – Model Separates The Information Required & The Business From the Activities Performed.

ER – Model Benefits

- It Documents Information for The Organization in a Clear, Precise Format.
- Provides a Clear Picture of the Scope of The Information Requirement.
- Provides an Easily Understood Pictorial Map for the Database Design.
- It Offers An Effective Framework FOR Integrating Multiple Applications.

Key Components in ER – Model

- Entity
  - It is a Thing of Significance about Which the Information Need to be known.
- Attributes
  - It is something that Describes OR Qualifies an Entity.
  - Each Attribute May be Mandatory OR Optional.
- Relation
  - It is any Rule OR Situation that Should be Satisfied Between the Attribute AND Attributes of an Entity OR Entities.

Relational Database Terminology

Row OR Tuple

- It Represents All Data Required for a Particular Instance in an Entity.
- Each Row in an Entity is Uniquely Identified By Declaring it As PRIMARY KEY OR UNIQUE .
- The Order of the Rows is Not Significant, While Retrieving the Data.

Column OR Attribute

- It Represents One Kind of Data in a Table Vertically Collected.
- The Column Order is not significant when storing the Data.

Field

- It can be found at the Intersection of a Row and a Column.
- A Field Can Have Only One Value, OR May Not Have a Value At All.
- The Absence of Value in Oracle is represented as NULL.

Relating Multiple Tables

- Each Table Should Contain Data that Describes Exactly Only one Entity.
- Data about Different Entities is Stored in Different tables.
- RDBMS enables the data in one table to be related to another table by using the foreign keys.
- A Foreign Key is a Column OR a Set of Columns That Refer to a Primary Key in the Same Table OR Another Table.

Structured Query Language Statements in Oracle

- ORACLE SQL Compiles with Industry Accepted Standards.
- The Different Categories Into Which the SQL Statements Fall Are As Follows.

Data Retrieval Statement

- SELECT Statement

Data Manipulation Language Statements

- INSERT Statement.
- UPDATE Statement.
- DELETE Statement.

Data Definition Language Statements

- CREATE Statement.
- ALTER Statement.
- DROP Statement.
- RENAME Statement.
- TRUNCATE statement.

Transaction Control Language Statements

- COMMIT Statement.
- ROLLBACK Statement.
- SAVEPOINT Statement.

Data Control Language Statements

- GRANT Statement.
- REVOKE Statement.

About The SQL Buffer

- All Commands of SQL are Typed at the SQL prompt.
- Only One SQL Statement is Managed in The SQL Buffer.
- The Current SQL Statement Replaces the Previous SQL Statement in the Buffer.
- The SQL Statement Can be Divided Into Different Lines Within the SQL Buffer.
- Only One Line i.e., The Current Line Can be Active at a Time in the SQL Buffer.
- At SQL Prompt, Editing is Possible Only in The Current SQL Buffer Line.
- Every statement of SQL should be terminated Using Semi Colon ";".
- One SQL statement can Contain Only One Semi Colon.
- To Run the Previous OR Current SQL Statement in the Buffer Type "/" at SQL Prompt.
- To Open The SQL Editor Type "ED" at SQL Prompt.

Connecting to Oracle OR SQL*Plus

- Double Click the SQL*Plus Short Cut on the Desktop.
- Start → Run → Type SQLPlus OR SQLPlusW in Open Box and Click OK.
- Start → Programs → Oracle → Application Development → SQL*Plus.
- In the Login Box OR Login Prompt Type the User Name and Password as Supplied by the Administrator.
- The Host String is Optional and is provided by the Administrator.

Creating and Managing Tables.

- An Oracle Database is a Collection of Multiple Data Structures.
- These Data Structures are Called as Database objects.
- The Major Database Objects Associated with Database Programmer are...
    - Table → Used to Store Data, Basic Unit in Oracle.
    - View → Logically represents subsets of data from one OR more tables.
    - Sequence → Used to Generate Primary Key values.
    - Synonym → Used to give alternate names to objects.
    - Index → It is Used to Improve The Performance of Some Queries.

Tables in ORACLE 8 / 9 / 10

- Tables can be Created at Any Time, Even When The Users are Using the Database.
- Size of The Table Need Not be Specified.
- The Structure of The Table Can be Modified Online.

Rules to Create A Table :

- The User Should Have Permission on CREATE TABLE Command, and Storage Area Should be Allocated.
- The Table Name Should Begin with a Letter and can be 1 – 30 Characters Long.
- Table Names can Contain Combination of...
  A – Z (OR) a - z (OR) 0 – 9 (OR) -, $, #
- Names Cannot be Duplicated for Another Object Name in the Same ORACLE Server.
- Names Cannot be Oracle Servers Reserved Words.
- Table Names Are Not Case Sensitive in Oracle.
- Table is a Collection of Attribute Names and Oracle Data Types and Required Width along With Required Constraints.

Create Table Statement

Syntax

```
SQL> CREATE TABLE <Table_Name>
      (
       Column_Name1  <DataType>(Width),
       Column_Name2  <DataType>(Width),
       Column_NameN  <DataType>(Width)
       );
```

Note

- All Data Types May Not Have The Width Property.
- No Two Columns in The Same Table Can Have the Same Name.

Building Blocks of SQL Statements

- Data Types.
- Literals.
- Format Models.
- NULLS.
- Pseudo Columns.
- Comments.
- Database Objects.
- Schema Object Names and Qualifiers.
- Syntax for Schema Objects.
- Parts of SQL Statements.

Data Types in Oracle

- Each Value in ORACLE is Manipulated by a Data Type.
- The Data Type Associates a Fixed Set of Properties With that Value Stored.
- The Values of One Data Type are Different From Another Data Type.
- The Data Type Defines The Domain of Values That Each Column Can Contain.

Built-in Data Types

- CHARACTER DATA TYPES.
- NUMBER DATA TYPES.
- LONG AND RAW DATA TYPES.
- DATETIME DATA TYPES.
- LARGE OBJECT DATATYPES.
- ROWID DATATYPES.

Character Data Types

- They Store Character Data, Which Can be Alphanumeric Data Collection.
- The Information can be
  - Words OR Free – Form Text
  - Database OR National Character set
- This Data Type is Less Restrictive Than Other Data Types and has Very Few Properties.
- The Data is Stored in Strings With Byte Collections as Values.
- The Information Can Belong to Either
  - 7 – Bit ASCII(American Standard Code for Information Interchange) Character Set.
  - EBCDIC (Extended Binary Coded Decimal Interchange Code)
- ORACLE Supports Both Single & Multi Byte Character Sets.
- The Different Character Data Types are :
  - CHAR
  - NCHAR

- VARCHAR2
- NVARCHAR2

CHAR Data Type

- It Specifies Fixed Length Character Strings.
- The Size should be specified.
- If the Data is less Than the Original Specified Size, Blank Pads Are Applied.
- The Default Length is 1 Byte and the Maximum is 2000 Bytes.
- The Size of a Character Can Range From 1 Byte to 4 Bytes Depending on The Database Character Set.

NCHAR Data Type

- First Time Defined in ORACLE 9i, and Contains Unicode Data Only.
- Maximum Length is Determined by the National Character Set Definition.
- Maximum Size is 2000 Bytes and Size Has to be Specified.
- Padded if Data is Shorter than Specified.

VARCHAR2 Data Type

- Specifies the Variable Length Character String.
- Minimum Size is 1 Byte and the Maximum Size is 4000 Bytes.
- It Occupies Only that Space for Which the Data is Supplied.

NVARCHAR2 Data Type

- First Time Defined in ORACLE 9i.
- It is Defined for UNICODE Data Only.
- The Minimum Size is 1 Byte and Maximum Size is 4000 Bytes.

Numeric Data Types

Number Data Type

- It Stores Zero, Positive and Negative Fixed and Floating Point Numbers.
- The Range of Magnitude is
  1.0 * 10-130 to 9.9...9 * 10+125
- The General Declaration is
  - NUMBER(P, S)

  P → It Specifies the Precision, i.e., The Total Number of Digits (1 to 38).
  S → It Specifies the Scale, i.e., the Number of Digits to the Right of the Decimal Point, Can Range From –84 to 127.

Float Data Type

- It Facilitates to Have a Decimal Point Anywhere From the First to The Last Digit, OR Can Have no Decimal Point at All.
- The Scale Value is Not Applicable to Floating Point Numbers, as the Number of Digits That Can appear After the Decimal Point is Not Restricted.

Syntax

- FLOAT → It Specifies a Floating Point Number With Decimal Precision 38 OR Binary Precision of 126.
- Float (b) → It specifies a Floating Point Number with Binary Precision b.
- The precision can range from 1 to 126.
  - To Convert From Binary to Decimal Precision Multiply 'b' by 0.30103.
  - To Convert From Decimal to Binary Precision Multiply the Decimal Precision by 3.32193.
- The Maximum of 126 Digits of Binary Precision is Roughly Equivalent to 38 Digits of Decimal Precision.

Long Data Type

- This Data Type Stores Variable Length Character Strings.
- It is Used to Store Very Lengthy Text Strings.
- The Length of LONG Values May be Limited By The Memory Available on the Computer.
- LONG Columns Can be Referenced in

- SELECT Lists
- SET Clause of UPDATE Statements
- VALUES Clause of INSERT Statements.

Restrictions

- A Single Table Can Contain Only One LONG Column.
- Object Types Cannot be Created on LONG Column Attribute.
- LONG Columns Cannot Appear in WHERE Clauses OR in Integrity Constraints.
- INDEXES Cannot be Created on LONG Columns.
- LONG Can be Returned Through a Function, but not Through a Stored Procedure.
- It Can be Declared in a PL/SQL Unit But Cannot be Referenced in SQL.

DATE & TIME Data Types

DATE Data Type

- It is Used to Store DATE and TIME Information.
- The Dates Can be Specified as Literals, Using the Gregorian Calendar.
- The Information Revealed by DATE is
  - Century, Year, Month, Date, Hour, Minute, Second
- The Default DATE Format in ORACLE is DD-MON-YY, and is Specified in NLS_DATE_ FORMAT Variable.
- The Default TIME Accepted by ORACLE DATE is 12:00:00 AM (Midnight).
- The Default DATE Accepted by ORACLE DATE is The First Day of the Current Month.
- The DATE Range Provided by ORACLE is JANUARY 1, 4712 BC to DECEMBER 31, 9999 AD.

TIMESTAMP Data Type

- It is an Extension of the DATE Data Type.
- It stores The Data in the Form of
  - Century, Year, Month, Date, Hour, Minute, Second

Syntax

- TIMESTAMP[{Fractional–Seconds–Precision}]
- Fractional–Seconds–Precision Optionally Specifies the Number of Digits in the Fractional Part of The SECONDS to be Considered in Datetime Field.
- It Can be a Number in the Range of 0 – 9 ,With Default as 6.

RAW & LONG RAW Data Types

- RAW and LONG RAW Data Types are Intended for Storage of Binary Data OR BYTE Strings.
- RAW and LONG RAW are Variable Length Data Types.
- They are Mostly Used to Store Graphics, Sounds, Scanned Documents etc.
- The ORACLE Converts the RAW & LONG RAW Data into HEXADECIMAL Form.
- Each Hexadecimal Character Represents Four Bits of RAW Data.

LARGE OBJECT (LOB) Data Types

- The Built in LOB Data Types are
  * BLOB * CLOB * NCLOB
- These Data Types Store the Data Internally.
- The Bfile is a LOB Which Stores the Data Externally.
- The LOB Data Types Can Store Large and Unstructured Data Like Text, Image, Video and Spatial Data.
- The Maximum Storage Size is Up to 4 GB.
- LOB Columns Contain LOB Locators, Which Can Refer to Out-of-Line OR In-Line LOB Values.
- LOB's Selection Actually Returns the LOB's Locator.

BLOB Data Type

- It Stores Unstructured Binary Large Objects.

- They are Bit Streams With no Character Set Semantics.
- They are Provided with Full Transactional Support with Special Methods.

CLOB Data Type

- It Stores Single Byte and Multi Byte Character Data.
- Both Fixed Width and Variable Width Character sets are Supported.
- They are Provided With Full Transactional Support with Special Methods.

NCLOB Data Type

- It Stores UNICODE Data Using the NATIONAL CHARACTER Set.

BFILE Data Type

- It Enables Access to Binary File LOB's Which are Stored in the File Systems Outside ORACLE.
- A BFILE Column OR The Attribute Stores the BFILE Locator.
- The BFILE Locator Maintains The Directory Alias and the Filename.
- The Binary File LOB's do not Participate in Transactions and are Not Recoverable.
- The Maximum Size is 4 GB.

ROWID Data Type

- Each Row in the Database Has an Address.
- The Rows Address Can be Queried Using the PSEUDO Column ROWID.
- ROWID's Efficiently Support Partitioned Tables and Indexes.

Illustrative Example To Create A Table

```
SQL> CREATE TABLE Students
     (
     StudID      NUMBER(6),
     Fname       VARCHAR2(30),
     Lname       VARCHAR2(30),
     DOB         DATE,
     DOJ DATE,
     Fees NUMBER(7,2),
     Gender      VARCHAR2(1)
     );
SQL> CREATE TABLE LabSpecification
     (
     ProdID          NUMBER(6),
     ProdPhoto       BLOB,
     ProdGraphic     BFILE,
     ProdDesc        LONG
     );
```

Populating the Data into Tables

INSERT Statement

The INSERT Statement is Used to Add Rows To a

- Relational Table.
- Views Base Table.
- A Partition of a Partition Table.
- A Sub Partition of a Composite Partitioned Table
- An Object Table
- An Object View's Base Table.

Insertion of Data into a Table Can be Executed in Two Ways.

- Conventional INSERT.
- Direct – Path INSERT.

In Conventional Insert Statement, ORACLE Reuses Free Space in the Table into Which the Data in Being Inserted and Maintains Referential Integrity Constraints.

In Direct - Path Insert, ORACLE Appends the Inserted Data after Existing Data in the Table; the Free Space is not reused.

Syntax

```
SQL> INSERT INTO <Table_Name>(COL1, COL2, ..... )
        VALUES(VALUE1, VALUE2, ....);
```

Let us understand the Different Situations

Inserting Data into All Columns of a Table

```
SQL> INSERT INTO Students
     VALUES (1234, 'SAMPATH', 'KUMAR', '29-JAN-80', '30-MAR-95', 25000, 'M' );
```

- In This Case the Values Should be Provided to All the Columns That Exist Inside the Table.
- The Order of Values Declared in the VALUES Clause Should Follow the Original Order of the Columns in Table.
- The CHAR, VARCHAR and DATE Type Data should be declared in Single Quotes.
- Numerical Information can be applied normally.

Inserting Data into Required Columns

```
SQL>INSERT INTO Students ( Studno, Fname, Lname, DOJ, Gender )
     VALUES ( 1235, 'Raj', 'Kumar', '20-Feb-85', 'M' );
```

- In This Case the Order of Columns Declared in INSERT Need Not Be the Same As That of the Original Table Order.
- The Data Values in the VALUES Clause Should Match With that of INSERT List.
- The Columns Not Supplied With Data are Filled With NULL Values, Until the NOT NULL Constraint is Declared.

Inserting NULL Values into Table

- NULL Values can be inserted in two ways.
  - IMPLICIT → Omit The Column From List.
  - EXPLICT → Specify The NULL Keyword.
- It is Better To Specify Empty String '' Only FOR Character Strings and Dates.
- The Targeted Column Should Not be Set With NOT NULL Constraint.

```
SQL>INSERT INTO Students ( StudNo, Fname, Lname, DOB, DOJ, Fees, Gender ) VALUES
                         ( 1234, 'Krishna', NULL, '', '28-FEB-04', NULL, 'M' );
```

Inserting Special Values into Table

SYSDATE Function

- It is a PSEUDO COLUMN Provided by The Oracle.
- The Function Returns the CURRENT DATE and TIME from the System Clock.

USER Function

- It is Special Function, Which Records the Current USER Name.

```
SQL>INSERT INTO Students ( StudNo, Fname, DOJ, Fees, Gender, InsertBy ) VALUES
                         ( 1234, 'Mohan', SYSDATE, 25000, 'M', USER );
```

Supplying Data at Run Time

Substitution Variables

- These Variables are Used to Store Values Temporarily.
- The Values can be Stored Temporarily Through
  - Single Ampersand (&)
  - Double Ampersand (&&)
  - DEFINE and ACCEPT Commands
- The Single Ampersand Substitution Variable Applies for Each Instance When The SQL Statement is Create OR Executed.
- The Double Ampersand Substitution Variable Applies for All Instances Until That SQL Statement is Existing.

Single Ampersand Substitution Variable

```
SQL>INSERT INTO Students ( StudID, Fname, LName, Dob, Fees) VALUES
                         ( &StudNo, '&FirstName', &LastName, '&DateofBirth', 25000 );
```

Double Ampersand Substitution Variable

```
SQL>INSERT INTO Students ( StudNo, Fname, Lname, Dob, Doj, Fees, Gender ) VALUES
      ( &StudNo, '&FirstName', '&LastName', '&Dob', SYSDATE, &&Fees, '&Gender' );
```

Saving The SQL Buffer Text as SQL Script.

Method 1

- At the SQL Prompt Use the Save Command including the Required Path and File Name.

```
SQL> SAVE C:\MyScripts\First.SQL
```

- To Replace the Existing SQL Script File With New Information.

```
SQL>SAVE C:\MyScripts\First.SQL REPLACE
```

Method 2

- Open Any Text Editor Type the Required SQL Statement.
- SAVE The SQL Statement in the Required Drive and Folder USING SQL Extension.

Data Retrieval Standards Using SELECT Statement

Querying the Data From Tables

QUERY

- It is an Operation That Retrieves Data From One OR More TABLES OR VIEWS.

SELECT Statement

- The SELECT Statement is Used to Retrieve Data From One OR More
  - TABLES.
  - OBJECT TABLES.
  - VIEWS.
  - OBJECT VIEWS.
  - MATERIALIZED VIEWS.

Prerequisites

- The User Must Have the SELECT Privileges on the Specified Object.
- The SELECT ANY TABLE Allows to Select the Data from Any Recognized Object.

Capabilities of SQL SELECT Statement

- The SELECT Statement Can be Used to Select OR Retrieve Data From the Object Using Any One of the Following Criteria.
  - SELECTION
  - PROJECTION
  - JOIN
- SELECTION
  - It Chooses the Rows in a Table That are Expected to be Returned by a Query.
- PROJECTION
  - It Chooses the Columns in a Table that are Expected to Return by a Query.
- JOIN
  - It Chooses the Data in From One OR More Number of Tables by Creating a Link Between Them.

Basic SELECT Syntax

```
SQL> SELECT
        [DISTINCT]
        [*] {Column1 [Alias] , ..... }
        FROM  Table_Name;
```

- SELECT Keyword Identifies Columns.
- FROM Clause Identifies Tables .

SELECT → Specifies a List of Columns.

DISTINCT → Suppresses Duplicates.

* → Projection Operator to Select All Columns from the Table.

COLUMN → Selects the Named Column.

Alias → Gives Selected Columns Alternate Column Name.

FROM Table_Name → Specifies the Table Containing the Columns.

Sample Database Used :

Table Name: DEPT

Column Name	Data Type
Deptno	NUMBER
Dname	VARCHAR2
Loc	VARCHAR2

Table Name: Emp

Column Name	Data Type
Empno	NUMBER
Ename	VARCHAR2
Deptno	NUMBER
Job	VARCHAR2
Sal	NUMBER
Comm	NUMBER
Mgr	NUMBER
HireDate	DATE

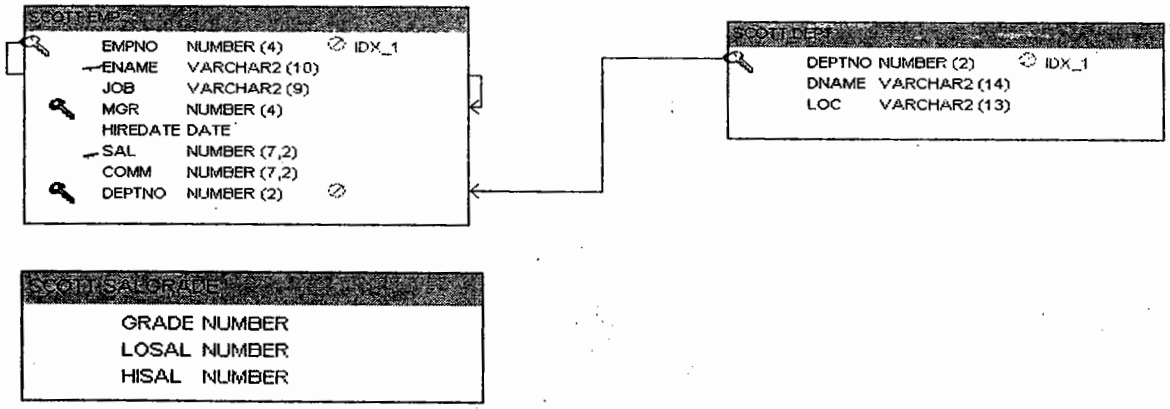
Table Name: SalGrade

Column Name	Data Type
Hisal	NUMBER
Losal	NUMBER
Grade	NUMBER

ER Model FOR Class Room Sessions

Retrieving Data From All Columns of a Table

- FOR This Purpose The Projection Operator '*' is Used.
- The Operator Projects Data From All The Columns Existing in The Table With All Records.
- The Data is Displayed in a Table Format.



Illustrative Examples

SQL> SELECT * FROM Emp;

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7839	KING	PRESIDENT		17-NOV-81	5000		10
7698	BLAKE	MANAGER	7839	01-MAY-81	2850		30
7782	CLARK	MANAGER	7839	09-JUN-81	2450		10
7566	JONES	MANAGER	7839	02-APR-81	2975		20
7654	MARTIN	SALESMAN	7698	28-SEP-81	1250	1400	30
7499	ALLEN	SALESMAN	7698	20-FEB-81	1600	300	30
7844	TURNER	SALESMAN	7698	08-SEP-81	1500	0	30
7900	JAMES	CLERK	7698	03-DEC-81	950		30
7521	WARD	SALESMAN	7698	22-FEB-81	1250	500	30
7902	FORD	ANALYST	7566	03-DEC-81	3000		20
7369	SMITH	CLERK	7902	17-DEC-80	800		20
7788	SCOTT	ANALYST	7566	09-DEC-82	3000		20
7876	ADAMS	CLERK	7788	12-JAN-83	1100		20
7934	MILLER	CLERK	7782	23-JAN-82	1300		10

14 rows selected.

SQL> SELECT * FROM Dept;

DEPTNO	DNAME	LOC
10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS
30	SALES	CHICAGO
40	OPERATIONS	BOSTON

SQL> SELECT * FROM SalGrade;

GRADE	LOSAL	HISAL
1	700	1200
2	1201	1400
3	1401	2000
4	2001	3000
5	3001	9999

- The SELECT Statement can be typed in one Line OR can be Split into Multiple Lines.

Retrieving Data From Specific Columns

SQL> SELECT Empno, Ename, Sal FROM Emp;
SQL> SELECT Ename, Job, Sal, Deptno FORM Emp;
SQL> SELECT Deptno, Dname, Loc FROM Dept;
SQL> SELECT HiSal, LoSal, Grade FROM SalGrade;
SQL> SELECT Empno, Ename, Sal, HireDate FROM Emp;

- The Column Names Need Not Be in The Same Order as Table.
- The Columns Should be Separated Using Comma.
- The Column Names Can be Separated Onto Different Lines Within the SQL BUFFER.
- The Casing of Column Names is Not Important.

Column Heading Defaults

- The Default Justification of The Data After it is Retrieved From the Table is..
  - LEFT → DATE and CHARACTER Data.
  - RIGHT → NUMERIC Data.
- The Default Display of the Data is Always is UPPER Casing.
- The CHARACTER and DATE Column Headings can be Truncated, But Number Columns Cannot be Truncated.

Applying Arithmetical Operations

- Arithmetic Expressions Can be Implemented Through SELECT Statement.
- Arithmetic Expressions Can be Implemented to...
  - Modify the Way the Data is Displayed.
  - Perform Calculations.
  - Implement WHAT – IF Scenarios.
- An Arithmetic Expression can contain
  - Simple Column Names
  - Constant Numeric Values
  - Arithmetic Operators.

Arithmetic Operators

- The Arithmetic Operators Can be Used to Create Expressions on NUMBER and DATE Data Type Columns.
- The Arithmetic Operators Supported Are
  - Addition → +
  - Subtraction → -
  - Multiply → *
  - Divide → /
- The Arithmetic Operators Can be Used in Any Clause of a SQL Statement, Except the FROM Clause.
- SQL*Plus Ignores Blank Spaces Before and After the Arithmetic Operator.

```
SQL> SELECT Empno, Ename, Sal, Sal + 500  FROM Emp;
SQL> SELECT Empno, Ename, Sal, Sal – 1000 FROM Emp;
```

Operator Precedence

- Multiplication and Division Take Priority over Addition and Subtraction(*, /, +, -)
- Operators of the Same Priority are Evaluated from Left to Right.
- To Prioritize Evaluation and to Increase Clarity Parenthesis Can be Implemented.

```
SQL> SELECT Empno, Ename, Sal, (12 * Sal ) + 100  FROM Emp;
SQL> SELECT Empno, Ename, Sal, 12 * (Sal + 500)  FROM Emp;
```

Handling NULL Values

NULL: It is a Value Which is...

- Unavailable.
- Unassigned.
- Unknown.
- Inapplicable.

- A NULL is Not Same as Zero OR Blank Space.
- If a Row Lacks the Data for a Particular Column, Than That Value is Said to Be NULL OR to Containing NULL.

```
SQL> SELECT Ename, Job, Sal, Comm FROM Emp;
```

- If Any Column Value in an Arithmetic Expression is NULL, The Overall Result is Also NULL.
- The Above Situation is Termed as NULL PROPAGATION and Has to be Handled Very Carefully.

```
SQL> SELECT Ename, Job, Sal, Comm, Sal + Comm  FROM Emp;
SQL> SELECT Ename, Job, Sal, Comm, 12 * (Sal + Comm)  FROM Emp;
```

NVL Function

- The NVL Function is Used to Convert a NULL Value to an Actual Value.

Syntax

NVL(Expr1, Expr2)

Expr1: It is the Source Value OR Expression That May Contain NULL.

Expr2: It is the Target Value for Converting NULL.

- NVL Function Can be Used to Convert Any Data Type, The Return Value is Always The Same as the Data Type of Expr1.
- The Data Types of The Source And Destination Must Match.
  - NVL(Comm, 0)
  - NVL(Hiredate, '01-JUN-99')
  - NVL(Job , 'Not Assigned')

```
SQL> SELECT Ename, Sal, Comm, Sal + NVL(Comm, 0)  FROM Emp;
SQL> SELECT Ename, Sal, Comm, (Sal * 12) + NVL(Comm, 0)FROM Emp;
SQL> SELECT Ename, Sal, Comm, (Sal + 500) + NVL (Comm, 0) FROM Emp;
```

Working With Aliases

- An ALIAS is an Alternate Name Given for any ORACLE OBJECT.
- Aliases in Oracle are of two types…

Column Alias

- Column Alias Renames a Column Heading in a Query.
- The Column Alias is Specified in The SELECT List by Declaring the Alias After the Column Name by Using the Space Separator.
- ALIAS Heading Appears in UPPER Casing by Default.
- The Alias Should be Declared in Double Quotes if it is Against the Specifications of Naming Conversions of Oracle.
- The AS Keyword Can be Used Between The Column Name and Alias.
- An Alias Effectively Renames the SELECT List Item for the Duration of that Query Only.
- An Alias Cannot be Used, Any Where in THE SELECT List for Operational Purpose.

Table Alias

- Table Alias Renames the Original Name of the Table in a SQL Statement.
- Table Aliases are Very Important When Working With Self Joins.
- The Table Alias is Applied for the Current SQL Statement Only.
- It is an Important Source When Implementing the Standards of MERGE Statements, Correlated Queries and Analytic Functions.

```
SQL> SELECT  Empno  Numbers, Ename  Name,  Sal  "Basic Salary", Job Designation FROM Emp;
SQL> SELECT  Deptno AS  "Department ID", Dname AS  "Department Name", Loc  AS  Place
                                                                          FROM Dept;
SQL> SELECT Hisal  As  "Maximum Range", Losal  As  "Minimum Range",
            Grade FROM Salgrade ;
```

Literals in ORACLE

- A LITERAL and a CONSTANT Value are Synonyms to One Another and Refer to a Fixed Data Value.
- The Types of LITERALS Recognized by ORACLE are…
  - TEXT Literals.
  - INTEGER Literals.
  - NUMBER Literals.
  - INTERVAL Literals.

TEXT Literals

- It Specifies a TEXT OR CHARACTER literal.
- It is Used to Specify Values Whenever 'TEXT' OR CHAR Appear in
  - Expressions
  - Conditions
  - SQL Function
  - SQL Statements.
- TEXT Literal Should be Enclosed in Single Quotes.
- They have Properties of Both CHAR and VARCHAR2 Data Types.
- A TEXT Literal Can Have a Maximum Length of 4000 Bytes.

Example:

'Employee Information'
'Manager''s Specification'
N'nchar Literal'

Using LITERAL Character Strings

- A Literal that is Declared in a SELECT List Can be a CHARACTER, a NUMBER, OR a DATE.
- A Literal is Not a Column Name OR a Column Alias.
- A LITERAL is Printed for Each Row, That is Retrieved by the SELECT Statement.
- LITERAL Strings of Free-Form Text Can be Included in The Query as Per The Requirement.
- DATE and CHARACTER Literals Must be Enclosed Within the Single Quotation Marks.
- LITERALS Increase the Readability of The Output.
- A LITERAL is Printed for Each Row, That is Retrieved by the SELECT Statement.
- LITERAL Strings of Free-Form Text Can be Included in The Query as Per The Requirement.
- DATE and CHARACTER Literals Must be Enclosed Within the Single Quotation Marks.
- LITERALS Increase the Readability of The Output.

```
SQL> SELECT Ename||' : '||' Month Salary = '|| Sal AS Salaries FROM Emp;
SQL> SELECT  'The Designation of '|| Ename||' is '||Job As Designation  FROM Emp;
SQL> SELECT  'The Annual Salary of ' ||Ename||' is '||Sal * 12 AS Annual_Salary FROM Emp;
SQL> SELECT Dname||' Department is Located At '||Loc Locations FROM Dept;
SQL> SELECT Ename|| ' Joined The Organization on '||Hiredate FROM Emp;
SQL> SELECT  Ename||' Works in Department Number '||Deptno ||' as '||Job FROM Emp;
```

Applying Concatenation Operator

- The Concatenation Operator Links Columns to Other Columns, Arithmetic Expressions, or Constant Values.
- Columns on Either Side of the Operator are Combined to Make a Single Output Column.
- The Resultant Column is Treated as an CHARACTER EXPRESSION.
- The Concatenation Operator is Represented in ORACLE by Double Pipe Symbol (||).

```
SQL> SELECT Empno||'  '||Ename||', Designation is ' ||Job "Employees Information"  FROM  Emp;
```

Suppressing Duplicate Rows in Output

- Until it is Instructed SQL*Plus Displays the Results of a Query Without Eliminating Duplicate Rows.
- To Eliminate the Duplicate Rows in the Result, the DISTINCT Keyword is Used.
- Multiple Columns can be Declared After the DISTINCT Qualifier.
- The DISTINCT Qualifier Affects all the Selected Columns, and Represents a DISTINCT Combination of the Columns.

```
SQL> SELECT DISTINCT Deptno  FROM Emp;
SQL> SELECT DISTINCT Deptno  FROM Emp;
SQL> SELECT DISTINCT MGR FROM Emp;
SQL> SELECT DISTINCT Job, Deptno  FROM Emp;
SQL> SELECT DISTINCT Deptno, Job FROM Emp;
```

- Filtering of Records
- The Number of Rows Returned by a QUERY can be Limited Using the WHERE Clause.
- A WHERE Clause Contains a Condition That Must be Met and Should Directly Follow the FROM Clause.

Syntax

```
SQL> SELECT [DISTINCT] [*] {Column1 [Alias] , ....}
        FROM  Table_Name
        [WHERE Condition(s)];
```

- The WHERE Clause Can Compare
  - Values in Columns
  - Literal Values
  - Arithmetic Expressions

- Functions
- The Components of WHERE Clause are
  - Column Name.
  - Comparison Operator.
  - Column Name (OR) Constant (OR) List of Values.
- The CHARACTER Strings and DATES Should be Enclosed in Single Quotation Marks.
- CHARACTER Values are Case Sensitive and DATE Values are Format Sensitive (DD-MON-YY)
- The Comparison Operators are Used in Such Conditions That Compare One Expression to Another.
- The Different Comparison Operators are
  - Equality Operator → =
  - Not Equality Operator → <>, !=, ^=.
  - Greater Than Operator → >
  - Less Than Operator → <
  - Greater Than or Equal to Operator → >=
  - Less Than or Equal to Operator → <=
- The Format of The WHERE Clause is
  WHERE Expr OPERATOR VALUE.

```
SQL> SELECT  Ename, Sal, Job FROM Emp  WHERE Job = 'MANAGER';
SQL> SELECT Ename,  Hiredate , Deptno, Sal  FROM Emp  WHERE Deptno = 10;
SQL> SELECT Empno, Ename, Sal  FROM Emp  WHERE Sal >= 3000;
SQL> SELECT  Ename||' Joined on '||Hiredate  "Employees Joining Dates" FROM Emp
       WHERE Hiredate = '01-JAN-95';
SQL> SELECT  Ename|| ' Works in Department'||Deptno "Employees and Deaprtments"
       FROM Emp  WHERE Deptno <> 20;
SQL> SELECT Ename, Sal, Deptno, Job FROM Emp  WHERE Job <> 'CLERK';
SQL> SELECT Ename Name , Sal Basic,  Sal * 12 Annual FROM Emp WHERE Sal * 12 > 60000;
```

Applying Logical Operators to Filters

- The LOGICAL OPERATORS Combine the Results of Two COMPONENT Conditions to Produce a Single Result.
- The LOGICAL OPERATORS Provided by ORACLE are
  - Logical Conjunction Operator → AND
  - Logical Disjunction Operator → OR
  - Logical Negation operator → NOT

AND Operator

- It Returns TRUE if Both or All Component Conditions are TRUE.
- It Returns FALSE if Either is FALSE, Else Returns Unknown.

Truth Table

AND	TRUE	FALSE	NULL
TRUE	T	F	NULL
FALSE	F	F	F
NULL	NULL	F	NULL

```
SQL> SELECT Ename, Sal, Deptno, Job FROM Emp WHERE Deptno=20 AND Job='MANAGER';
SQL> SELECT Empno, Ename, Job, Sal  FROM Emp WHERE Sal >= 1100 AND Job = 'CLERK';
SQL> SELECT Empno, Ename , Job , Sal  FROM Emp WHERE Deptno = 10 AND Job = 'CLERK';
SQL> SELECT Ename, Sal , Job  FROM Emp  WHERE Sal >= 1500 AND Sal > 5000;
SQL> SELECT Ename, Sal, Job  FROM Emp  WHERE (Sal >= 1500 AND Sal <= 5000) AND
       Job = 'MANAGER';
```

OR Operator

- It Returns TRUE if Either of the Component Condition is TRUE.

- It Returns FALSE if Both are FALSE, Else Returns Unknown.

Truth Table

OR	TRUE	FALSE	NULL
TRUE	T	T	T
FALSE	T	F	NULL
NULL	T	NULL	NULL

```
SQL> SELECT Ename, Sal, Deptno, Job FROM Emp WHERE  Deptno = 20 OR Job= 'MANAGER';
SQL> SELECT Empno, Ename, Job, Sal  FROM Emp  WHERE Sal >= 1100 OR Job = 'CLERK';
SQL> SELECT Empno, Ename, Job, Sal  FROM Emp  WHERE Deptno=10 OR Job ='MANAGER';
SQL> SELECT Ename, Sal, Job  FROM Emp  WHERE Sal >= 1500 OR Sal >= 5000;
SQL> SELECT Ename, Sal, Job, Deptno  FROM Emp  WHERE Deptno = 10 OR Deptno = 20;
SQL> SELECT Ename, Sal, Job  FROM Emp  WHERE Job = 'CLERK' OR Job = 'MANAGER';
SQL> SELECT Ename, Sal, Job FROM Emp  WHERE (Sal <= 2500 OR Sal >= 5000) OR
        Job = 'MANAGER';
```

NOT Operator

- It Returns TRUE if the Following Condition is FALSE.
- It Returns FALSE if the Following Condition is TRUE.
- If the Condition is Unknown, it Returns Unknown.

Truth Table

NOT	TRUE	FALSE	NULL
NOT	F	F	NULL

```
SQL> SELECT Ename, Sal, Job  FROM Emp WHERE NOT Job = 'MANAGER';
SQL> SELECT Ename, Sal, Job  FROM Emp  WHERE NOT Sal > 5000;
SQL> SELECT Ename, Sal, Job FROM Emp WHERE NOT Sal < 5000 ;
SQL> SELECT Ename, Sal, Hiredate  FROM Emp WHERE NOT Hiredate = '20-FEB-81';
SQL> SELECT Ename, Job, Sal, Deptno FROM Emp WHERE NOT Job = 'SALESMAN'  AND
                                                        Deptno = 30;
```

Combination of AND and OR Operators

```
SQL> SELECT Empno, Ename, Job, Sal  FROM Emp  WHERE (Sal > 1100 OR Job = 'CLERK')
                                                        AND Deptno = 20;
SQL> SELECT Empno, Ename, Job, Sal  FROM Emp  WHERE (Deptno = 10 AND
        Job = 'MANAGER' ) OR  Sal >= 3000;
SQL> SELECT Empno, Ename, Job, Sal FROM Emp  WHERE (Deptno = 10 AND
        Job = 'MANAGER' ) OR  (Deptno = 20 AND Sal >= 3000);
```

Some Things To Note

```
SQL> SELECT Ename, Sal, Job FROM Emp  WHERE Job > 'MANAGER';
SQL> SELECT Ename, Sal, Job FROM Emp  WHERE Job < 'MANAGER';
SQL> SELECT Ename, Sal, Hiredate  FROM Emp WHERE Hiredate > '20-FEB-1981';
SQL> SELECT Ename, Sal, Hiredate  FROM Emp WHERE Hiredate < '20-FEB-1981';
SQL> SELECT Ename, Sal, Hiredate  FROM Emp  WHERE Hiredate <> '20-FEB-1981';
SQL> SELECT Ename, Sal, Hiredate  FROM Emp  WHERE Job <> 'CLERK';
SQL> SELECT Ename, Sal, Comm  FROM Emp WHERE Comm IS NULL;
SQL> SELECT Ename, Sal, Comm  FROM Emp  WHERE Comm IS NOT NULL;
SQL> SELECT Ename, Sal, Job FROM Emp WHERE NOT Job > 'MANAGER';
SQL> SELECT Ename, Sal, Hiredate FROM Emp WHERE NOT Hiredate = '17-DEC-1980';
SQL> SELECT Ename, Sal, Hiredate FROM Emp WHERE NOT Hiredate > '17-DEC-1980';
SQL> SELECT Ename, Sal, Hiredate FROM Emp WHERE NOT Hiredate >'17-DECEMBER-1980';
```

Rules of Operator Precedence

- The Default Precedence Order is...
  - All Comparison Operators
  - NOT Operator
  - AND Operator
  - OR Operator
- The Precedence Can be Controlled Using Parenthesis.

```
SQL> SELECT Ename, Deptno, Job, Sal FROM Emp WHERE Deptno = 10 OR  Deptno = 20 AND
       Job = 'SALESMAN' AND  Sal > 2500 OR  Sal < 1500;
SQL> SELECT Ename, Deptno, Job, Sal FROM Emp WHERE Deptno = 10 OR
       (Deptno = 20 AND  Job = 'SALESMAN') AND  (Sal > 2500 OR Sal < 1500);
```

SQL*Plus Operators

BETWEEN ... AND ... Operator

- This Operator is Used to Display Rows Based on a Range of Values.
- The Declared Range is Inclusive.
- The Lower Limit Should be Declared First.
- The Negation of this Operator is NOT BETWEEN ... AND...

```
SQL> SELECT Ename, Sal, Job  FROM Emp WHERE Sal BETWEEN 1000 AND 1500;
SQL> SELECT Ename, Sal, Job FROM Emp WHERE Sal NOT BETWEEN 1000 AND 1500;
SQL> SELECT Ename, Sal, Job FROM Emp WHERE Job BETWEEN 'MANAGER' AND
                                                        'SALESMAN';
SQL> SELECT Ename, Sal, Job FROM Emp  WHERE Job  NOT BETWEEN 'MANAGER' AND
                                                        'SALESMAN';
SQL> SELECT Ename, Sal, Job, Hiredate FROM Emp  WHERE Hiredate
                                    BETWEEN  '17-FEB-1981' AND '20-JUN-1983';
SQL> SELECT Ename, Sal, Job, Hiredate FROM Emp  WHERE Hiredate
                                    NOT BETWEEN  '17-FEB-1981' AND '20-JUN-1983' ;
```

IN Operator

- The Operator is Used to Test for Values in a Specified List.
- The Operator Can be Used Upon Any Data type.
- The Negation of the Operator is NOT IN.

```
SQL> SELECT Ename, Sal, Job  FROM Emp WHERE Ename IN('FORD', 'ALLEN') ;
SQL> SELECT Ename , Sal, Job  FROM Emp WHERE Ename NOT IN('FORD', 'ALLEN');
SQL> SELECT Ename, Sal, Deptno FROM Emp WHERE Deptno IN(10, 30);
SQL> SELECT Ename, Sal, Deptno  FROM Emp WHERE Deptno NOT IN(10, 30);
SQL> SELECT Ename, Sal, Hiredate FROM Emp WHERE Hiredate
                                    IN('20-FEB-1981', '09-JUN-1981');
SQL> SELECT Ename, Sal, Hiredate  FROM Emp  WHERE Hiredate
                                    NOT IN('20-FEB-1981', '09-JUN-1981');
```

IS NULL Operator

- The Operator Tests for NULL Values.
- It is the Only Operator That can be Used to Test for NULL's.
- The Negation is IS NOT NULL.

```
SQL> SELECT Ename, Deptno, Comm  FROM Emp WHERE Comm IS NULL;
SQL> SELECT Ename, Deptno, Job, MGR FROM Emp WHERE MGR IS NULL;
SQL> SELECT Ename, Deptno, Comm FROM Emp WHERE Comm IS NOT NULL;
SQL> SELECT Ename, Deptno, Comm, MGR  FROM Emp WHERE MGR IS NOT NULL;
```

LIKE Operator

- The LIKE Operator is Used to Search for a Matching Character Patterns.
- The Character Pattern Matching Operation is Referred as a WILD CARD SEARCH.
- The Available WILD CARDS in Oracle are
  - % → Used to Represent Any Sequence of Zero or More Characters.

- _ → Represents Any Single Character, Only At That Position.
- The WILD CARD Symbols Can be Used in Any Combination With Literal Characters.
- For Finding Exact Match For '%' and '_' the ESCAPE Option Has to be Used along with '\' Symbol.

```
SQL> SELECT Ename, Job  FROM Emp WHERE Ename LIKE 'S%';
SQL> SELECT Ename, Job FROM Emp WHERE Ename NOT LIKE 'S%';
SQL> SELECT Ename, Job FROM Emp WHERE Ename LIKE '_A%';
SQL> SELECT Ename Job FROM Emp WHERE Ename NOT LIKE '_A%';
SQL> SELECT Ename Sal FROM Emp WHERE Ename = 'SM%';
SQL> SELECT Ename, Sal FROM Emp WHERE 'SM%' LIKE Ename;
SQL> SELECT Ename, Hiredate FROM Emp WHERE Hiredate LIKE '%-FEB-1981';
SQL> SELECT Ename, Hiredate FROM Emp WHERE Hiredate LIKE '03-%-1981';
SQL> SELECT * FROM Dept WHERE Dname LIKE '%\_%'  ESCAPE '\';
```

Ordering Information

- The Order of Rows Returned in the Result of a Query Undefined.
- The ORDER BY Clause Can be Used to SORT The Rows in the Required Order.
- The ORDER BY Clause Should be the Last Clause in the Order of All Clauses in The SELECT Statement.
- An Expression or an Alias Can be Specified to ORDER BY Clause for Sorting.
- Default Ordering of Data is Ascending
  - Numbers → 0 – 9
  - Dates → Earliest – Latest.
  - Strings → A – Z.
  - NULLS → Last.

Syntax

```
SQL> SELECT [DISTINCT] [*] {Column1 [Alias] , ....}
       FROM  Table_Name
       [WHERE Condition(s)]
       [ORDER BY  {Column, Expr}[ASC/DESC]];
```

- The Default Ordering Upon a Column is ASCENDING, to Change the Default Ordering DESC Should be Used After the Column Name.
- Sorting Can be Implemented on Column Aliases, and Can Also be Implemented Upon Multiple Columns.
- The Controversy of Sorting is Broken Only When There Arises a Conflict of Consistency Upon the Data in a Column.

```
SQL> SELECT Ename, Job, Deptno, Hiredate  FROM Emp ORDER BY Hiredate;
SQL> SELECT Ename, Job, Deptno, Hiredate FROM Emp ORDER BY Hiredate DESC;
SQL> SELECT Ename, Job, Sal FROM Emp WHERE Job = 'MANAGER' ORDER BY Sal;
SQL> SELECT Ename, Job, Sal FROM Emp WHERE Sal >= 2500 ORDER BY Job, Ename DESC;
SQL> SELECT Empno, Ename, Sal, Sal * 12 Annsal FROM Emp ORDER BY Annsal;
SQL> SELECT Empno, Ename, Sal  FROM Emp ORDER BY Deptno, Sal, Hiredate;
SQL> SELECT Empno, Ename, Sal  FROM Emp WHERE Sal >= 2000
                                        ORDER BY Hiredate,Sal DESC;
```

Experiencing the Power of SQL Functions

SQL Functions are Built into ORACLE and are available for use in Various Appropriate SQL Statements.

- The SQL Functions Can be Used to...
  - Perform Calculations on Data.
  - Modify Individual Data Items.
  - Manipulate Output for Groups of Rows.
  - Format DATES and NUMBERS for Display.
  - Convert Column Data Types.
- SQL Functions May Accept Arguments and Always Return a Value, and Can be Nested.
- If an SQL Function is Called with a NULL Argument, then a NULL is Returned.

SQL Function Types

- SQL Identifies Two Types of Functions
  - SINGLE Row Functions.
  - MULTIPLE Row Functions.

SINGLE Row Functions

- These Functions Return a Single Result for Every Row of a Queried Table or View.

MULTIPLE Row Functions

- These Functions Manipulate GROUPS of Rows and Return One Result Per Group of Rows.
- The Single Row Functions Can Appear in
  - SELECT List.
  - WHERE Clause and ORDER BY Clause.
  - START WITH Clause.
  - CONNECT BY Clause.
- The Single Row Functions are Categorized as...
  - CHARACTER Functions.
  - NUMBER Functions.
  - DATE Functions.
  - CONVERSION Functions.
- SINGLE Row Functions are Used to Manipulate Data Items.
- They Can Accept One or More Arguments and Return One Value For Each Row Returned By the Query.
- The Argument for a SINGLE Row Function can be...
  - User Supplied Constant.
  - Variable Value.
  - Column Name.
  - Expression.

Syntax

FunctionName(Column/Expr, [Arg1, Arg2,...])

Single Row Functions Features

- Acts on Each Row Returned in the Query.
- Returns One Result Per Row.
- May Return a Data Value of a Different Data Type Than That Referenced.
- May Expect One or More Arguments.
- Can be Used in SELECT, WHERE and ORDER BY Clauses.
- It Can be Nested.

Specification Behavior of Functions

Character Functions

- Accept Character Input and Can Return Both Character and Number Values.

Number Functions

- Accept Numeric Input and Return Numeric Values.

Date Functions

- Operate on Values of Date Data Type and Can Return Both Date and Number.

Conversion Function

- Covert a Value From One Data Type to Another Data Type.

General Functions

- NVL → Operates on NULL Values.
- DECODE → Operates on Any Data Type and Can Return Any Data Type.

Character Functions

- They are Functions That Return CHARACTER Values, Unless Stated.
- They Return the Data Type VARCHAR2, Limited to a Length of 4000 Bytes.
- If the Return Value Length Exceeds, Then the Return Value is Truncated, Without an Error.
- The Functions are Categorized as
  - Case Conversion Functions.
  - Character Manipulation Functions.

Case Conversion Functions

- These Functions are Used to Convert the Casing of the Existing Character from one Type to Another.
  - LOWER Function.
  - UPPER Function.
  - INITCAP Function.

Lower Function

- It Converts Alpha Character Values to Lower Case.
- The Return Value Has the Same Data Type as Argument CHAR Type (CHAR or VARCHAR2)

Syntax: LOWER(Column/Expression)

```
SQL> SELECT 'ORACLE CORPORATION' String, LOWER('ORACLE CORPORATION') Lower
                                                        FROM DUAL;
SQL> SELECT Ename, LOWER('MY INFORMATION')  Lower FROM Emp;
SQL> SELECT Ename, LOWER(Ename) Lower FROM Emp WHERE Job = 'MANAGER';
SQL> SELECT 'The '||Ename||'''s Designation is '||Job Employee FROM Emp
                                        WHERE LOWER(Job) = 'manager';
```

Upper Function

- It Converts the Alpha Character Values to Upper Case.
- The Return Value Has the Same Data Type as the Argument CHAR.

Syntax: UPPER(Column/Expression)

```
SQL> SELECT 'oracle corporation' String, UPPER('oracle corporation') Upper FROM DUAL;
SQL> SELECT Ename, UPPER('my information') Upper FROM DUAL;
SQL> SELECT Ename,LOWER(Ename),UPPER(Ename)  FROM Emp WHERE Job='MANAGER';
SQL> SELECT Ename, Job  FROM Emp WHERE Job = UPPER('Manager');
SQL> SELECT Ename, Job, Sal, Sal * 12 FROM Emp  WHERE Job =
                                                UPPER(LOWER('MANAGER'));
SQL> SELECT 'The '||Ename||'''s Designation is '||LOWER(Job)  FROM Emp
                                    WHERE Job = UPPER ('manager') ORDER BY Sal;
SQL> SELECT  UPPER('The '||Ename||'''s Basic Salary is Rupees '||Sal) FROM Emp
        WHERE Job IN('MANAGER', UPPER('clerk')) ORDER BY Sal DESC;
```

INITCAP Function

- It Converts the Alpha Character Values into Uppercase For The First Letter of Each Word, Keeping all Other Letters in Lower Case.
- Words are Delimited by White Spaces or Characters That are Not Alphanumeric.

Syntax: INITCAP(Column/Expression)

```
SQL> SELECT  'oracle corporation' String, INITCAP('oracle corporation') InitCap FROM DUAL;
SQL> SELECT  'The Job Title for '||INITCAP(Ename)||' is '||LOWER(Job) FROM Emp;
SQL> SELECT Ename, UPPER(Ename), LOWER(Ename), INITCAP(Ename) FROM Emp;
SQL> SELECT Empno, INITCAP(Ename), Deptno  FROM Emp
                                    WHERE Ename = UPPER('blake');
```

CONCAT Function

- It Concatenates the First Characters Value to the Second Character Value.
- It Accepts Only Two Parameters Accept.
- It Return The Character Data Type.

Syntax: CONCAT(Column1/Expr1, Column2/Expr2)

```
SQL> SELECT 'Oracle' String1, 'Corporation' String2, CONCAT('Oracle' , 'Corporation') Concat
                                                                    FROM DUAL;
SQL> SELECT Ename, Job, CONCAT(Ename, Job) Concat FROM Emp  WHERE Deptno = 10;
SQL> SELECT CONCAT ('The Employee Name is ' , INITCAP(Ename)) Info FROM Emp
                                                        WHERE Deptno IN(10,30);
SQL> SELECT CONCAT(CONCAT(INITCAP(Ename), ' is a '), Job) Job  FROM Emp
                                                    WHERE Deptno IN(10, 20);
```

SUB STRING Function

- Returns Specified Characters Form Character Value, Starting From a Specified Position 'm' to 'n' Characters Long.

Syntax: SUBSTR(Col/Expr, m, n)

Points to Remember

- If "m" is 0 , it is Treated as 1.
- If "m" is Positive, Oracle Counts From the Beginning of String to Find the First Character.
- If "m" is Negative, Oracle Counts Backwards From the End of The String.
- If "n" is Omitted , Oracle Returns All Characters to the End of String.
- If "n" is Less Than 1 or 0 , A NULL is Returned.
- Floating Point Numbers Passed as Arguments to SUBSTR are Automatically Converted to Integers.

```
SQL> SELECT 'ABCDEFGH' String, SUBSTR('ABCDEFGH' , 3, 4) SubString FROM DUAL;
SQL> SELECT  'ABCDEFGH' String, SUBSTR('ABCDEFGH' , -5, 4) SubString FROM DUAL;
SQL> SELECT  'ABCDEFGH' String, SUBSTR('ABCDEFGH' , 0 ,4) SubString FROM DUAL;
SQL> SELECT 'ABCDEFGH' String, SUBSTR('ABCDEFGH' , 4) SubString FROM DUAL;
SQL> SELECT 'ABCDEFGH' String, SUBSTR('ABCDEFGH', 4, 0) SubString  FROM DUAL;
SQL> SELECT  'ABCDEFGH' String, SUBSTR('ABCDEFGH', 4 , -2) SubString FROM DUAL;
SQL> SELECT Ename, Job  FROM Emp WHERE SUBSTR(Job, 4, 3) = UPPER('age');
SQL> SELECT CONCAT(INITCAP(Ename),  CONCAT(' is a ', CONCAT(INITCAP(
       SUBSTR(Job, 1, 3)), 'Eater.' ))) FROM Emp  WHERE SUBSTR(Job, 4, 3) = UPPER('Age');
```

LENGTH Function

- Returns the Number of Characters in a Value.
- If the String has Data Type CHAR, The Length Includes All Trailing Blanks.
- If The String is NULL , It Returns NULL .

Syntax: LENGTH(Column/Expression)

```
SQL> SELECT  'ORACLE' String, LENGTH('ORACLE') Length FROM DUAL;
SQL> SELECT  LENGTH (Ename)||' Characters exit in '||INITCAP(Ename) ||'''s Name.' AS
       "Names and Lengths"  FROM Emp;
SQL> SELECT INITCAP(Ename), Job  FROM Emp WHERE LENGTH(Job) = 7;
SQL> SELECT INTICAP(Ename), Job FROM Emp  WHERE
                        SUBSTR(Job, 4, LENGTH(SUBSTR(Job, 4, 3))) = 'AGE';
```

INSTRING Function

- It Returns The Numeric Position of a Named Character.

Syntax: INSTR(Column/Expression, Char, m, n)

- The INSTR Functions Search String for Substring that is Supplied.
- The Function Returns an Integer Indicating the Position of the Character in String That is The First Character of This Occurrence.
- Searches for Column OR Expression Beginning With its 'm'th Character For The 'n'th Occurrence of Char2, and Returns the Position of the Character in Char1, That is the First Character of This Occurrence.

- 'm' Can be Positive or Negative, if Negative Searches Backward From The End of Column OR Expression.
- The Value of 'n' Should be Positive .
- The Default Values of Both 'm' and 'n' Are 1.
- The Return Value is Relative to The Beginning of Char1 Regardless of The Value of 'm', and is Expressed in Characters.
- If the Search is Unsuccessful, the Return Value is Zero.

```
SQL> SELECT 'STRING' String, INSTR('STRING', 'R') InString FROM DUAL;
SQL> SELECT 'CORPORATE FOOR' String, INSTR('CORPORATE FOOR', 'OR', 3, 2) InString
       FROM DUAL;
SQL> SELECT 'CORPORATE FOOR' String, INSTR('CORPORATE FLOOR', 'OR', -3, 2)
                                                          InString FROM DUAL;
SQL> SELECT Job, INSTR(Job, 'A', 1, 2) Position FROM Emp WHERE Job = 'MANAGER';
SQL> SELECT Job, INSTR(Job, 'A', 2, 2) Position FROM Emp WHERE Job = 'MANAGER';
SQL> SELECT Job, INSTR(Job, 'A', 3, 2) Position FROM Emp WHERE Job = 'MANAGER';
SQL> SELECT Job, INSTR(Job, 'A', 2) Position FROM Emp WHERE Job = 'MANAGER';
```

LPAD Function

- Pads The Character Value Right Justified to a Total Width of 'n' Character Positions.
- The Default Padding Character is Space.

Syntax: LPAD(Char1, n, 'Char2')

```
SQL> SELECT 'Page 1' String, LPAD('Page 1', 15, '*.') LPadded  FROM DUAL;
SQL> SELECT 'Page 1' String, LPAD('Page 1', 15) LPadded FROM DUAL;
SQL> SELECT Ename, LPAD(Ename, 10, '-') LPadded FROM Emp WHERE Sal >= 2500;
```

RPAD Function

- Pads the Character Value Left Justified to a Total Width of 'n' Character Positions.
- The Default Padding Character is Space.

Syntax: RPAD(Char1, n, 'Char 2')

```
SQL> SELECT 'Page 1' String, RPAD('Page 1' , 15, '*.') RPadded FROM DUAL;
SQL> SELECT 'Page 1' String, RPAD('Page 1' , 15) RPadded FROM DUAL;
SQL> SELECT Ename, Rpad(Ename, 10, '-') RPadded FROM Emp  WHERE Sal > = 2500;
SQL> SELECT Ename,  LPAD(Ename, 10, '-') LPadded,  RPAD(Ename, 10, '-') RPadded
                                  FROM Emp;
SQL> SELECT Ename, LPAD(RPAD(Ename, 10, '-'), 15, '-') Centered FROM Emp;
```

LTRIM Function

- It Enables to TRIM Heading Characters From a Character String.
- All The Leftmost Characters That Appear in The SET are Removed.

Syntax: LTRIM(Char, SET)

```
SQL> SELECT 'xyzXxyLAST WORD' String, LTRIM('xyzXxyLAST WORD', 'xy') LTrimmed
                                                          FROM DUAL;
SQL> SELECT Job, LTRIM(Job, 'MAN') LTrimmed FROM Emp
                                  WHERE Job LIKE 'MANAGER';
```

RTRIM Function

- It Enables the Trimming of Trailing Characters From a Character STRING.
- All the Right Most Characters That Appear in The SET are Removed.

Syntax: RTRIM(Char, SET)

```
SQL> SELECT  'BROWNINGyxXxy' String, RTRIM('BROWNINGyxXxy', 'xy') RTrimmed
                                                          FROM DUAL;
SQL> SELECT  RTRIM(Job, 'ER'), Job  FROM Emp  WHERE LTRIM(Job, 'MAN') LIKE 'GER';
```

TRIM Function (8i)

- It Enables to TRIM Heading or Trailing Characters or Both From a Character String.
- If LEADING is Specified Concentrates On Leading Characters.
- If TRAILING is Specified Concentrates On Trailing Characters.
- If BOTH OR None is Specified Concentrates Both on LEADING and TRAILING.
- Returns the VARCHAR2 Type.

Syntax:
TRIM(LEADING/TRAILING/BOTH, Trimchar FROM TrimSource)

```
SQL> SELECT 'MITHSS' String, TRIM('S' FROM 'MITHSS') Trimmed FROM DUAL;
SQL> SELECT 'SSMITH' String, TRIM('S' FROM 'SSMITH') Trimmed FROM DUAL;
SQL> SELECT 'SSMITHSS' String, TRIM('S' FROM 'SSMITHSS') Trimmed FROM DUAL;
SQL> SELECT 'SSMITHSS' String, TRIM(TRAILING 'S' FROM 'SSMITHSS') Trimmed
                                                              FROM DUAL;
SQL> SELECT 'SSMITHSS' String, TRIM(LEADING 'S' FROM 'SSMITHSS') Trimmed
                                                              FROM DUAL;
SQL> SELECT 'SSMITHSS' String, TRIM(BOTH 'S' FROM 'SSMITHSS') Trimmed
                                                              FROM DUAL;
```

REPLACE Function

- It Returns the Every Occurrence of Search String Replaced by The Replacement String.
- If the Replacement String is Omitted or NULL, All Occurrences of Search String are Removed.
- It Substitutes One String for Another as Well as Removes Character Strings.

Syntax: REPLACE(Char, Search_Str, Replace_Str)

```
SQL> SELECT 'JACK AND JUE' String, REPLACE('JACK AND JUE' , 'J', 'BL') Replaced
                                                              FROM DUAL;
SQL> SELECT Ename, REPLACE(JOB, 'MAN', 'DAM') Replaced FROM Emp
                                                    WHERE Job = 'MANAGER';
SQL> SELECT Job, REPLACE (Job, 'P') FROM Emp  WHERE Job = 'PRESIDENT';
SQL> SELECT Job,  REPLACE (Job, 'MAN', 'EXECUTIVE') FROM Emp
                                                  WHERE Job = 'SALESMAN';
```

TRANSLATE Function

- Used to Translate Character by Character in a String.

Syntax: TRANSLATE(char, From, To)

- It Returns a CHAR With All Occurrences of Each Character in 'From' Replaced By Its Corresponding Character in 'To'.
- Characters in Char That Are Not in FROM Are not Replaced.
- The Argument FROM Can Contain More Characters Than TO.
- If The Extra Characters Appear in CHAR, They are Removed From the Return Value.

```
SQL> SELECT  Job, TRANSLATE(Job, 'P', ' ')  FROM Emp  WHERE Job = 'PRESIDENT';
SQL> SELECT Job, TRANSLATE(Job, 'MN', 'DM')  FROM Emp WHERE Job = 'MANAGER';
SQL> SELECT Job,  TRANSLATE(Job, 'A', 'O')  FROM Emp WHERE Job = 'SALESMAN';
```

CHR Function

- It Returns a Character Having the ASCII Equivalent to 'n'.
- It Returns the Equivalent For 'n' in Database Character Set or National Character Set.

Syntax:

CHR(n)
CHR(n USING NCHAR_CS)

```
SQL> SELECT CHR(67)||CHR(65)||CHR(84) Sample FROM DUAL;
SQL> SELECT CHR(16705 Using NCHAR_CS) FROM DUAL;
```

ASCII Function

- It Returns The ASCII Representation in the Character Database Set of The First Characters of the CHAR.

Syntax: **ASCII(Char)**

```
SQL> SELECT ASCII('A'), ASCII('APPLE') FROM DUAL;
```

NUMBER Functions

- These Functions Accept Numeric Input and Return Numeric Values as Output.
- Many Functions Return Values That are Accurate to 38 Decimal Digits.

ROUND Function

Syntax: ROUND(n, m)

- It Returns 'n' Rounded to 'm' Places Right of The Decimal Point.
- If 'm' is Omitted, 'n' is Rounded to 0, Places.
- 'm' can be Negative, and Rounds off The Digits to The Left of The Decimal Point.
- 'm' Must be an Integer.

```
SQL> SELECT 15.193 Num1, ROUND(15.193, 1) Rounded FROM DUAL;
SQL> SELECT 15.193 Num1, ROUND(15.193, -1) Rounded FROM DUAL;
SQL> SELECT 45.923 Num1, ROUND(45.923, 2) Rounded, ROUND(45.923, 0) Rounded,
                         ROUND(45.923, -1) Rounded FROM DUAL;
```

TRUNCATE Function

Syntax: TRUNC(n, m)

- It Returns 'n' Truncated to 'm' Decimal Places.
- If 'm' is Omitted, 'n' is Truncated to 0 Decimal Places.
- 'n' Can be Negative to Truncate 'm' Digits Left of The Decimal Point.

```
SQL> SELECT 15.79 Num1, TRUNC(15.79, 1) Truncated FROM DUAL;
SQL> SELECT 15.79 Num1, TRUNC(15.79, -1) Truncated FROM DUAL;
SQL> SELECT 15.79 Num1, TRUNC(45.923, 2) Truncated, TRUNC(42.923) Truncated,
                        TRUNC(45.923, -1) Truncated FROM DUAL;
```

CEIL Function

Syntax: CEIL(n)

- Returns the Smallest Integer Greater Than or Equal to 'n'.
- The Adjustment is Done to The Highest Nearest Decimal Value.

```
SQL> SELECT 15.7 Num1, CEIL(15.7) Ceiled FROM DUAL;
SQL> SELECT 14.27 Num1, CEIL(14.27) Ceiled, CEIL(14.2) Ceiled, CEIL(14) Ceiled
                                                        FROM DUAL;
```

FLOOR Function

Syntax: FLOOR(n)

- Returns the Largest Integer Less Than or Equal Than 'n'.
- The Adjustment is Done to The Lowest Nearest Decimal Values.

```
SQL> SELECT 15.7 Num1, FLOOR(15.7) Floor FROM DUAL;
SQL> SELECT 14.27 Num1, FLOOR(14.27) Floor, FLOOR(14.2) Floor, FLOOR(14) Floor
                                                        FROM DUAL;
```

MODULUS Function

Syntax: MOD(m, n)

- It Returns Remainder of 'm' Divided by 'n'.
- It Returns 'm' if 'n' is 0.

```
SQL> SELECT MOD(11, 4) Modulus, MOD(10, 2) Modulus FROM DUAL;
```

POWER Function

Syntax: POWER(m, n)

- Returns 'm' Raised to The 'n'th Power.
- The Base 'm' and The Exponent 'n' Can be Any Numbers.
- If 'm' is Negative, 'n' Must be an Integer.

```
SQL> SELECT POWER(3, 2) Power, POWER(-3, 2) Power FROM DUAL;
SQL> SELECT POWER(3, -2) Power, POWER(-3, -2) Power FROM DUAL;
SQL> SELECT POWER(-3.5, -2) Power, POWER(3, -2.5) Power FROM DUAL;
SQL> SELECT POWER(-3.5, 2.5) Power, POWER(3.5, -2.5) Power FROM DUAL;
```

SQUARE ROOT Function

Syntax: SQRT(n)

- It Returns Square Root of 'n' as Real Value.
- The Value of 'n' Cannot be Negative.

```
SQL> SELECT SQRT(25) FROM DUAL;
```

ABSOLUTE Function

Syntax: ABS(n)

- It Returns the Absolute Value of 'n'.

SQL> SELECT ABS(-15) FROM Dual;

SQL> SELECT Sal, Comm, Sal – Comm, ABS(Sal – Comm) FROM Emp WHERE Comm = 1400;

SIGN Function

Syntax: SIGN(n)

- It Returns the SIGN, Specification of a Number.
  - If n < 0 , Returns –1
  - If n = 0 , Returns 0
  - If n > 0, Returns 1

SQL> SELECT SIGN(-15), SIGN(15), SIGN(0) FROM DUAL;

SQL> SELECT Sal, Comm, SIGN(Sal – Comm) FROM Emp WHERE SIGN(Sal – Comm) = -1;

Working With Dates

- Oracle Stores Dates in an Internal Numeric Format.
- The Dates in Oracle Range From JANUARY 1, 4712 BC to DECEMBER 31, 9999 AD.
- The Default Display and Input Format for any Date is DD-MON-YY.
- The Internal Date Format Represents
  - Century ← Year ← Month ← Day ← Hours ← Minutes ← Seconds

SYSDATE:

- It is a Date Function That Returns Current DATE and TIME.
- SYSDATE is Generally Selected Upon a DUMMY Table.

SQL> SELECT SYSDATE FROM DUAL;

Date Arithmetic

- As Database Stores Dates as Numbers, Arithmetic Operations can be Implemented.
- Number Constants Can be Added or Subtracted Upon Dates.
- The Operations That Can be Applied are
  - Date + Number → Returns Date
    - Adds Number of Days to a Date.
  - Date - Number → Returns Date.
    - Subtracts Number of Days From a Date.
  - Date – Date → Returns Number of Days.
    - Subtracts One Date from Another Date.
  - Date + Number/24 → Returns Date.
    - Adds Number of Hours to a Date.

SQL> SELECT SYSDATE, SYSDATE + 3 FROM DUAL;

SQL> SELECT SYSDATE, SYSDATE – 3, SYSDATE + 72 / 24 FROM DUAL;

SQL> SELECT Ename, Hiredate, Hiredate + 3 FROM Emp;

SQL> SELECT Ename, Hiredate, Hiredate – 3 FROM Emp;

SQL> SELECT Ename, Hidrdate, SYSDATE – Hiredate FROM Emp;

SQL> SELECT Ename, (SYSDATE – Hiredate) / 7 Weeks FROM Emp WHERE Deptno = 10;

DATE Functions

ADD_MONTHS Function

Syntax: ADD_MONTHS(D, n)

- It Returns the Date 'D' Plus OR Minus 'n' Months
- The Argument 'n' Can be Any Positive OR Negative Integer.

SQL> SELECT SYSDATE, ADD_MONTHS(SYSDATE, 2) FROM DUAL;

SQL> SELECT Sal, Hiredate, ADD_MONTHS(Hiredate, 2) FROM Emp WHERE Deptno = 20;

MONTHS_BETWEEN Function

Syntax: Months_Between(D1, D2)

- It Returns Number of Months Between Dates 'd1' and 'd2'.

- If 'd1' is Later Than 'd2', The Result is Positive, else Negative.
- If 'd1' and 'd2' are Either the Same Days of The Months or Both Last Days of The Months, The Result is Always An Integer.

```
SQL> SELECT Ename, HireDate, SYSDATE, MONTHS_BETWEEN(SYSDATE, Hiredate)
                                                        FROM Emp;
SQL> SELECT Empno, Hiredate, MONTHS_BETWEEN(SYSDATE , Hiredate) FROM Emp
       WHERE MONTHS_BETWEEN(SYSDATE, Hiredate) < 200;
```

Next_Day Function

Syntax: NEXT_DAY(d, Char)

- It Returns The Date of The First Week Day Named By CHAR, That is Later Than the Date 'd'.
- The CHAR Must be a Day of The Week in the Sessions Date Language.
- The Day of The Week Can Be Full Name or The Abbreviation.

```
SQL> SELECT SYSDATE, NEXT_DAY(SYSDATE, 'WED') FROM DUAL;
SQL> SELECT Sal, Hiredate, NEXT_DAY(Hiredate, 'MONDAY')FROM Emp;
```

LAST_DAY Function

Syntax: LAST_DAY(D)

- It Returns The Date of The Last Day of The Month That Contains 'D'.
- Mostly Used to Determine How Many Days Are Left in the Current Month .

```
SQL> SELECT SYSDATE, LAST_DAY(SYSDATE) LastDay FROM DUAL;
SQL> SELECT LAST_DAY(SYSDATE) Last, SYSDATE,
              LAST_DAY(SYSDATE) – SYSDATE Daysleft FROM DUAL;
```

Rounding of Dates

Syntax: ROUND(Date, 'Format')

- Returns Date Rounded to The Unit Specified by The Format.
- If Format is Omitted, Date is Rounded to the Nearest Day.

```
SQL> SELECT SYSDATE, ROUND(SYSDATE , 'DAY') FROM DUAL;
SQL> SELECT SYSDATE, ROUND(SYSDATE, 'MONTH') FROM DUAL;
SQL> SELECT SYSDATE, ROUND(SYSDATE , 'YEAR') FROM DUAL;
```

Truncating Dates

Syntax: TRUNC(Date, 'Format')

- Date is Truncated to the Nearest Date with the Time Portion of the Day Truncated to the Specified Unit.
- If Format is Omitted Date is Truncated to the Nearest Day.

```
SQL> SELECT SYSDATE, TRUNC(SYSDATE, 'DAY')  FROM DUAL;
SQL> SELECT SYSDATE, TRUNC(SYSDATE, 'MONTH') FROM DUAL;
SQL> SELECT SYSDATE, TRUNC(SYSDATE, 'YEAR')  FROM DUAL;
```

Conversion Functions

- The Conversion Functions Convert a Value From One Data Type to Another.
- The Data Type Conversion in Oracle is of Two Types
  - Implicit Data Type Conversion
  - Explicit Data Type Conversion

Implicit Data type Conversion

- Implicit Data type Conversions work According to The Convention Specified by Oracle.
- The Assignment Succeeds if the Oracle Server Can Convert the Data Type of The Value.
- CHAR to NUMBER Conversions Succeed Only if The Character Strings Represent a Valid NUMBER.
- CHAR to DATE Conversions Succeed Only if the Character Strings Represent the Default Format of Date DD-MON-YY.

In Assignment Operation

- VARCHAR2 / CHAR → NUMBER
- VARCHAR2 / CHAR → DATE
- NUMBER → VARCHAR2
- DATE → VARCHAR2

In Expressions

- VARCHAR2/ CHAR → Number
- VARCHAR2 / CHAR → Date

Explicit Data type Conversion

- SQL provides three functions to convert a value from one data type to another.
- The Explicit Conversion Functions are
  - TO_CHAR → To Character Conversion.
  - TO_DATE → To Date Conversion
  - TO_NUMBER → To Number Conversion.

TO_CHAR Conversion Function

- This Function Can be Used in Two Different Flavors.
  - TO_CHAR(Number Conversion)
  - TO_CHAR(Date Conversion)

TO_CHAR(Number Conversion)

Syntax: TO_CHAR(NUMBER , fmt, 'nlsparms')

- Converts NUMBER of Number data Type to a Value of VARCHAR2 data Type.
- 'fmt' is The Optional Number Format, That Can be Used.
- The 'nlsparms' Specifies the Characters Returned by the Number Format Element

TO_CHAR(Date Conversion)

Syntax: TO_CHAR(DATE, fmt, 'nlsparams')

- Converts Date of DATE Data Type to a Value of VARCHAR2 Data Type in The Format Specified.
- 'fmt' is the Optional Date Format, That Can be Used.
- The 'nlsparams' Specifies the Language in Which Month and Day Names And Abbreviations are Returned.

Working With Number Format Models

Format Model

- A Format Model is a Character Literal That Describes the FORMAT of DATE or NUMBER Data Stored in a CHARACTER STRING.

Decimal Indicator: D → 99D99

- It Returns the Specified Position of The Decimal Character.
- The Default Decimal delimiter is Period '.'
- Only One Decimal Indicator Can Be Specified in a Number Format Model.

```
SQL> SELECT 1234, TO_CHAR(1234, '9999D99') FROM DUAL;
SQL> SELECT 1234, TO_CHAR(1234, '999D99') FROM DUAL;
```

Scientific Notation Indicator: EEEE → 9.9EEEE

- Returns a Numeric Value Using Scientific Notation.

```
SQL> SELECT  1234,  TO_CHAR(1234 , '9.9EEEE'),  TO_CHAR(1234,  '9.99EEEE')
                                                                FROM DUAL;
```

Group Separator: G → 9G999

- Returns the Specified Position of The Group Separator
- Multiple Group Separators Can Be Specified.

```
SQL> SELECT 1234567, TO_CHAR(1234567, '99G99G999') FROM DUAL;
SQL> SELECT Sal, TO_CHAR (Sal, '99G999D99') FROM Emp;
```

Locale Currency Indicator: L → L999 OR 999L

- Returns the Specified Position of The Locale Currency Symbol.

```
SQL> SELECT 1234, TO_CHAR(1234, 'L9999') FROM DUAL;
SQL> SELECT Sal, TO_CHAR(Sal, 'L999999') Currency  FROM Emp  WHERE Deptno = 10;
SQL> SELECT Sal, TO_CHAR(Sal, 'L99G999D99', 'NLS_CURRENCY = IndRupees') Sal
                                                    FROM Emp WHERE Deptno = 20;
```

Trailing Minus Indicator: MI → 9999MI

- Returns Negative Value With a Trailing Minus Sign '-'.
- Returns Positive Value With a Trailing Blank.

- 'MI' Format Should be Declared as Trailing Argument Only.

SQL> SELECT -10000, TO_CHAR(-10000, 'L99G999D99MI') FROM DUAL;
SQL> SELECT Sal, Comm, Comm, Sal, TO_CHAR(Comm – Sal , 'L99999MI') FROM Emp
WHERE Deptno IN(10, 20, 30);

Negative Number Indicator: PR → 9999PR

- Returns Negative Number in '<>'.
- It Can Appear Only as Trailing Declaration.

SQL> SELECT TO_CHAR(-1000, 'L99G999D99PR') FROM DUAL;
SQL> SELECT Sal, Comm, Comm – Sal, TO_CHAR(Comm – Sal, 'L99999PR') FROM Emp;

Roman Number Indicator: RN OR rn

- RN → Returns Upper Roman Number.
- rn → Returns Lower Roman Number.
- The Value can be an Integer Between 1 and 3999.

SQL> SELECT 1000, TO_CHAR(1000 , 'RN'),TO_CHAR(1000, 'rn') FROM DUAL;

Sign Indicator: S → S99999 OR 99999S

- Returns Negative Value With a Leading Minus Sign.
- Returns Positive Value With a Leading Plus Sign.
- Returns Negative Value With Trailing Minus Sign.
- Returns Positive Value With a Trailing Plus Sign.
- 'S' Can Appear as First or Last Value.

SQL> SELECT 1000, TO_CHAR(1000, 'S9999'), TO_CHAR(-1000, 'S9999') FROM DUAL;
SQL> SELECT TO_CHAR(1000, '9999S'), TO_CHAR(-1000, '9999S') FROM DUAL;
SQL> SELECT Sal, TO_CHAR(Sal, 'S99999'), TO_CHAR(Sal, '99999S') FROM Emp;
SQL> SELECT Sal, Comm, TO_CHAR(Comm - Sal, 'S99999'),TO_CHAR(Comm – Sal, '99999S')
FROM Emp;

Hexadecimal Indicator: X → XXXX

- Returns the Hexadecimal Value of The Specified Number of Digits.
- If the Number is Not an Integer, Oracle Rounds it to An Integer.
- Accepts Only Positive Values OR 0.

SQL> SELECT 1000, TO_CHAR(1000, 'XXXX') FROM DUAL;
SQL> SELECT Ename, Sal, TO_CHAR(Sal, 'XXXXX') HexSal FROM Emp;

Group Separator: , → 9,999

- Returns a Comma in the Specified Position.
- Multiple Commas Can be Specified.

SQL> SELECT 10000, TO_CHAR(10000, '99,999.99') FROM DUAL;
SQL> SELECT Ename, Sal, TO_CHAR(Sal, '99,999.99') FROM Emp;

Decimal Indicator: . → 99.99

- Returns a Decimal Point, At the Specified Position.
- Only one Period Can be Specified in a Number Format Model.

SQL> SELECT 10000, TO_CHAR(10000, 'L99,999.99') FROM DUAL;
SQL> SELECT Ename, Sal, TO_CHAR(Sal, 'L99,999.99') FROM Emp;

Dollar Indicator: $ → $9999

- Returns Value With a Leading Dollar Sign.

SQL> SELECT 10000, TO_CHAR(10000, '$99,999.99') FROM DUAL;
SQL> SELECT Ename, Sal, TO_CHAR(Sal, '$99,999.99') FROM Emp;

Zero Indicator: 0 → 0999 OR 9990

- Returns leading OR Trailing Zeros.

SQL> SELECT 10000, TO_CHAR(1000, '0999999'), TO_CHAR(1000, '09999990') FROM DUAL;
SQL> SELECT Ename, Sal, TO_CHAR(Sal, '$099,999.99') FROM Emp;

Digit Place Marker: 9 → 9999

- Returns Value With a Specified Number of Digits With a Leading Space When Positive or Leading Minus When Negative.

SQL> SELECT 1000, 600, TO_CHAR(1000 – 600 , '99999'), TO_CHAR(600 – 1000 , '99999')

FROM DUAL;

SQL> SELECT 20.25, 20, TO_CHAR(20.25 – 20, '99999') FROM DUAL;

ISO Currency Indicator: C → C9999

- Returns Specified Position of The ISO Currency Symbol.

SQL> SELECT 1000, TO_CHAR(1000, 'C9999.99')FROM DUAL;
SQL> SELECT Ename, Sal, TO_CHAR(Sal, 'C9999.99') FROM Emp;

Date Format Models:

- The Date Format Models Can be Used in The TO_CHAR Function to Translate a DATE Value From Original Format to User Format.
- The Total Length of a Date Format Model Cannot Exceed 22 Characters.

Date Format Elements

- A Date Format Model is Composed of One or More Date Format Elements.
- For Input Format Models, Format Items Cannot Appear Twice, and Format Items That Represent Similar Information Cannot be Combined.
- Capitalization in a Spelled Word, Abbreviation, or Roman Numeral Follows Capitalization in the Corresponding Format Element.
- Punctuation Such as Hyphens, Slashes, Commas, Periods and Colons.

AD or A.D./BC or B.C. Indicator

- Indicates AD/BC With OR Without Periods.

SQL> SELECT SYSDATE, TO_CHAR(SYSDATE, 'AD') FROM DUAL;
SQL> SELECT TO_CHAR(SYSDATE, 'B.C.'), TO_CHAR(SYSDATE, 'A.D.') FROM DUAL;
SQL> SELECT Ename, Sal, Hiredate, TO_CHAR(Hiredate, 'A.D.') FROM Emp;

Meridian Indicator: AM OR A.M./PM OR P.M.

- It Indicates Meridian Indicator With or Without Periods.

SQL> SELECT SYSDATE, TO_CHAR(SYSDATE, 'A.M.'), TO_CHAR(SYSDATE, 'PM')
FROM DUAL;
SQL> SELECT Ename, Sal, Hiredate, TO_CHAR(Hiredate, 'AM') FROM Emp;

Century Indicator: CC/SCC

- Indicates The Century , S Prefixes BC Date With '–'.

SQL> SELECT SYSDATE, TO_CHAR(SYSDATE, 'CC - AD') FROM DUAL;
SQL> SELECT Sal, Hiredate,TO_CHAR(Hiredate , 'SCC - AD') FROM Emp;

Numeric Week Day Indicator: D → (1 – 7)

- Returns The Week Day Number

SQL> SELECT SYSDATE, TO_CHAR(SYSDATE, 'D') FROM DUAL;
SQL> SELECT Ename, Hiredate, TO_CHAR(Hiredate, 'D') FROM Emp;

Week Day Spelling Indicator: → Day.

- Pads to a Length of 9 Characters.

SQL> SELECT SYSDATE, TO_CHAR(SYSDATE, 'DAY') FROM DUAL;
SQL> SELECT Sal, Hiredate, TO_CHAR(Hiredate, 'DAY') FROM Emp
WHERE TO_CHAR(Hiredate, 'DAY') = 'WEDNESDAY';

Month Day Indicator: DD

- It Indicates the Day of the Month(1– 31)

SQL> SELECT SYSDATE, TO_CHAR(SYSDATE, 'DD - DAY') FROM DUAL;
SQL> SELECT Hiredate, TO_CHAR(HireDate, 'DD - DAY') FROM Emp;
SQL> SELECT HireDate, TO_CHAR(HireDate, 'DD - DAY') FROM Emp WHERE
TO_CHAR(HireDate, 'DD - DAY') = '03 - WEDNESDAY';

Year Day Indicator: DDD

- It Indicates the Day of the Year(1 – 366)

SQL> SELECT SYSDATE, TO_CHAR(SYSDATE, 'DDD') FROM DUAL;
SQL> SELECT Ename, HireDate , TO_CHAR(HireDate, 'DDD') FROM Emp
WHERE TO_CHAR(HireDate, 'DAY') = 'WEDNESDAY';

Abbreviated Week Day: DY

- It Indicates The Abbreviated Name of The Week Day.

SQL> SELECT SYSDATE, TO_CHAR(SYSDATE, 'D-DY-DAY') FROM DUAL;
SQL> SELECT Ename, HireDate, TO_CHAR(HireDate, 'D-DY-DAY') FROM Emp
WHERE Deptno IN(10, 20);

ISO Standard Year Week Indicator: IW

- Specifies the Week of The Year (1 - 52 or 1 – 53) Based on The ISO Standard.

SQL> SELECT SYSDATE, TO_CHAR (SYSDATE, 'IW') FROM DUAL;
SQL> SELECT Ename, HireDate, TO_CHAR(HireDate, 'IW') FROM Emp;

ISO Standard 4 Digit Year Indicator: IYYY

- Specifies 4 Digit Year Based on The ISO Standard.
- It can Even be Used in Combination of IYY, IY, I.

SQL> SELECT SYSDATE, TO_CHAR(SYSDATE, 'IYYY') FROM DUAL;
SQL> SELECT Ename, HireDate, TO_CHAR(HIREDATE, 'IYYY')FROM Emp
WHERE TO_CHAR(HireDate, 'DAY') = 'WEDNESDAY';

Four Digit Year Indicator: YYYY OR SYYYY

- Returns Four Digit Year, 'S' Prefixes BC Dates With '–'.
- It Can Even be Used in Combination of YYY OR YY OR Y.
- Y,YYY Returns Year With Comma in That Position.

SQL> SELECT SYSDATE, TO_CHAR(SYSDATE, 'YYYY') Four,
TO_CHAR(SYSDATE, 'YYY') Three FROM DUAL;
SQL> SELECT Ename, HireDate, TO_CHAR(HireDate, 'YYYY') FROM Emp
WHERE Deptno = 20 ;

Spelled Year Indicator: YEAR OR SYEAR

- Returns the Numerical Year in Spelling.
- 'S' Prefixes BC Dates With '–'.

SQL> SELECT SYSDATE, TO_CHAR(SYSDATE, 'YEAR') FROM DUAL;
SQL> SELECT Ename, HireDate, TO_CHAR(HireDate, 'YEAR') FROM Emp;

Week of the Month Indicator: W

- Specifies the Week of the Month(1 – 5).
- Week Starts on The First Day of The Month and Ends on the Seventh Day.

SQL> SELECT SYSDATE, TO_CHAR(SYSDATE, 'W') FROM DUAL;
SQL> SELECT Ename, HireDate, TO_CHAR(HireDate, 'W') FROM Emp;

Year Week Indicator: WW

- Specifies the Week of the Year (1 – 53).
- Week 1 Starts on the First Day of the Year and Continues to the Seventh Day in That Year.

SQL> SELECT SYSDATE, TO_CHAR(SYSDATE, 'WW') FROM DUAL;
SQL> SELECT Ename, HireDate, TO_CHAR(HireDate, 'WW') FROM Emp;

Quarter of the Year Indicator: Q

- Returns the Quarter of The Year.
- Quarter Starting With the Month of January and Ending With Every Three Months.

SQL> SELECT SYSDATE, TO_CHAR(SYSDATE, 'Q') FROM DUAL;
SQL> SELECT Ename, HireDate, TO_CHAR(HireDate, 'Q') FROM Emp
WHERE TO_CHAR(HireDate, 'Q') = 4;

Julian Day Indicator: J

- Returns the JULIAN DAY of the Given Date.
- It is The Number of Days Since January 1, 4712 BC.
- Numbers Specified With 'J' Must be Integers.

SQL> SELECT SYSDATE,TO_CHAR(SYSDATE, 'J') FROM DUAL;
SQL> SELECT Ename, TO_CHAR(HireDate, 'J-DDD-DD-D')FROM Emp;

Numeric Month Indicator: MM

- Returns the Numeric Abbreviation of the Month.

SQL> SELECT SYSDATE, TO_CHAR(SYSDATE, 'MM-YYYY') FROM DUAL;
SQL> SELECT Ename, HireDate, TO_CHAR(HireDate, 'DD-MM-YYYY') FROM Emp
WHERE TO_CHAR(HireDate , 'MM') = 12

Abbreviated Month Indicator: MON

- Returns the Abbreviated Name of The Month.

```
SQL> SELECT SYSDATE, TO_CHAR(SYSDATE, 'MM-MON') FROM DUAL;
```

Month Spelling Indicator: MONTH

- Spells the Name of the Month, Padded to a Length of 9 Characters.

```
SQL> SELECT SYSDATE, TO_CHAR(SYSDATE, 'MON-MONTH') FROM DUAL;
SQL> SELECT Ename, HireDate,TO_CHAR(HireDate, 'MONTH, YYYY') FROM Emp;
```

Twelve Hour Clock Mode: HH OR HH12

- Returns the Hour of The Day in Twelve Hour Clock Mode.

```
SQL> SELECT SYSDATE, TO_CHAR(SYSDATE, 'HH'), TO_CHAR(SYSDATE, 'HH12, AM')
                                                        FROM DUAL;
SQL> SELECT Ename, HireDate , TO_CHAR(HireDate, 'HH12 : AM') FROM Emp;
```

Twenty Hour Clock Mode: HH24

- Returns the Hour of the Day in Twenty Four Hour Clock Mode.(0 – 23)

```
SQL> SELECT SYSDATE,TO_CHAR(SYSDATE, 'HH24') FROM DUAL;
```

Minutes Indicator: MI

- Returns the Minutes From The Given Date(0 – 53).

```
SQL> SELECT SYSDATE, TO_CHAR(SYSDATE, 'MI'), TO_CHAR(SYSDATE, 'HH:MI')
                                                        FROM DUAL;
SQL> SELECT Ename, Sal, TO_CHAR(HireDate, 'HH:MI') FROM Emp WHERE Job = 'CLERK';
```

Roman Month Indicator: RM

- Returns the Roman Numeral Month (I–XII).

```
SQL> SELECT SYSDATE,  TO_CHAR(SYSDATE, 'RM'), TO_CHAR(SYSDATE, 'DD-RM-YY')
                                                        FROM DUAL;
SQL> SELECT Ename, Sal, TO_CHAR(HireDate, 'DD-RM-YY') FROM Emp;
```

Seconds Indicator: SS

- Returns Seconds From the Given Date(0–59).

```
SQL> SELECT SYSDATE, TO_CHAR(SYSDATE, 'SS'), TO_CHAR(SYSDATE, 'HH:MI:SS')
                                                        FROM DUAL;
SQL> SELECT SYSDATE, TO_CHAR(SYSDATE, 'DD-MONTH-YYYY, HH:MI:SS A.M.')
                                                        FROM DUAL;
SQL> SELECT Ename, Sal, HireDate, TO_CHAR(HireDate, 'HH24:MI:SS') FROM Emp
                                                WHERE Deptno IN(10, 30);
```

Seconds Past Mid Night: SSSSS

- Display Seconds past Midnight(0–86399).

```
SQL> SELECT SYSDATE, TO_CHAR(SYSDATE, 'SSSSS')
        FROM DUAL;
```

Date Format Punctuators:

- The Punctuation Marks That Can be Used in Date Formats Are…
  '-', '/', '!', '.', ';', ':', "text"

Date Format Element Suffixes: TH OR SP

- TH → Suffixes the Ordinal Number With 'ST' OR 'ND' OR 'RD' OR 'TH'.
  Example: DDTH → 20TH.

```
SQL> SELECT SYSDATE,  TO_CHAR(SYSDATE, 'DDTH, MONTH,YYYY') FROM DUAL;
SQL> SELECT Ename, Sal, HireDate, TO_CHAR(HireDate, 'DDTH, MONTH, YYYY')
                                                        FROM Emp;
```

- SP → Spells Ordinal Numbers.
  Example: DDSP → TWENTY

```
SQL> SELECT SYSDATE, TO_CHAR(SYSDATE, 'DDSP , Month, YYY') FROM DUAL;
SQL> SELECT Ename, Sal, TO_CHAR(HireDate, 'DDSP, Month, YYYY')  FROM Emp;
SQL> SELECT SYSDATE, TO_CHAR(SYSDATE, 'DDSPTH, Month, YYYY')  FROM DUAL;
SQL> SELECT Ename, Sal, TO_CHAR(HIREDATE, 'DDSPTH, Month, YYYY')  FROM Emp;
SQL> SELECT SYSDATE, TO_CHAR(SYSDATE, 'DDSPTH Month YYYYSP') FROM DUAL;
```

Date Format Elements Restrictions:

- The Suffixes When Added to Date Return Values Always in English.
- Date Suffixes Are Valid Only on Output, Hence Cannot be Used to Insert a Date Into the Database.

Format Model Modifiers:

Fill Mode Indicator: FM

- It Suppresses Blank Padding in The Return Value of The TO_CHAR Function.

Format Exact: FX

- It Specifies Exact Matching For The Character Argument And Date Format Model.

```
SQL> SELECT SYSDATE, TO_CHAR(SYSDATE, 'DDSPTH MONTH YYYYSP'),
                        TO_CHAR(SYSDATE, 'FMDDSPTH MONTH YYYYSP')
                                                        FROM DUAL;
```

TO_NUMBER Function

Syntax: TO_NUMBER(Char, fmt, 'nlsparam')

- It Converts a Char, Value of CHAR or VARCHAR2 Data Type Containing a NUMBER in the Format Specified By The Optional Format Model 'fmt', To a Value of NUMBER Data Type.

```
SQL> SELECT '$10,000.00', TO_NUMBER('$10,000.00', 'L99,999.99') FROM DUAL;
SQL> SELECT '$1,000.00', TO_NUMBER('$1,000.00', 'L9,999.99') + 500 FROM DUAL;
```

TO_DATE Function

Syntax: TO_DATE(Char, fmt, 'nlsparam')

- Converts Given Char of CHAR or VARCHAR2 Data Type to a Value of DATE Data Type.
- The 'fmt' is an Optional Date Format Specifying the Format of CHAR.

```
SQL> SELECT Ename, HireDate, ADD_MONTHS(TO_DATE
        ('17-DEC-1980','DD-MON-YY' ), 3) FROM Emp WHERE HireDate = '17-DEC-1980';
SQL> SELECT Ename, HireDate, ADD_MONTHS(TO_DATE('1980-DECEMBER-17',
                'YYYY-MONTH-DD'), 3) FROM Emp WHERE HireDate = '17-DEC-1980';
SQL> SELECT Ename, HireDate, ADD_MONTHS( TO_DATE('1980-DECEMBER-17',
        'YYYY-MONTH-DD'), 3) FROM Emp WHERE  TO_CHAR(HireDate, 'FMYYYY-
                                                MONTH-DD') = '1980-DECEMBER-17';
SQL> SELECT '12-August-2007', TO_DATE('12-August-2007', 'DD-Month-YYYY') + 3
                                                        FROM DUAL;
```

Let Us Revisit INSERT Statement Once Again

Working With Invalid Numbers

```
SQL> CREATE TABLE SampIns
        ( SampleNum NUMBER(6),
          SampleDate DATE );
SQL> INSERT INTO SampIns(SampleNum, SampleDate) VALUES
                    (TO_NUMBER('1,23,456', '9G99G999'), SYSDATE );
SQL> INSERT INTO SampIns(SampleNum, SampleDate) VALUES
                    (TO_NUMBER('1,23,457-', '9G99G999MI'), SYSDATE );
```

Working With Invalid Dates

```
SQL> INSERT INTO SampIns(SampleNum, SampleDate) VALUES
                    (123458, '02-AUG-07');
SQL> INSERT INTO SampIns(SampleNum, SampleDate) VALUES( 123459, SYSDATE);
SQL> INSERT INTO SampIns(SampleNum, SampleDate) VALUES (123460, TO_DATE
                ('02-August-2007, 06:45:36 P.M.', 'DD-Month-YYYY, HH:MI:SS P.M.'));
```

Inserting B.C. Dates Into Tables

```
SQL> INSERT INTO SampIns(SampleNum, SampleDate) VALUES (123461, '02-AUG-2007');
SQL> INSERT INTO SampIns(SampleNum, SampleDate) VALUES(123462, TO_DATE
            ('02-AUG-2007 B.C.', 'DD-MON-YYYY B.C.'));
SQL> INSERT INTO SampIns(SampleNum,SampleDate)VALUES(123463,
                        TO_DATE(988216, 'J'));
```

Watch These Inserts

```
SQL> INSERT INTO SampIns(SampleNum, SampleDate) VALUES( 123464, '02-AUG-07');
```

```
SQL> INSERT INTO SampIns(SampleNum, SampleDate) VALUES(123465, '02-AUG-2007');
SQL> INSERT INTO SampIns(SampleNum, SampleDate) VALUES(123466, '02-AUG-97');
SQL> INSERT INTO SampIns(SampleNum, SampleDate) VALUES(123467, '02-AUG-1997');
```

RR Date Format Element

- The RR Date Format Element is Similar to The YY Date Format Element.
- The RR Format Element Provides Additional Flexibility For Storing Date Values in Other Centuries.
- The RR Date Format Element Allows to Store The Date to The Previous as Well as The Next Centuries.
- The RR Format Element Should be Used in Association With TO_DATE Conversion Function Only, Else The System Treats it as YY Format Element.
- The Format Can Be Used As Either 'RR' OR 'RRRR' Format Element.

Chart of Identification

Last Two Digits of The Year Managed By Clock		0 To 49	50 To 99
Last Two Digits of The Year You Supplied	0 To 49	Returns The Date in Current Century	Returns The Date in The Next Century
	50 To 99	Returns The Date in The Preceding Century	Returns The Date in Current Century

Let us See Some Combinations

```
SQL> SELECT  TO_CHAR( ADD_MONTHS(Hiredate, 1), 'DD-MON-YYYY' ) "Next Month"
                                        FROM Emp  WHERE Ename = 'SMITH';
SQL> SELECT CONCAT(CONCAT(Ename, ' is a '), Job) Designation FROM Emp
                                                WHERE Empno = 7900;
SQL> SELECT TRUNC(TO_DATE('27-OCT-92', 'DD-MON-YY'), 'YEAR') 'New Year"
                                                FROM DUAL;
SQL> SELECT TO_CHAR(ADD_MONTHS(LAST_DAY(Hiredate) , 5), 'DD-MON-YYYY')
                        "FIVE MONTHS" FROM Emp WHERE Ename = 'MARTIN';
SQL> SELECT MONTHS_BETWEEN(TO_DATE('02-02-1995', 'MM-DD-YYYY'),
                TO_DATE('01-01-1995',  'MM-DD-YYYY')) MONTHS FROM DUAL;
SQL> SELECT NEXT_DAY('15-MAR-98', 'TUESDAY') "Next Day" FROM DUAL;
SQL> SELECT Ename, NVL(TO_CHAR(Comm), 'Not Applicable') "Commission" FROM Emp
                                                WHERE Deptno = 30;
SQL> SELECT ROUND(TO_DATE('27-OCT-92' , 'DD-MON-YY'),  'YEAR') "New Year"
                                                FROM DUAL;
SQL> SELECT TO_CHAR(HireDate, 'MONTH DD, YYYY') FROM Emp  WHERE
                                                Ename = 'BLAKE';
SQL> SELECT Ename, TO_CHAR(HireDate, 'FMMonth, DD YYYY') HireDate FROM Emp
                                                WHERE Deptno = 20;
SQL> SELECT TO_CHAR(TO_DATE('27-OCT-98', 'DD-MON-RR'), 'YYYY') YEAR
                                                FROM DUAL;
SQL> SELECT TO_CHAR(TO_DATE('27-OCT-17', 'DD-MON-RR'),'YYYY') Year
                                                FROM DUAL;
```

Assumption: Queries Are Issued Between Year 1950 – 1999.

```
SQL> SELECT TO_CHAR(TO_DATE('27-OCT-98', 'DD-MON-RR'),  'YYYY' ) Year
                                                FROM DUAL;
SQL> SELECT TO_CHAR(TO_DATE('27-OCT-17', 'DD-MON-RR'),  'YYYY') Year
                                                FROM DUAL;
```

Assumption: Queries Are Issued Between Year 2000 – 2049

```
SQL> SELECT TO_CHAR(SYSDATE, 'FMDDTH')||' of '|| TO_CHAR( SYSDATE, 'Month')||' , '||
        TO_CHAR (SYSDATE, 'YYYY') Idea FROM DUAL;
```

```
SQL> SELECT TO_CHAR(SYSDATE, 'FMDAY' )||''s Special' "MENU" FROM DUAL;
SQL> SELECT Ename, Job, NVL(TO_CHAR(MGR), 'Supreme Authority') "Managers"
                                                    FROM Emp  ORDER BY Sal DESC;
```

Spelling a Number

```
SQL> SELECT TO_CHAR(TO_DATE('&GiveNumber', 'J'),'JSP') "Spelled Number"
                                                                FROM DUAL;
```

Selecting a Date Specific to Its Century

```
SQL> SELECT TO_CHAR(TO_DATE(HireDate, 'DD-Mon-RRRR'),  'DD-Mon-YYYY') HireDate
                                                                FROM Emp;
```

Text Encryption

```
SQL> SELECT  'SATISH KUMAR YELLANKI' OrgName,  TRANSLATE( 'SATISH KUMAR
        YELLANKI','ABCDEFGHIJKLMNOPQRSTUVWXYZ',
                '1234567890!@#$%^&*()-=_+;,.' ) EncryptedName  FROM DUAL;
```

Text Decryption

```
SQL> SELECT '(1)9(8.!-#1*.;5@@1$!9' EncryptedName, TRANSLATE
      ('(1)9(8.!-#1*.;5@@1$!9', '1234567890!@#$%^&*()-=_+;,.',
            'ABCDEFGHIJKLMNOPQRSTUVWXYZ' ) DecryptedName FROM DUAL;
```

Let Us Leap For Aggregating the Data

Aggregate or Group Functions:

- These Functions Return a Single Row Based on Groups of Rows.
- These Functions Can Appear in SELECT Lists and HAVING Clauses Only.
- These Functions Operate on Sets of Rows to Give One Result Per Group.
- The Sets May Be The Whole Table or The Table Split Into Groups.

Guidelines to use Group Functions:

- DISTINCT Makes The Function to Consider Only Non Duplicate Values.
- ALL Makes The Function to Consider Every Value Including Duplicates.

Syntax: GroupFunctionName(Distinct/ALL Col)

- The Data Types For Arguments May Be CHAR, VARCHAR2, NUMBER OR DATE.
- All Group Functions Except COUNT(*) Ignore NULL Values. To Substitute a Value For NULL Value, Use the NVL Function.
- When a Group Function is Declared in a SELECT List, no Single Row Columns Should be Declared.
- When a Group Function is Declared in a SELECT List, Other Columns Can Be Declared, But They Should Be Grouped Columns, And All The Non Functional Columns Should Be Declared Into a GROUP BY Clause.

Average Function

Syntax: AVG(DISTINCT/ALL Col)

- It Returns the AVERAGE Value of Column.
- It Ignores NULL Values.

```
SQL> SELECT AVG(Sal), AVG(DISTINCT Sal)  FROM Emp;
SQL> SELECT AVG(Comm),AVG(DISTINCT AVG) FROM Emp;
```

SUM Function

Syntax: SUM(DISTINCT/ALL Col)

- It Returns the SUM Value of Column.
- It ignores NULL Values.

```
SQL> SELECT SUM(Sal), SUM(DISTINCT Sal) FROM Emp;
SQL> SELECT SUM(Comm), SUM(DISTINCT Comm) FROM Emp;
```

MAXIMUM Function

Syntax: MAX(DISTINCT/ALL Col)

- It Returns the Maximum Value of Column.
- It Ignores NULL Values.

```
SQL> SELECT MAX(Sal), MAX(DISTINCT Sal) FROM Emp;
SQL> SELECT MAX(Comm) , MAX(DISTINCT Comm)FROM Emp;
```

MINIMUM Function

Syntax: MIN(DISTINCT/ALL Col)

- It Returns the Minimum Value of The Column.
- It Ignores NULL values.

SQL> SELECT MIN(Sal), MIN(DISTINCT Sal)FROM Emp;
SQL> SELECT MIN(Comm), MIN(DISTINCT Comm) FROM Emp;

STANDARD DEVIATION Function

Syntax: STDDEV(DISTINCT/ALL Col)

- It Returns the Standard Deviation of The Column
- It Ignores NULL Values.

SQL> SELECT STDDEV(Sal), STDDEV(DISTINCT Sal) FROM Emp;
SQL> SELECT STDDEV(Comm), STDDEV(DISTINCT Comm) FROM Emp;

VARIANCE Function

Syntax: VARIANCE(DISTINCT/ALL Col)

- It Returns the Variance of N.
- It Ignores the NULL Values.

SQL> SELECT VARIANCE(Sal), VARIANCE(DISTINCT Sal) FROM Emp;
SQL> SELECT VARIANCE(Comm), VARIANCE(DISTINCT Comm) FROM Emp;

COUNT Function

Syntax: COUNT(*/DISTINCT/ALL Col)

- It Returns the Number of Rows in The Query.
- If '*' is Used Returns All Rows, Including Duplicated And NULLs.
- It Can Be Used to Specify The Count of All Rows or Only Distinct Values of Col.

SQL> SELECT COUNT(*) FROM Emp;
SQL> SELECT COUNT(Job), COUNT(DISTINCT Job) FROM Emp;
SQL> SELECT COUNT(Sal), COUNT(Comm) FROM Emp;
SQL> SELECT COUNT(Empno), COUNT(DISTINCT MGR) FROM Emp;

Creating Groups of Data

- The Group By Clause is used to decide the rows in a table into groups.

Syntax 1:

```
SELECT ColumnName1, ColumnName2, ...
                FROM TableName
                WHERE Condition(s)
                GROUP BY ColumnName(s)
                ORDER BY Column(s);
```

Syntax 2:

```
SELECT ColumnName, GRP_FUN(Column)
FROM TableName
WHERE Condition(s)
GROUP BY ColumnName(s)
ORDER BY Column(s);
```

Guidelines to Use GROUP BY Clause:

- If The GROUP Function is Included in a SELECT Clause, We Should Not Use Individual Result Columns.
- The Extra Non Group Functional Columns Should Be Declared in The GROUP BY Clause.
- Using WHERE Clause, Rows Can Be Pre Excluded Before Dividing Them Into Groups.
- Column Aliases Cannot Be Used in GROUP BY CLAUSE.
- By Default, Rows are Sorted By Ascending Order of The Columns Included in The GROUP BY LIST.
- The Column Applied Upon GROUP BY Clause Need Not be Part of SELECT list.

SQL> SELECT Deptno FROM Emp GROUP BY Deptno;
SQL> SELECT Job FROM Emp GROUP BY Job;
SQL> SELECT MGR FROM Emp GROUP BY MGR;

```
SQL> SELECT TO_CHAR(HireDate, 'YYYY') YearGroup FROM Emp  GROUP BY
                                                    TO_CHAR(HireDate, 'YYYY');
SQL> SELECT TO_CHAR(HireDate, 'Month') MonthGroup FROM Emp
                                    GROUP BY TO_CHAR(HireDate, 'Month');
SQL> SELECT TO_CHAR(HireDate, 'Month') MonthGroup FROM Emp WHERE
                                    TO_CHAR(HireDate, 'Month') <> 'September'
                                        GROUP BY TO_CHAR(HireDate, 'Month');
```

Creating Group Wise Summaries

```
SQL> SELECT Deptno, AVG(Sal)  FROM Emp GROUP BY Deptno;
SQL> SELECT Deptno, AVG (Sal)  FROM Emp GROUP BY Deptno  ORDER By AVG (Sal) ;
SQL> SELECT Deptno, MIN(Sal), MAX(Sal) FROM Emp  GROUP BY Deptno;
SQL> SELECT Deptno, Job, SUM(Sal) FROM Emp GROUP BY Deptno, Job;
```

NOTE: The Above Specification Falls Under The Principle of Groups Within Groups.

```
SQL> SELECT Deptno, MIN(Sal), MAX(Sal)  FROM Emp  WHERE Job = 'CLERK'
                                                    GROUP BY Deptno;
SQL> SELECT Deptno, SUM(Sal), AVG(Sal)  FROM Emp  WHERE Job = 'CLERK'
                                                    GROUP BY Deptno;
```

Excluding Groups of Results

Having Clause

- It is Used to Specify Which Groups Are to be Displayed.
- The Clause is Used to Filter Data That is Associated With Group Functions.

Syntax:

```
SELECT Column, Group_Function
FROM Table
[WHERE Condition(s)]
[GROUP BY Group_By_Expr]
[HAVING Group_Condition(s)]
[ORDER BY Column_Name/Alias];
```

Steps Performs By Having Clause:

- First The Rows are Grouped.
- Second The Group Function is Applied to The Identified Groups.
- Third The Groups That Match The Criteria in The HAVING Clause are Displayed.
- The HAVING Clause can Precede GROUP BY Clause, But it is More Logical to Declare it After GROUP BY Clause.
- GROUP BY Clause Can Be Used, Without a Group Function in The SELECT list.
- If Rows are Restricted Based On The Result of a Group Function, We Must Have a GROUP BY Clause as well as the HAVING Clause.
- Existence of GROUP BY Clause Does Not Guarantee The Existence of HAVING Clause, But The Existence of HAVING Clause Demands the Existence of GROUP BY Clause.

```
SQL> SELECT Deptno, AVG(Sal)  FROM Emp  GROUP BY Deptno  HAVING MAX(Sal) > 2900;
SQL> SELECT Job, SUM(Sal) Payroll  FROM Emp  WHERE Job NOT LIKE 'SALES%';
            GROUP BY Job HAVING SUM(Sal) > 5000 ORDER BY SUM (Sal);
SQL> SELECT Deptno, MIN(Sal), MAX(Sal) FROM Emp  WHERE Job = 'CLERK'
                                    GROUP BY Deptno HAVING MIN (Sal) < 1000;
SQL> SELECTDeptno,SUM(Sal) FROM Emp GROUP BY Deptno  HAVING COUNT(Deptno) > 3;
SQL> SELECT Deptno, AVG(Sal), SUM(Sal), MAX(Sal), MIN(Sal) FROM Emp GROUP BY
                                    Deptno  HAVING COUNT(*) > 3;
SQL> SELECT Deptno, AVG(Sal), SUM(Sal)  FROM Emp  GROUP BY Deptno
                                                HAVING AVG ( Sal ) > 2500;
SQL> SELECT Deptno, Job, SUM(Sal), AVG(Sal)  FROM Emp  GROUP BY Deptno, Job
                                HAVING AVG(Sal) > 2500;
```

Nesting of Group Functions:

- Group Functions Can be Nested To a Depth of Two Levels.

SQL> SELECT MAX(AVG(Sal)) FROM Emp GROUP BY Deptno;
SQL> SELECT MAX(SUM(Sal)), MIN(SUM(SAL)) FROM Emp GROUP BY Deptno;
SQL> SELECT MAX(SUM(Sal)), MIN(AVG(Sal)) FROM EmpGROUP BY Job;

Miscellaneous Functions

GREATEST Function

Syntax: GREATEST(Expr1, Expr2, ...)

- Returns the Greatest of The List of expr.
- All Exprs After The First are Implicitly Converted to The Data Type of The First Expr, Before The Comparison.
- Oracle Compares The Exprs Using Non Padded Comparison Semantics.
- Character Comparison is Based on The Value of The Character in The Data Base Character List.

SQL> SELECT GREATEST('HARRY', 'HARRIOT') FROM DUAL;
SQL> SELECT GREATEST(1000, 2000, 200) FROM DUAL;
SQL> SELECT GREATEST('10-JUL-05', '20-JUL-05') FROM DUAL;

LEAST Function

Syntax: LEAST(Expr1, Expr2, ...)

- It Returns the Least of The List of Exprs.
- All Exprs After The First Are Implicitly Converted to The Data Type of The First Expr Before the Comparison .

SQL> SELECT LEAST('HARRY' , 'HARRIOT') FROM DUAL;
SQL> SELECT LEAST(1000, 2000, 200)FROM DUAL;
SQL> SELECT LEAST('10-JUL-05', '20-JUL-05') FROM DUAL;

USER Function

Syntax: USER

- It Returns the Current Oracle Users Name Within The VARCHAR2 Data Type.
- The Function Cannot Be Used in The Condition of The CHECK Constraint .

SQL> SELECT USER FROM DUAL;

UID Function

Syntax: UID

- It Returns an Integer that Uniquely Identifies the Current User.

SQL> SELECT UID FROM DUAL;
SQL> SELECT USER, UID FROM DUAL;

USERENV Function

Syntax: USERENV(Option)

- Returns Information of VARCHAR2 Data Type Above the Current Session.

The Values in Options are

- 'ISDBA' → Returns 'TRUE' if DBA Role is Enabled.
- 'LANGUAGE' → Returns The Language And Territory Used in Current Session
- 'TERMINAL' → Returns The OS Identifier for The Current Session's Terminal.
- 'SESSIONID' → Returns The Auditing Session Identifier.
- 'ENTRYID' → Returns The Available Auditing Entry Identifier.
- 'LANG' → Returns The ISD Abbreviation For The Language Name.
- 'INSTANCE' → Returns The Instance Identification Number of The Current Instance.
- 'CLIENT_INFO' → Returns up to 64 Bytes of User Session Information.

SQL> SELECT USERENV('ISDBA') FROM DUAL;
SQL> SELECT USERENV('LANGUAGE') FROM DUAL;
SQL> SELECT USERENV('TERMINAL') FROM DUAL;
SQL> SELECT USERENV('SESSIONID') FROM DUAL;
SQL> SELECT USERENV('LANG') FROM DUAL;

VSIZE Function

Syntax: VSIZE(Expr)

- It Returns The NUMBER of Bytes in The Internal Representation of Expr.
- If Expr is NULL, Function Returns NULL .

SQL> SELECT Ename, VSIZE(Ename) FROM Emp;
SQL> SELECT Ename, Sal, VSIZE(Sal) FROM Emp;
SQL> SELECT Ename, HireDate, VSIZE(HireDate) FROM Emp;

SOUNDEX Function

Syntax: SOUNDEX(CHAR)

- It Returns a Character String Containing the Phonetic Representation of CHAR.
- It Allows Comparison of Words That are Spelled Differently, But Sound Alike in English.

SQL> SELECT Ename FROM Emp WHERE SOUNDEX(Ename) = SOUNDEX('SMYTHE');
SQL> SELECT Ename, Job FROM Emp WHERE SOUNDEX(JOB) = SOUNDEX('CLRK');
SQL> SELECT Ename, Job FROM Emp WHERE SOUNDEX(JOB) = SOUNDEX('manger');

Let us Increase the Integrity and Quality of Our Database

Data Integrity in Data Bases

Data Integrity

- It is a State in Which All The Data Values Stored in The Data Base Are Correct.
- Enforcing Data Integrity Ensures The Quality of The Data in The Data Base.

Categories of Data Integrity

- Entity Integrity
- Domain Integrity
- Referential Integrity
- User Defined Integrity

Entity Integrity

- It Defines a Row As a UNIQUE Entity For a Particular Table.
- Entity Integrity Enforces The Integrity of the Identifier Columns, or the PRIMARY KEY of a Table.

Illustration: 1

Student Name	Date of Birth	Date of Admission	Course Name	Course Fee
Sampath	02-Jan-76	15-Jun-00	MCA	25000.00
Sampath	02-Jan-76	15-Jun-00	MBA	25000.00
Srinivas	02-Jan-76	15-Jun-00	MBA	25000.00
Srinivas	02-Jan-76	25-Aug-02	MBA	25000.00
Sampath	10-Dec-80	25-Aug-02	MCA	25000.00
Srinivas	02-Jan-76	25-Aug-02	M.Sc	15000.00

Illustration: 2

Student Name	Date of Birth	Date of Admission	Course Name	Course Fee	Email ID
Sampath	02-Jan-76	15-Jun-00	MCA	25000.00	sampath@gmail.com
Sampath	02-Jan-76	15-Jun-00	MBA	25000.00	sampath@yahoo.com
Srinivas	02-Jan-76	15-Jun-00	MBA	25000.00	srinivas@gmail.com
Srinivas	02-Jan-76	25-Aug-02	MBA	25000.00	srinivas@yahoo.com
Sampath	10-Dec-80	25-Aug-02	MCA	25000.00	sampath@rediff.com
Srinivas	02-Jan-76	25-Aug-02	M.Sc	15000.00	srinivas@rediff.com

Illustration: 3

Student ID	Student Name	Date of Birth	Date of Admission	Course Name	Course Fee	Email ID
1000	Sampath	02-Jan-76	15-Jun-00	MCA	25000.00	sampath@gmail.com
1001	Sampath	02-Jan-76	15-Jun-00	MBA	25000.00	sampath@yahoo.com
1002	Srinivas	02-Jan-76	15-Jun-00	MBA	25000.00	srinivas@gmail.com
1003	Srinivas	02-Jan-76	25-Aug-02	MBA	25000.00	srinivas@yahoo.com
1004	Sampath	10-Dec-80	25-Aug-02	MCA	25000.00	sampath@rediff.com
1005	Srinivas	02-Jan-76	25-Aug-02	M.Sc	15000.00	srinivas@rediff.com

Domain Integrity

- Domain Integrity Validates The Entries For a Given Column.
- It Can Enforced Through.

- Restricting Type (Data Types)
- By Format (CHECK Constraint)
- By Range of Possible Values Using
  - FOREIGN KEY Constraint.
  - CHECK Constraint.
  - DEFAULT Key Word.
  - NOT NULL Constraint.
- Domain Integrities Major Fact of Concentration is on the Data That is Being Collected in That Column.

Illustration: 4

Student ID	Student Name	Date of Birth	Date of Admission	Course Name	Course Fee	Email ID
1000	Sampath	02-Jan-76	15-Jun-00	MCA	25000	sampath@gmail.com
1001	SAMPATH		15-Jun-00	MBA	25000	
1002	srinivas	02-Jan-76	15-Jun-00	MBA	25000	srinivas@gmail.com
1003	SRINIVAS	02-Jan-98	04-Aug-07	MBA	25000	srinivas@yahoo.com
1004	sampath		25-Aug-02	MCA	25000	
1005	Srinivas	02-Jan-76		M.Sc	15000	srinivas@rediff.com

Referential Integrity

- It Preserves The Defined Relationship Between Tables When Records Are Entered or Deleted.
- It Ensures That Key Values Are Consistent Across Tables.
- When Referential Integrity is Enforced, it Prevents From…
  - Adding Records to a Related Table if There is no Associated Record in The Primary Table.
  - Changing Values in a Primary Table That Result in Orphaned Records in a Related Table.
  - Deleting Records From a Primary Table if There are Matching Related Records.

Illustration: 5

Courses Information

Course ID	Course Name	Course Fees
C001	MBA	25000
C002	MCA	25000
C003	M.Sc	15000

Students Information

Student ID	Student Name	Date of Birth	Date of Admission	Course ID	Email ID
1000	Sampath	02-Jan-76	15-Jun-00	C002	sampath@gmail.com
1001	SAMPATH		15-Jun-00	C001	
1002	srinivas	02-Jan-76	15-Jun-00	C001	srinivas@gmail.com
1003	SRINIVAS	02-Jan-98	04-Aug-07	C001	srinivas@yahoo.com
1004	sampath		25-Aug-02	C002	
1005	Srinivas	02-Jan-76		C003	srinivas@rediff.com

User Defined Integrity

- It Allows To Define Specific Business Rules That Do Not Fall Into Any One of The Other Integrity Categories.
- These are Business Rules Which Can be Handled at Run Time, Usually Designed Using Database Triggers in PL/SQL.
- These are Rules Generally Specific to the Organizational Business Process.
- Can be Any Situation That Looks Abnormal to the Current Systems Process.

Constraints in Oracle

- Constraints in Data Bases Are Used to Define An Integrity Constraint, As a Rule That Restricts The Values in a Data Base.
- As Per Oracle There are Six Types of Constraints...
  - NOT NULL Constraint.
  - UNIQUE Constraint.
  - PRIMARY KEY Constraint.
  - FOREIGN KEY Constraint.
  - CHECK Constraint.
  - REF Constraint.

Declaration Style

- Column Level (OR) IN LINE Style.
- Table Level (OR) OUT OF LINE Style.

Column Level:

- They Are Declared As Part of The Definition of An Individual Column or Attribute.
- Usually Applied When the Constraint is Specific To That Column Only.

Table Level:

- They Are Declared As Part of The Table Definition.
- Definitely Applied When the Constraint is Applied on Combination of Columns Together.

Note: NOT NULL Constraint is The Only Constraint Which Should Be Declared As INLINE Only.

- Every Constraint is Managed By Oracle With a Constraint Name in The Meta Data.
- Hence When We Declare a Constraint if We Do not Provide a Constraint Name Oracle Associates the Constraint With Name.
- With in a Single User No Two Constraints Can Have The Same Name.
- Rather Than Depending on the Oracle Supplied Constraint Name, it is Better to Define Our Own Name for all Constraints.
- When Constraints are Named We Should Use 'CONSTRAINT' Clause.

The CONSTRAINT Clause Can Appear in

- CREATE And ALTER Table Statement.
- CREATE And ALTER View Statement.
- Oracle Does Not Support Constraints on Columns or Attributes Whose Data Type is
  - USER_DEFINED OBJECTS.
  - NESTED TABLES And VARRAY.
  - REF And LOB.

Exceptions

- NOT NULL Constraint is Supported For an Attribute Whose Data Type is USER_DEFINED Object, VARRAY, REF, LOB.
- NOT NULL, FOREIGN KEY, and REF Constraints Are Supported on a Column of TYPE REF.

NOT NULL Constraint:

- A NOT NULL Constraint Prohibits a Column From Containing NULL Values.
- NOT NULL Should Be Defined Only At COLUMN Level.
- The Default Constraint if Not Specified is NULL Constraint.
- To Satisfy the Rule, Every Row in The Table Must Contain a Value For The Column.

Restrictions:

- NULL or NOT NULL Cannot Be Specified as View Constraints.
- NULL or NOT NULL Cannot Be Specified For An Attribute of An Object.

Syntax

```
SQL> CREATE Table <Table_Name>
      (   Column_Name1 <Data Type>(Width) NOT NULL,
          Column_Name2 <Data Type>(Width)
          CONSTRAINT ConsName NOT NULL,
          Column_NameN  <Data Type>(Width)   );
```

Illustration

```
SQL> CREATE TABLE Students
       ( StudNo              NUMBER(6) CONSTRAINT StudnoNN NOT NULL,
         StudName            VARCHAR2(25) CONSTRAINT StudNameNN NOT NULL,
         CourseName          VARCHAR2(25) CONSTRAINT CourseNameNN NOT NULL,
         JoinDate            DATE NOT NULL );
```

UNIQUE Constraint

- The UNIQUE Constraint Designates a Column As a UNIQUE Key.
- A Composite UNIQUE Key Designates a Combination of Columns As The UNIQUE Key.
- A Composite UNIQUE Key is Always Declared At The Table Level.
- To Satisfy a UNIQUE Constraint, No Two Rows in The Table Can Have The Same Value For The UNIQUE Key.
- UNIQUE Key Made Up of a Single Column Can Contain NULL Values.
- Oracle Creates An Index Implicitly on The UNIQUE Key Column.

Restrictions:

- A Table or View Can Have Only One UNIQUE Key Column.
- UNIQUE Key Cannot Be Implemented on Columns Having
  - LOB, LONG, LONG RAW, VARRAY, NESTED TABLE, OBJECT, BFILE, REF, TIMESTAMP WITH TIME ZONE .
- A Composite UNIQUE Key Cannot Have More Than 32 Columns.
- Same Column or Combination of Columns Cannot Be Designated As Both PRIMARY KEY and UNIQUE KEY.
- We Cannot Specify a UNIQUE Key When Creating a Sub Table or Sub View in An Inheritance Hierarchy.
- The UNIQUE Key Can Be Specified Only For The Top Level (Root) Table or View.

Syntax

```
SQL> CREATE Table <Table_Name>
      (
        Column_Name1  <Data Type>(Width) UNIQUE,
        Column_Name2  <Data Type>(Width)
        CONSTRAINT ConsName UNIQUE,
        Column_NameN  <Data Type>(Width)
      );
```

Illustration: 1

Column Level Syntax

```
SQL> CREATE Table Promotions
       ( Promo_ID  NUMBER(6)  CONSTRAINT PromoIDUNQ UNIQUE,
         PromoName  VARCHAR2(20),  PromoCategory  VARCHAR2(15),
         PromoCost  NUMBER(10, 2), PromoBegDate  DATE, PromoEndDate DATE );
```

Illustration: 2

Table Level Syntax

```
SQL> CREATE Table Promotions
       ( Promo_ID NUMBER(6),  PromoName  VARCHAR2(20),
         PromoCategory  VARCHAR2(15), PromoCost  NUMBER(10, 2),
         PromoBegDate DATE,  PromoEndDate  DATE, CONSTRAINT PromoIDUNQ
```

```
                                        UNIQUE(Promo_ID) );
```

Illustration: 3

Composite UNIQUE Constraint Syntax

```
SQL> CREATE Table WareHouse
        ( WareHouseID NUMBER(6), WareHouseName VARCHAR2(30),
          Area NUMBER(4), DockType VARCHAR2(50), WaterAccess VARCHAR2(10),
          RailAccess VARCHAR2(10), Parking VARCHAR2(10), Vclearance NUMBER(4),
              CONSTRAINT WareHouseUNQ UNIQUE(WareHouseID,WareHouseName) );
```

Illustration: 4

```
SQL> CREATE TABLE Students
        ( StudID NUMBER(6) CONSTRAINT StudIDUNQ UNIQUE, Fname VARCHAR2(30),
          DOB DATE, DOJ DATE, EmailID VARCHAR2(50) CONSTRAINT EmailIDUNQ
                                        UNIQUE );
```

PRIMARY KEY Constraint

- A PRIMARY KEY Constraint Designates a Column As The PRIMARY KEY of a TABLE or VIEW.
- A COMPOSITE PRIMARY KEY Designates a Combination of Columns As The PRIMARY KEY.
- When The Constraint is Declared At Column Level Only PRIMARY KEY Keyword is Enough.
- A Composite PRIMARY KEY is Always Defined At Table Level Only.
- A PRIMARY KEY Constraint Combines a NOT NULL and UNIQUE Constraint in One Declaration.

Restrictions:

- A TABLE or VIEW Can Have Only One PRIMARY KEY.
- PRIMARY KEY Cannot Be Implemented on Columns Having...
  - LOB, LONG, LONG RAW, VARRAY, NESTED TABLE, OBJECT, BFILE, REF, TIMESTAMP WITH TIME ZONE
- The Size of A PRIMARY KEY Cannot Exceed Approximately One Database Block.
- A Composite PRIMARY KEY Cannot Have More Than 32 Columns.
- The Same Column or Combination of Columns Cannot Be Designated Both As PRIMARY KEY and UNIQUE KEY.
- PRIMARY KEY Cannot Be Specified When Creating a Sub Table or Sub View in An Inheritance Hierarchy.
- The PRIMARY KEY Can be Specified Only For The Top Level (ROOT)TABLE or VIEW.

Syntax

```
SQL> CREATE Table <Table_Name>
     (    Column_Name1    <Data Type>(Width)
          CONSTRAINT ColNamePK PRIMARY KEY,
          Column_Name2    <Data Type>(Width),
          Column_NameN    <Data Type>(Width) );
```

Illustration: 1

Column Level Syntax:

```
SQL> CREATE TABLE Locations
        ( LocationID   NUMBER(4) CONSTRAINT LocIDPK PRIMARY KEY,
          StAddress VARCHAR2(40) NOT NULL, PostalCode VARCHAR2(6)
          CONSTRAINT PCNN NOT NULL, City VARCHAR2(30) CONSTRAINT CityNN
                                                    NOT NULL );
```

Illustration: 2

Table Level Syntax

```
SQL> CREATE TABLE Locations
        ( LocationID NUMBER(4), StAddress   VARCHAR2(40) NOT NULL,
          PostalCode VARCHAR2(6) CONSTRAINT PCNN NOT NULL,
          City VARCHAR2(30) CONSTRAINT CityNN NOT NULL,
```

CONSTRAINT LocIDPK PRIMARY KEY(LocationID));

Analyze The Following Data For Primary Key

SaleID	CustID	ProdID	Qty	SaleDate	SaleDesc
S001	C001	P001	250.00	01-Aug-07	Cash
S001	C001	P002	125.00	01-AUG-07	Cash
S002	C002	P003	50.00	01-Aug-07	Cash
S002	C002	P004	75.00	01-Aug-07	Credit
S002	C002	P010	225.00	01-Aug-07	Credit
S002	C002	P003	125.00	01-Aug-07	Cash
S003	C001	P005	200.00	01-Aug-07	Credit
S003	C001	P002	25.00	01-Aug-07	Cash
S003	C001	P015	354.00	01-Aug-07	Credit
S004	C003	P100	245.00	02-Aug-07	Cash
S005	C001	P002	125.00	03-Aug-07	Cash
S006	C002	P004	75.00	03-Aug-07	Cash

Illustration: 2

```
SQL> CREATE TABLE SalesInfo
       ( SaleID  NUMBER(6), CustID NUMBER(6), ProdID NUMBER(6),
         Quantity  NUMBER(6) NOT NULL, SaleDate DATE NOT NULL,
         SaleDesc  LONG NOT NULL, CONSTRAINT ProdCustIDPK
                                  PRIMARY KEY(SaleID, ProdID, CustID));
```

- PRIMARY KEY Constraint With Composite Key Constraint Style.

FOREIGN KEY Constraint

- It is Also Called As REFERENTIAL INTEGRITY CONSTRAINT.
- It Designates a Column as FOREIGN KEY And Establishes a RELATION Between The FOREIGN KEY And a Specified PRIMARY or UNIQUE KEY.
- A COMPOSITE FOREIGN KEY Designates a Combination of Columns As The FOREIGN KEY.
- The TABLE or VIEW Containing The FOREIGN KEY is Called the Child Object.
- The TABLE or VIEW Containing The REFERENCED KEY is Called the Parent Object.
- The FOREIGN KEY And The REFERENCED KEY Can Be in The Same TABLE or VIEW.
- The Corresponding Column or Columns of the FOREIGN KEY And The REFERENCED KEY Must Match in ORDER and DATA TYPE.
- A FOREIGN KEY CONSTRAINT Can Be Defined on a Single Key Column Either Inline or Out of Line.
- A COMPOSITE FOREIGN KEY on Attributes Should Be Declared at TABLE LEVEL or Out of Line Style.
- We Can Designate the Same Column or Combination of Columns as Both a FOREIGN Key and a PRIMARY or UNIQUE KEY.
- A COMPOSITE FOREIGN KEY CONSTRAINT, Must Refer To a COMPOSITE UNIQUE KEY or a COMPOSITE PRIMARY KEY in the PARENT TABLE or VIEW.

Restrictions:

- The FOREIGN KEY Columns Cannot be Applied on...
  - LOB, LONG, LONG RAW, VARRAY, NESTED TABLE, OBJECT, BFILE, REF, TIMESTAMP WITH TIME ZONE.

- The REFERENCED UNIQUE/PRIMARY KEY Constraint on The PARENT TABLE OR VIEW Must Already Be Detected.
- A COMPOSITE FOREIGN KEY Cannot Have More Than 32 Columns.
- The Child and Parent Tables Must Be on The Same Database.
- To Enable REFERENTIAL INTEGRITY Across Nodes of a Distributed Database TRIGGERS Are Used.

REFERENCES Clause:

- The REFERENCES CLAUSE Should be Used When The FOREIGN KEY Constraint is INLINE.
- When The Constraint is OUT OF LINE, We Must Specify The FOREIGN KEY, Key Word.

ON DELETE Clause:

- The ON DELETE Clause Determines How ORACLE Automatically Maintains REFERENTIAL INTEGRITY if The REFERENCED PRIMARY or UNIQUE KEY Value is Removed From Master Table.
- CASCADE Option Can Be Specified if We Want ORACLE to Remove DEPENDENT FOREIGN KEY Values.
- Specify SET NULL if We Want ORACLE to Convert Dependent FOREIGN KEY Values to NULL.
- ON DELETE Clause Cannot be Specified for a View Constraint.
- Declared on FOREIGN KEY Column Only.

Things To Note Before We Apply Relations

Relation Model Symbols

- Entity OR Table 
- Column OR Attribute 
- Relation OR Association 
- Associative Entity OR Table 
- Connecting Line 

Types of Relations:

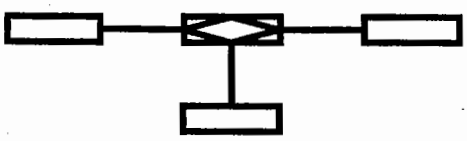
- Unary Relation.
- Binary Relation.
- Ternary Relation.
- N'Ary Relation

Relation Model Representation:

Unary Relation:

Binary Relation:

Ternary Relation:



Relation Cardinality:

- One-TO-One. → 1..1
- One-TO-Many. → 1..* OR 1:M
- Many-TO-Many. → *..* OR M:N OR M..N

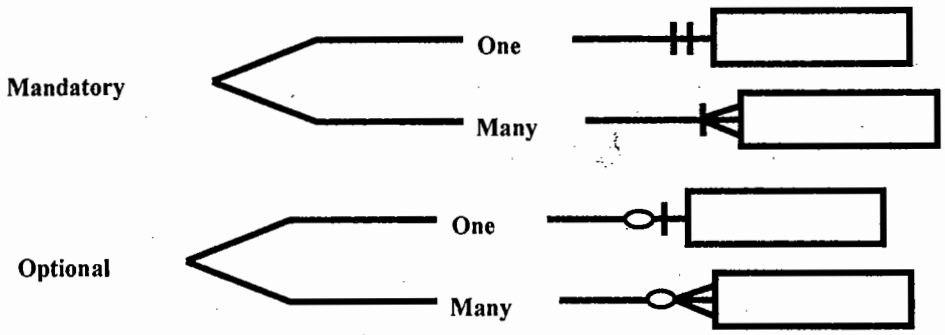
Relation State:

- Mandatory State.
- Optional State.

Relation State With Cardinality:

Types of Tables To Be Identified in Design Process

1. Pure Masters:



- These are Tables Which Contain Only Primary Keys, And All The Remaining Columns are Non Keys.

2. Master Details:

- These are Tables Containing Their Own Primary Key and are Also Related To Them Selves or Other Tables With Foreign Keys.

3. Pure Details:

- These are Tables Which Contain Only Foreign Keys, Related To Other Table or Tables Primary Key.

Steps Followed for Creating Foreign Key Constraint are as Follows:

Step 1: Create Primary Master's / Pure Master's

```
SQL> CREATE TABLE Dept
       ( Deptno NUMBER(2)  CONSTRAINT Deptno_Pk PRIMARY KEY,
         Dname  VARCHAR2(20)  CONSTRAINT Dname_NN NOT NULL,
         Location VARCHAR2(20)  CONSTRAINT Loc_NN NOT NULL  );
```

Step 2: Create Detailed / Child Table

- These Are Tables Which Can Contain Primary Key of Their Own As Well As Foreign Key's Referring To Other Primary Master's or To Them Selves.
- These Tables Are Also Called As Dependent Tables or Referential Tables.

```
SQL> CREATE TABLE Employee (
        EmployeeID  NUMBER(6) CONSTRAINT Emp_ID_PK PRIMARY KEY,
        Ename VARCHAR2(30)  CONSTRAINT Ename_NN NOT NULL,
        Designation VARCHAR2(30)  CONSTRAINT Desig_NN NOT NULL,
        ManagerID NUMBER(6) CONSTRAINT Mgr_ID_FK_Self     REFERENCES
                                          Employee(EmployeeID) ON DELETE SET NULL,
        HireDate DATE CONSTRAINT HireDate_NN NOT NULL,
        Commission NUMBER(7, 2),   DeptID NUMBER(2)
        CONSTRAINT DeptID_FK REFERENCES Dept(Deptno) ON DELETE CASCADE );
```

Working With Composite Keys

Step 1: Create Pure Masters

```
SQL> CREATE TABLE  SampleMaster1
      ( SampleID1 NUMBER(4) CONSTRAINT Samp_ID1_PK PRIMARY KEY,
        SampName1 VARCHAR2(20)  CONSTRAINT SampName1_NN NOT NULL ,
        SampDate1 DATE  CONSTRAINT SampDate1_NN NOT NULL );
```

```
SQL> CREATE TABLE  SampleMaster2
      (  SampleID2  NUMBER(4) CONSTRAINT Samp_ID2_PK PRIMARY KEY,
         SampName2  VARCHAR2(20) CONSTRAINT SampName2_NN NOT NULL ,
         SampDate2  DATE CONSTRAINT SampDate2_NN NOT NULL  );
```

Step 2: Create The Pure Details

```
SQL> CREATE TABLE SampRef
      (SampIDRef1  NUMBER(4) CONSTRAINT SampIDRef1_FK
                                 REFERENCES SampMaster1(SampID1),
       SampIDRef2 NUMBER(4) CONSTRAINT SampIDRef2_FK
                                 REFERENCES SampMaster2(SampID2),
       SampNameRef VARCHAR2(20),
       SampDateRef DATE,
       CONSTRAINT SampRef_Comp_PK   PRIMARY KEY(SampIDRef1, SampIDRef2));
```

CHECK Constraint

- It Defines a Condition That Each Row Must Satisfy.
- To Satisfy The Constraint, Each Row in The Table Must Make The Condition Either TRUE or UNKNOWN.
- ORACLE Does Not Verify That CHECK CONDITIONS Are Mutually Exclusive.

Restrictions

- The Condition of a CHECK Constraint Can Refer To Any Column in The Same Table, But It Cannot Refer to Columns of Other Tables.
- The Constructs That Cannot Be Included Are…
  - Queries to Refer to Values in Other Rows
  - Calls to Functions SYSDATE, UID, USER, USERENV.
  - The Pseudo Columns CURRVAL, NEXTVAL, LEVEL or ROWNUM.
  - DATE Constant That Are Not Fully Specified.
- A Single Column Can Have Multiple CHECK Constraints That Can Reference The Column in The Definition.
- There is no Limit To The Number of CHECK Constraints That Can Be Defined On a Column.
- The CHECK Constraints Can Be Defined At The Column Level or TABLE Level.

DEFAULT Option

- The DEFAULT Option is Given to Maintain a DEFAULT Value in a Column.
- The Option Prevents NULL Values From Entering The Columns, if a Row is Inserted Without a Value For a Column.
- The DEFAULT Value Can be a Literal, An Expression or a SQL Function.
- The DEFAULT Expression Must Match The Data Type of the COLUMN.

Example

```
SQL> CREATE TABLE Dept ( Deptno NUMBER(2)   CONSTRAINT CHK_Deptno
                                    CHECK(Deptno BETWEEN  10 AND  90),
       Dname VARCHAR2(15) CONSTRAINT Chk_Dname_UP
                                    CHECK(Dname = UPPER(Dname)) DISABLE ,
       Loc VARCHAR2(15)  CONSTRAINT CHK_Loc CHECK(
              Loc IN('DALLAS', 'BOSTON', 'NEW YORK','CHICAGO')) );
SQL> CREATE TABLE Emp  ( Empno NUMBER(4)  CONSTRAINT PR_Empno  PRIMARY KEY
      ,Ename VARCHAR2(25) NOT NULL  CONSTRAINT CHK_Ename  CHECK
                                    (Ename = UPPER(Ename)),
       JobVARCHAR2(30)  CONSTRAINT Job_NN NOT NULL,
                        CONSTRAINT CHK_Job  CHECK(Job = UPPER(Job)),
       MGR NUMBER(4), HireDate DATE   DEFAULT  SYSDATE,
       Sal NUMBER(7, 2) CONSTRAINT Sal_NN NOT NULL ,
                  CONSTRAINT CHK_Sal  CHECK(Sal BETWEEN 2000 AND 100000)),
       Comm  NUMBER(7, 2),  Deptno   NUMBER(2),
              CONSTRAINT Tot_Sal_Chk CHECK(Sal + Comm <= 100000) );
```

Constraints Maintenance

Adding Constraints to a Table

- A Constraint Can Be Added To a Table At Any Time After The Table Was Created By Using ALTER TABLE Statement, Using ADD Clause.

Syntax

```
SQL> ALTER TABLE <TableName>   ADD   [CONSTRAINT <ConstraintName>]
                                          CONS_TYPE(Column_Name);
```

- The Constraint Name in the Syntax is Optional, But Recommended.

Guidelines

- We Can ADD, DROP, ENABLE, or DISABLE a Constraint , but Cannot Modify The Physical Structure of The Table.
- A NOT NULL Can Be Added to Existing Column By Using The MODIFY Clause of the ALTER TABLE Statement.
- NOT NULL Can Be Defined Only When The Table Contains No Rows.

Example

```
SQL> ALTER TABLE Emp  ADD CONSTRAINT Emp_Mgr_FK  FOREIGN KEY(Mgr)
                                          REFERENCES Emp(Empno);
```

DROPPING Constraints

- To DROP a Constraint Identify The Constraint Name From The
  - USER_CONSTRAINTS
  - USER_CONS_COLUMNS Data Dictionary Views.
- The ALTER TABLE Statement is Used With The DROP Clause.
- The CASCADE Option of the DROP Clause Causes Any Dependent Constraints Also To Be Dropped.
- When a Constraint is Dropped, The Constraint is No Longer Enforced And is No Longer Available in The Data Dictionary.

Syntax

```
SQL> ALTER TABLE <Table_Name>  DROP  PRIMARY KEY/UNIQUE(Column)/
                                      CONSTRAINT ConstraintName[CASCADE];
```

Example

```
SQL> ALTER TABLE Dept  DROP PRIMARY KEY CASCADE;
SQL> ALTER TABLE Emp  DROP CONSTRAINT Emp_Mgr_FK;
```

ENABLING Constraints

- The Constraint Can Be Enabled Without Dropping it or Re-Creating it.
- The ALTER TABLE Statement With The ENABLE Clause is Used for the Purpose

Syntax

```
SQL> ALTER TABLE<TableName>  ENABLE CONSTRAINT<ConstraintName>;
```

Guidelines

- Enabling a Constraint Applies To All The Data in The Table At a Time.
  - When An UNIQUE or PRIMARY KEY Constraint is ENABLED, The UNIQUE or PRIMARY KEY Index is Automatically Created.
  - The ENABLE Clause Can Be Used Both in CREATE TABLE As Well As ALTER TABLE Statements.

Examples

```
SQL> ALTER TABLE Emp  ENABLE  CONSTRAINT Emp_Empno_FK;
```

VIEWING Constraints

- To View All Constraints On a Table Query Upon the Data Dictionary USER_CONSTRAINTS.
- The Codes That Are Revealed Are…
  - C → CHECK
  - P → PRIMARY KEY
  - R → REFERENTIAL INTEGRITY
  - U → UNIQUE KEY

Example

```
SQL> SELECT  CONSTRAINT_NAME,  CONSTRAINT_TYPE,  SEARCH_CONDITION
                        FROM USER_CONSTRAINTS  WHERE TABLE_NAME = 'EMP';
```

VIEWING The Columns Associated With Constraints

- The Names of The Columns That Are Involved in Constraints Can Be Known By Querying The USER_CONS_COLUMNS Data Dictionary View.

Example

```
SQL> SELECT CONSTRAINT_NAME, COLUMN_NAME  FROM USER_CONS_COLUMNS
                                                WHERE TABLE_NAME = 'EMP';
```

Let Us Make Ourselves More Stronger Using Multi Table Queries

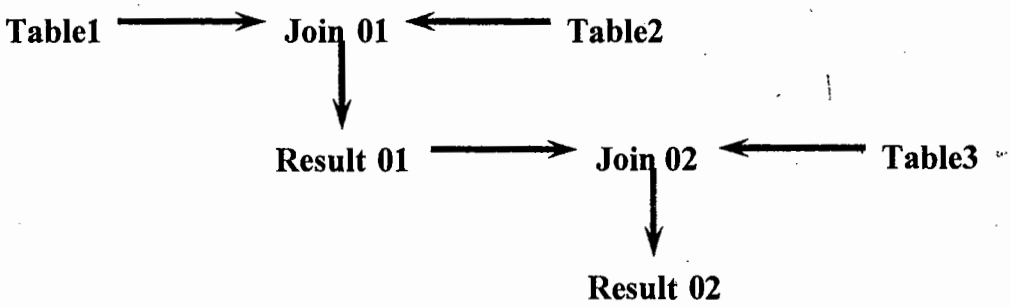
JOINS

- A Join is a Query That Combines Rows From Two or More Tables, Views, or Materialized Views.
- A Join is Performed Whenever Multiple Tables Appear in The Queries FROM Clause.
- The Queries SELECT List Can Select Any Columns From Any of These Tables.
- The Common Column Names Within The Tables Should Qualify All References To These Columns.

```
SQL> SELECT Empno,  Ename, Dname, Loc FROM Emp, Dept;
SQL> SELECT Empno,  Ename, Sal, Grade FROM Emp, SalGrade;
SQL> SELECT Empno,  Ename, Dname, Loc, SalGrade FROM Emp, Dept, SalGrade;
SQL> SELECT Empno,  Ename, Dept.Deptno, Dname, Loc FROM Emp, Dept;
```

JOIN Condition

- Many Join Queries Contain WHERE Clause, Which Compares Two Columns, Each From a Different Table.
- The Applied Condition is Called a JOIN CONDITION.
- To Execute a Join…
  - Oracle Combines Pairs of Rows, Each Containing One Row From Each Table, For Which The JOIN Condition Evaluates to TRUE.
- The Columns in The Join Conditions Need Not Be Part of The SELECT List.
- The WHERE Clause of Join Query Can Also Contain Other Conditions That Refer to Columns of Only One Table.
- To Execute a Join of Three or More Tables
  - Oracle First Joins Two of The Tables Based on The Join Conditions Comparing These Columns And Then Join's The Result To Another Join.



The Oracle Optimizer Determines The Order in Which ORACLE Should Join The Tables Based on…

- Given JOIN Condition(s).
- INDEXES Upon The Tables.
- STATISTICS For The Tables.

- The LOB Columns Cannot Be Specified in The WHERE Clause, When The WHERE Clause Contains Any JOINS.

Syntax

```
WHERE Table1.Column1 = Table2.Column2
```

Guidelines

- When Writing a SELECT Statement That JOIN'S Tables, Precede The Column Name With The Table Name For Clarity And Enhance Database ACCESS.
- If The Same Column Name Appears in More Than One Table, The Column Name Must Be Prefixed With The Table Name.
- To Join 'n' Tables Together, We Need a Minimum of 'n-1' Join Conditions .
- The Above Rule Does Not Apply, if The Table Contains a Concatenated Primary Key.

Equi Joins OR Simple Joins OR Inner Joins

- An EQUIJOIN is a Join With a Join Condition Containing An Equality Operator.
- It Combines Rows That Have Equivalent Values For The Specified Columns.

- The Total Size of Columns in The Equi Join Condition in a Single Table May Be Limited To The Size of a Data Block Minus Some Overhead.
- The Size of The Data Block is Specified By The Initialization Parameter DB_BLOCK_SIZE.

Qualifying Ambiguous Column Names:

- The Names of The Column Names Should Be Qualified in The WHERE Clause, With The Table Name To Avoid Ambiguity.
- If There Are no Common Column Names Between The Two Tables, The Qualification is Not Necessary But It is Better.

```
SQL> SELECT Emp.Empno  Empno, Emp.Ename  Ename,  Emp.Deptno  Deptno,
       Dept.Deptno  Deptno, Dept.Dname Dname, Dept.Loc Loc FROM Emp, Dept
                                                WHERE Emp.Deptno = Dept.Deptno;
SQL> SELECT Empno, Ename, Emp.Deptno, Loc FROM Emp, Dept WHERE
        Emp.Deptno = Dept.Deptno AND Job = UPPER('manager');
SQL> SELECT Empno, Ename, Sal * 12 AnnSal, Emp.Deptno, Loc FROM Emp, Dept
                                                WHERE Emp.Deptno = Dept.Deptno;
```

Using Table Aliases:

- Tables Aliases Can Be Used Instead of Original Table Names.
- A Table Alias Gives an Alternate Name For The Existing Queried Table.
- Table Aliases Help in Keeping The SQL Code Smaller, Hence Using Less Memory.
- The Table Alias is Specified in The FROM Clause.

Guidelines:

- A Table Alias Can Be Up To 30 Characters in Length.
- If a Table Alias is Used For a Particular Table Name in The FROM Clause, Then That Table Alias Must be Substituted For The Table Name Through Out The SELECT Statement.
- A Table Alias Should Be Meaningful and Should Be Maintained as Short as Possible.
- A Table Alias is Valid Only For The Current SELECT Statement Only.

```
SQL> SELECT E.Empno, E.Ename, D.Deptno, D.Dname FROM Emp E, Dept D
                                                WHERE E.Deptno = D.Deptno;
SQL> SELECT E.Ename, E.Job, D.Deptno, D.Dname , D.Loc FROM Emp E, Dept D
              WHERE E.Deptno = D.Deptno AND E.Job IN('ANALYST', 'MANAGER' );
SQL> SELECT E.Ename, E.Job, D.Dname, D.Loc FROM Emp E , Dept D
                        WHERE E.Deptno = D.Deptno AND D.Dname <> 'SALES';
```

Self Joins

- It is a Join of a Table To Itself.
- The Same Table Appears Twice in The FROM Clause And is Followed By Table Aliases.
- The Table Aliases Must Qualify The Column Names in The Join Condition.
- To Perform a Self Join, Oracle Combines And Returns Rows of The Table That Satisfy The Join Condition .

Syntax

```
SQL> SELECT Columns FROM Table1 T1, Table1 T2 WHERE T1.Column1 = T2.Column2
```

Illustrations

```
SQL> SELECT E1.Ename "Employee Name", E2. Ename "Managers Name"
                                        FROM Emp E1, Emp E2 WHERE E1.Mgr = E2.Empno;
SQL> SELECT E1.Ename||''s Managers is'|| E2.Ename  "Employees And Managers"
                                        FROM  Emp E1, Emp E2 WHERE E1.Mgr = E2.Empno;
SQL> SELECT E1.Ename||'Works For'|| E2.Ename "Employees And Managers"
                FROM Emp E1, Emp E2 WHERE(E1.Mgr = E2.Empno) AND  E1.Job = 'CLERK';
```

Cartesian Products

- The CARTESIAN PRODUCT is a Join Query, that Does Not Contains a Join Condition.
- During Cartesian Product Oracle Combines Each Row of One Table With Each Row of The Other.
- It Tends to Generate a Large Number of Rows And The Result is Rarely Useful.

```
SQL> SELECT Ename, Job, Dname  FROM Emp, Dept;
```

```
SQL> SELECT Ename, Job, Dname  FROM Emp, Dept  WHERE Job = 'MANAGER';
```

Non Equi Join

- It is a Join Condition That is Executed When no Column in One Table Corresponds Directly To a Column in The Other Table.
- The Data in The Tables in Directly Not Related But Indirectly or Logically Related Through Proper Values.

```
SQL> SELECT E.Ename , E.Sal , S.Grade  FROM Emp E, SalGrade S
                                    WHERE E.Sal BETWEEN S.Losal AND S.Hisal;
SQL> SELECT E.Ename, E.Sal, S.Grade  FROM Emp E, SalGrade S
                          WHERE (E.Sal >= S.LoSal AND E.Sal <= S.HiSal) AND S.Grade = 1;
```

Outer Joins

- An Outer Join Extends The Result of a Simple OR Inner Join.
- An OUTER Join Returns All Rows That Satisfy The Join Condition And Also Those Rows From One Table For Which No Rows From The Other Satisfy The Join Condition.
- To Perform An OUTER Join of Tables 'A' and 'B' and Returns All Rows From 'A', Apply The Outer Join Operator '(+)' to All Columns of Table 'B'.
- For all Rows in 'A' That Have no Matching Rows in 'B', Oracle Returns NULL For Any Select List Expressions Containing Columns of 'B'.

Syntax:

```
SQL> SELECT Table1.Column, Table2.Column  FROM Table1, Table2
                                    WHERE Table1.Column (+) = Table2.Column;
SQL> SELECT Table1.Column, Table2.Column  FROM Table1, Table2
                                    WHERE Table1.Column  = Table2.Column(+);
```

Rules And Restrictions:

- The (+) Operator Can Appear Only in The WHERE Clause.
- The (+) Operator Can Appear in The Context of The Left Correlation in The FROM Clause, and Can be Applied Only to a Column of a Table or View.
- If 'A' and 'B' Are Joined by Multiple Join Conditions, We Must Use The (+) Operator in All of These Conditions.
- The (+) Operator Can be Applied Only to a Column, Not To An Arbitrary Expressions.
- A Condition Containing The (+) Operator Cannot be Combined With Another Condition Using the OR Logical Operator.
- A Condition Cannot Use The IN Comparison Operator To Compare a Column Marked With The (+) Operator With an Expression.
- A Condition Cannot Compare Any Column Marked With The (+) Operator With a Sub Query.

```
SQL> SELECT E.Ename, D.Deptno, D.Dname  FROM Emp E, Dept D
                          WHERE E.Deptno (+) = D.Deptno  ORDER BY E.Deptno ;
SQL> SELECT E.Ename, D.Deptno, D.Dname FROM Emp E, Dept D
                WHERE E.Deptno(+) = D.Deptno  AND E.Deptno(+) = 10 ORDER BY E.Deptno;
SQL> SELECT E.Ename, D.Deptno, D.Dname FROM Emp E, Dept D
                WHERE E.Deptno = D.Deptno(+) AND E.Deptno(+) = 10 ORDER BY E.Deptno;
SQL> SELECT E.Ename Employee, NVL(M.Ename, 'Supreme Authority') Manager
                                  FROM Emp E, Emp M WHERE E.MGR = M.Empno(+)
```

Joining Data From More Than Two Table

- JOINS Can Be Established on More Than Two Tables.
- The Join is First Executed Upon The Two Most Relevant Tables And Then The Result is Applied Upon the Third Table.

```
SQL> SELECT C.Name, O.OrdID, I.ItemID, I.Itemtot, O.Total  FROM Customer C, Ord O, Item I
      WHERE C.CustID = O.CustID AND  O.OrdID = I.OrdID AND
                                                C.Name = 'TKB SPORT SHOP';
SQL> SELECT E.Ename, E.Deptno, M.Ename Manager, M.Deptno FROM Emp E, Dept D, Emp M
                                  WHERE E.MGR = M.Empno AND E.Deptno = D.Deptno;
```

```
SQL> SELECT E.Ename EName,  Dname,  E.Sal ESal,  E.Grade EGrade, M.Sal MSal,
          SM.Grade MGrade,  FROM Emp E, Dept D, Emp M, SalGrade SE, SalGrade SM
            WHERE E.Deptno = D.Deptno  AND E.MGR = M.Empno AND  E.Sal BETWEEN
              SE.LoSal AND E.HiSal AND M.Sal BETWEEN SM.LoSal AND SM.HiSal
```

Categories of Joins

Oracle Proprietary Joins (8i And Prior)

- Equi Join
- Non-Equi Join
- Outer Join
- Self Join

ANSI SQL : 1999 Compliant Joins

- Cross Joins
- Natural Joins
- Using Clause
- Full OR Two Sided Outer Joins
- Arbitrary Join Conditions For Outer Joins

ISO OR ANSI Joins

Cross Join

```
SQL> SELECT Ename, Dept.Deptno, Dname, Loc FROM Emp CROSS JOIN Dept
                                                WHERE Emp.Deptno = Dept.Deptno;
```

Natural Join

```
SQL> SELECT Ename, Deptno, Dname, Loc FROM Emp NATURAL JOIN Dept
```

USING Clause

```
SQL> SELECT Ename, Deptno, Dname, Loc FROM Emp JOIN Dept USING(Deptno)
```

Inner Join

```
SQL> SELECT Ename, Dept.Deptno, Dname, Loc FROM Emp JOIN Dept
                                                ON Emp.Deptno = Dept.Deptno;
```

Self Join

```
SQL> SELECT  E.Ename Employee,     M.Ename Manager  FROM Emp E INNER JOIN Emp M
                                                ON(E.MGR = M.Empno)
```

Join on More Than Two Tables

```
SQL> SELECT Ename, Sal, Grade, Dept.Deptno, Dname FROM Emp JOIN Dept ON
       Emp.Deptno = Dept.Deptno JOIN SalGradeON Emp.Sal BETWEEN LoSal AND HiSal;
SQL> SELECT E.Ename, M.Ename, Sal, Grade, D.Deptno, Dname FROM Emp E INNER JOIN
            Dept D ON E.Deptno = D.Deptno INNER JOIN Emp MON E.Empno = M.MGR
              INNER JOIN SalGrade S  ON E.Sal BETWEEN LoSal AND HiSal
```

Right Outer Join

```
SQL> SELECT Ename, Dept.Deptno, Dname, Loc FROM Emp RIGHT JOIN Dept
                                                ON Emp.Deptno = Dept.Deptno;
```

Left Outer Join

```
SQL> SELECT Ename, Dept.Deptno, Dname, Loc FROM Emp LEFT JOIN Dept
                                                ON Emp.Deptno = Dept.Deptno;
```

FULL Join

```
SQL> SELECT Ename, Dept.Deptno, Dname, Loc FROM Emp FULL JOIN Dept
                                                ON Dept.Deptno = Emp.Deptno
SQL> SELECT Ename, Dept.Deptno, Dname, Loc FROM Emp FULL JOIN Dept
                                                ON Emp.Deptno = Dept.Deptno
```

Some Complications

```
SQL> SELECT  E.Ename Employee, M.Ename Manager FROM Emp E LEFT OUTER JOIN Emp M
                                                ON(E.MGR = M.Empno) ORDER BY 2
SQL> SELECT E.Ename, M.Ename, Sal, Grade, D.Deptno, Dname  FROM Emp E INNER JOIN
            Dept D  ON E.Deptno = D.Deptno  INNER JOIN Emp M ON E.Empno = M.MGR
                INNER JOIN SalGrade S  ON E.Sal BETWEEN LoSal AND HiSal
```

Sub Queries OR Nested Select OR Sub Select OR Inner Select

- A Sub Query Answers Multiple-Part Questions.
- A Sub Query in The WHERE Clause of a SELECT Statement is Called as NESTED SUBQUERY.
- A Sub Query in The FROM Clause of a SELECT Statement is Called As INILINE VIEW.
- A Sub Query Can Be Part of a Column, in The SELECT List.
- A Sub Query Can Contain Another Sub Query.
- Oracle Imposes no Limit on The Number of Sub Query Levels in The FROM Clause of The Top-Level Query.
- Within The WHERE Clause Upto 255 Sub Queries Can be Nested.
- To Make The Statements Easier For Readability, Qualify The Columns in a Sub Query With The Table Name or Table Alias.

Purpose of A Sub Query

- To Define The Set of Rows To Be Inserted Into The Target Table of An INSERT or CREATE TABLE Statement.
- To Define The Set of Rows To Be Included in a View OR a Materialized View in a CREATE VIEW or CREATE MATERIALIZED VIEW Statement.
- To Define One or More Values To Be Assigned To Existing Rows in An UPDATE Statement.
- To Provide Values For Conditions in a WHERE Clause, HAVING Clause, START WITH Clause of SELECT, UPDATE, and DELETE Statements.
- To Define a Table to be Created on By a Containing Query.

Sub Query Principle

- Solve a Problem By Combining The Two Queries, Placing One Query Inside The Other Query.
- The Inner Query or The Sub Query Returns a Value That is Used By The Outer Query Upon The Main Query.

Sub Query Usage

- They Are Practically Very Useful When We Need to SELECT ROWS From a Table With a Condition That Depends on The Data in The Table Itself.

Syntax

```
SQL> SELECT SelectList  FROM TableName WHERE ColumnName Operator (
                                        SELECT SelectList FROM TableName );
```

- The Expressional Operators in Sub Queries Can Be Categorized into
  - Single Row Operators → =, <>, <, >, >=, <=
  - Multiple Row Operators → IN, ANY, ALL

Types of Sub Queries:

Single Row Sub Query:

- These Queries Return Only One Row From The Inner SELECT Statement.

Multiple Row Sub Query:

- These Queries Return More Than One Row From The Inner SELECT Statement.

Multiple Column Sub Query:

- These Queries Return More Than One Column From The Inner SELECT Statement.

Guidelines To Follow…

- A Sub Query Must be Enclosed in Parenthesis.
- A Sub Query Must Appear on The Right Side of The Comparison Operator Only.
- Sub Queries Should Not Contain An ORDER BY CLAUSE.
- Only One ORDER BY Clause Can Be Implemented For The Total SELECT Statement.
- Two Classes of Comparison Operators Can be Used in Sub Queries They Are…
  - Single Row Operators.
  - Multiple Row Operators.

Let Us Start With Single Row Sub Queries

```
SQL> SELECT Ename, Sal, Job FROM Emp WHERE Sal >(SELECT Sal  FROM Emp
                                        WHERE Empno = 7566);
```

```
SQL> SELECT Ename, Sal, Job  FROM Emp WHERE Job = (SELECT Job FROM Emp
                                    WHERE Ename = UPPER('smith') )  ORDER BY Sal;
SQL> SELECT Empno, Ename, Hiredate, Sal FROM Emp WHERE  Hiredate >
       (SELECT Hiredate FROM Emp WHERE Ename = 'TURNER') ORDER BY Sal;
SQL> SELECT Empno, Ename, Sal, Job  FROM Emp WHERE Deptno = (SELECT Deptno
                                                FROM Dept WHERE Dname = 'SALES');
SQL> SELECT Empno, Ename, Sal, Comm, Sal + NVL( Comm, 0 ) FROM Emp WHERE
                        Deptno = (SELECT Deptno FROM Dept WHERE Loc = 'DALLAS');
```

Applying Group Functions in Sub Queries

- The Data From The Main Query Can Be Displayed By Using a Group Function in a Sub Query.
- As a Group Function Returns a Single Row, The Query Passes Through The Success State.
- The Inner Sub Query Should Not Have a GROUP BY Clause in This Scenario.

```
SQL> SELECT Ename, Job, Sal FROM Emp WHERE Sal = (SELECT MAX(Sal) FROM Emp);
SQL> SELECT Ename, Job, Sal FROM EmpWHERE Sal = (SELECT MIN(Sal)FROM Emp);
SQL> SELECT Ename, Job, Sal FROM EmpWHERE Sal > (SELECT AVG(Sal)FROM Emp);
SQL> SELECT Ename, Job, Sal FROM EmpWHERE Sal < (SELECT STDDEV(Sal)FROM Emp);
```

Applying HAVING Clause With Sub Queries

- A Sub Query Can Be Also Applied in HAVING Clause.
- The Oracle Server Executes The Sub Query, And The Results Are Returned Into The HAVING Clause of the Main Query.
- The Inner Query Need Not Use Any GROUP Functions in This Scenario.
- The Outer Queries HAVING Clause Contains GROUP Function.

```
SQL> SELECT Deptno, MIN(Sal) FROM Emp  GROUP BY Deptno  HAVING  MIN(Sal) >
                                    (SELECT MIN(Sal) FROM EmpWHERE  Deptno = 20);
SQL> SELECT Job, AVG(Sal) FROM Emp GROUP BY Job HAVING AVG(Sal) =
                                (SELECTMIN(AVG(Sal)) FROM Emp GROUP BY Job);
SQL> SELECT Job, AVG(Sal)  FROM Emp GROUP BY Job HAVING AVG(Sal) <
                                (SELECT MAX(AVG(Sal)) FROM Emp GROUP BY Job );
SQL> SELECT Job, AVG(Sal), TO_CHAR(AVG (Sal), 'L99,999.99') FROM Emp GROUP BY Job
       HAVING  AVG(Sal) < (SELECT MAX(AVG(Sal)) FROM Emp GROUP BY Deptno);
```

Sub Queries Returning More Than One Row

- The Sub Queries That Return More Than One Row Are Called as MULTIPLE ROW SUB QUERIES.
- In This Case a Multiple Row Operator Should Be Used.
- The Multiple Row Operators Expect One or More Values as Arguments.
- The Multiple Row Operators Are...
  - IN → Equal to Any Member in The List.
  - ANY → Compares Value to Each Value Returned by Sub Query.
  - ALL → Compares Value to Every Value Returned by the Sub Query.

```
SQL> SELECT Ename, Sal, Deptno FROM Emp WHERE Sal IN(SELECT MIN(Sal)
                                                FROM Emp GROUP BY Deptno);
SQL> SELECT Ename, Sal, Deptno FROM Emp WHERE Sal IN(SELECT MAX(Sal)
                        FROM Emp  GROUP BY Deptno);
SQL> SELECT Ename, Sal , Deptno , Job FROM Emp WHERE Sal IN(SELECT MAX(Sal)
                                                FROM Emp  GROUP BY Job);
```

ANY Operator

```
SQL> SELECT Empno, Ename, Job FROM Emp WHERE Sal < ANY(SELECT Sal  FROM Emp
                                                WHERE Job = 'CLERK' );
SQL> SELECT Empno, Ename, Job, Sal FROM Emp WHERE Sal < ANY(SELECT Sal
                FROM Emp WHERE Deptno = 20 AND Job <> 'CLERK');
```

Note: <ANY Means Less Than The Maximum Value in The List.

```
SQL> SELECT Empno, Ename, Job  FROM Emp  WHERE Sal >ANY(SELECT Sal  FROM Emp
                                WHERE Job = 'CLERK');
```

Note: >ANY Means More Than The Minimum Value in The List.

```
SQL> SELECT Empno, Ename, Job  FROM Emp WHERE  Sal =ANY(SELECT Sal FROM Emp
                                WHERE Job = 'CLERK');
```

Note: =ANY It is Equivalent to IN Operator

ALL Operator

```
SQL> SELECT Empno, Ename, Job, Sal  FROM Emp WHERE Sal >ALL(SELECT AVG ( Sal )
                                                    FROM Emp GROUP BY Deptno);
```

Note: >ALL → It Means More Than The Maximum In The List.

```
SQL> SELECT Empno, Ename Job, Sal FROM Emp WHERE Sal <ALL(SELECT  AVG(Sal)
                                                    FROM Emp  GROUP BY Deptno);
```

Note: <ALL → It Means Less Than The Minimum In The List.

Sub Queries Returning Multiple Columns

- In Sub Queries Multiple Columns Can Be Compared in The WHERE Clause, By Writing a Compound WHERE Clause Using Logical Operators.
- Multiple Column Sub Queries Enable us to Combine the Duplicate WHERE Condition into a Single WHERE CLAUSE.

Syntax

```
SQL> SELECT Column1, Column2,... FROM TableName  WHERE
        (Column a, Column b ,...) IN(SELECT Column a, Column b,... FROM TableName
                                                    WHERE Condition );
```

- The Column Comparisons in a Multiple Column Sub Query Can Be
  - Pair Wise Comparison.
  - Non Pair Wise Comparison.
- In Pair Wise Comparisons Each Candidate Row in The SELECT Statement Must Have Both The Same Values Associated With Each Column in The Group.
- The Non Pair Wise Comparison is Also Called Cross Product,. We Can Use a WHERE Clause With Multiple Conditions.
- In Non Pair Wise Comparison, The Candidate Row Must Match The Multiple Conditions in The WHERE Clause But The Values Are Compared Individually.

Pairwise Comparision OR Compound WHERE Clause Based SubQuery

```
SQL> SELECT OrdID, ProdID, Qty FROM Item WHERE (ProdID, Qty) IN(SELECT ProdId, Qty
                                FROM Item WHERE OrdID = 605 ) AND OrdID < > 605;
```

Non Pairwise Comparision OR Component WHERE Clause Based SubQuery

```
SQL> SELECT  OrdID, ProdID, Qty  FROM Item WHERE ProdID IN (SELECT ProdID
        FROM ItemWHERE OrdID = 605) AND  Qty IN (SELECT Qty  FROM Item
                                        WHERE OrdID = 605) AND OrdID <> 605;
```

Handling NULL Values in Sub Queries:

- If One of The Values Returned By The Inner Query is NULL Value, Then The Entire Query Returns NO ROWS.
- All CONDITIONS That Compare a NULL Value Result in a NULL
- Whenever a NULL Could be Part of a Sub Query, it is Better Not to Use NOT IN Operator As it is Equivalent to !=ALL Operator.

```
SQL> SELECT E.Ename FROM Emp E WHERE E.Empno IN (SELECT  M.Mgr  FROM Emp M);
```

Applying Sub Query in From Clause

- A Sub Query in The From Clause is Equivalent To a View.
- The Sub Query in The From Clause Defines a Data Source For That Particular SELECT Statement And Only That SELECT Statement.

```
SQL> SELECT E.Ename, E.Sal , E.Deptno, E1.SalAvg FROM Emp E, (SELECT Deptno, AVG(Sal)
        SalAvg FROM Emp GROUP BY Deptno) E1 WHERE E.Deptno = E1.Deptno
                                                AND E.Sal > E1.SalAvg;
SQL> SELECT T1.Deptno, Dname, Staff  FROM Dept T1, (SELECT Deptno, COUNT(*) AS Staff
     FROM Emp GROUP BY Deptno ) T2 WHERE T1.Deptno = T2.Deptno AND  Staff >= 5;

SQL> SELECT Deptno, SUM(Sal),  SUM(Sal)/Tot_Sal * 100 "Salary%" FROM Emp,
        (SELECT SUM(Sal) Tot_Sal  FROM Emp) GROUP BY Deptno, Tot_Sal;
SQL> SELECT E.EmpCount, D.DeptCount FROM (SELECT COUNT(*) EmpCount
          FROM Emp) E,  (SELECT COUNT(*) DeptCount  FROM Dept) D;
SQL> SELECT E.EmpCount, D.DeptCount, S.GradeCnt, E.EmpCount + D.DeptCount + S.GradeCnt
        TotalRecCnt FROM  (SELECT COUNT(*) EmpCount  FROM Emp) E,
         (SELECT COUNT(*) DeptCount FROM Dept ) D, (SELECT COUNT(*) GradeCnt
                FROM SalGrade) S
SQL> SELECT A.Deptno "Department Number",(A.NumEmp / B.TotalCount )*100 "%Employees",
     (A.SalSum / B.TotalSal ) * 100 "%Salary" FROM (SELECT  Deptno,
        COUNT(*) NumEmp,  SUM(Sal) SalSum FROM Emp GROUP BY Deptno) A,
          (SELECT COUNT(*) TotalCount,  SUM(Sal) TotalSal  FROM Emp) B;
```

Sub Select Statements

- These Are SELECT Statements Declared as Part of The SELECT List.

```
SQL> SELECT Ename, Sal,(SELECT AVG(Sal) FROM Emp) "Organization Average" FROM Emp;
SQL> SELECT Ename, Sal, (SELECT MAX(Sal) FROM Emp) "Organization Maximum",
        (SELECT MIN(Sal) FROM Emp) "Organization Minimum" FROM Emp;
```

Correlated Sub Queries

- It is Another Way of Performing Queries Upon The Data With a Simulation of Joins.
- In This The Information From the Outer SELECT Statement Participates As a Condition in The INNER SELECT Statement.

Syntax

```
SQL> SELECT SelectList  FROM Table1 F_Alias1 WHERE Expr.Operator
        (SELECT SelectList  FROM Table2 F_Alias2 WHERE F_Alias1.Column
          OPERATOR F_Alias2.Column );
```

Steps Performed

- First the Outer Query is Executed.
- Passes The Qualified Column Value to the Inner Queries WHERE Clause.
- Then The Inner Query or Candidate Query is Executed, And The Result is Passed To The Outer Queries WHERE Clause.
- Depending on The Supplied Value The Condition is Qualified For The Specific Record.
- Successful Presented Else Suppressed From Display

```
SQL> SELECT Empno, Ename, E.Deptno, Sal, MGR FROM Emp E WHERE E.Sal >ANY
        (SELECT M.Sal FROM Emp M WHERE M.Empno  = E.MGR);
SQL> SELECT Deptno, Dname  FROM  Dept D WHERE  EXISTS (SELECT * FROM Emp E
                WHERE D.Deptno = E.Deptno);
SQL> SELECT Deptno, Dname  FROM  Dept D WHERE  NOT EXISTS (SELECT *  FROM Emp E
                                        WHERE D.Deptno = E.Deptno);
SQL> SELECT E.Ename FROM Emp E WHERE EXISTS (SELECT *  FROM Emp E1
                                        WHERE E1.Empno = E.MGR);
SQL> SELECT E.Ename FROM Emp E WHERE NOT EXISTS (SELECT *  FROM Emp E1
                                        WHERE E1.Empno = E.MGR);
SQL> SELECT E.Ename FROM Emp EWHERE EXISTS (SELECT * FROM Emp E1
                                        WHERE E1.Mgr = E.Empno);
SQL> SELECT E.Ename FROM Emp E WHERE NOT EXISTS (SELECT * FROM Emp E1
                                                WHERE E1.Mgr = E.Empno);
```

Hierarchical OR Recursive Queries

- Hierarchical Queries Are Queries That Are Executed Upon Tables That Contain Hierarchical Data.
- To Execute The Hierarchical Queries, We Need The Following Clauses.

START WITH
- It Specifies The Root Rows of The Hierarchy.

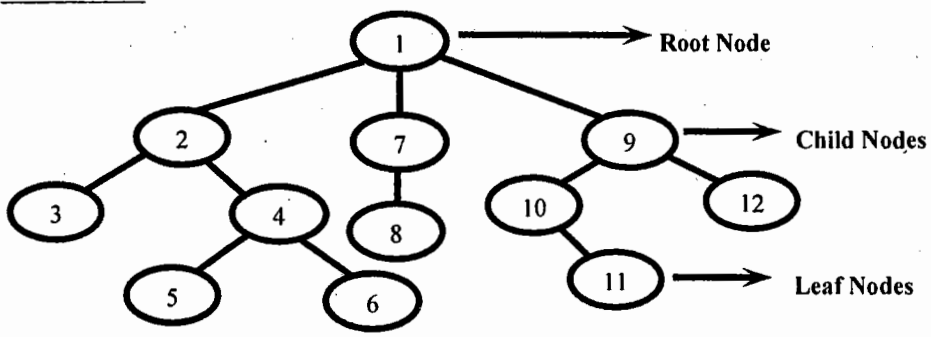
CONNECT BY
- It is Used To Specify The Relationship Between Parent Rows And Child Rows of The Hierarchy.

WHERE
- It is Used To Restrict The Rows Returned By The Query Without Affecting Other Rows of The Hierarchy.

Steps Followed By Oracle
- ORACLE Selects The ROOT ROW(s) of The Hierarchy, Which Satisfy The Condition of The START WITH Clause.
- Then ORACLE Selects The Child Rows of Each ROOT ROW.
- Each Child Row Must Satisfy The Condition of The CONNECT BY Clause, With Respect to One of The ROOT ROWS.
- ORACLE Selects Successive Generations of Child Rows By Identifying the Relation in The CONNECT BY Clause.
- ORACLE Selects Children By Evaluating The CONNECT BY Condition With Respect To The Current Parent Row Selected.
- If The Query Contains a WHERE Clause, ORACLE Removes All Rows From The Hierarchy That Do Not Satisfy The Condition of The WHERE Clause.

General Representation

Restrictions



- They Cannot Be Used To Perform Joins.
- They Cannot Select Data From a View, Whose Query Performs a Join.
- If ORDER BY Clause is Used, Then The Rows Are Returned As Per The Specification in The ORDER BY Clause.
- To Define Hierarchical Queries Properly we Must Use The Following Clauses.
  - START WITH Clause.
  - CONNECT BY Clause.

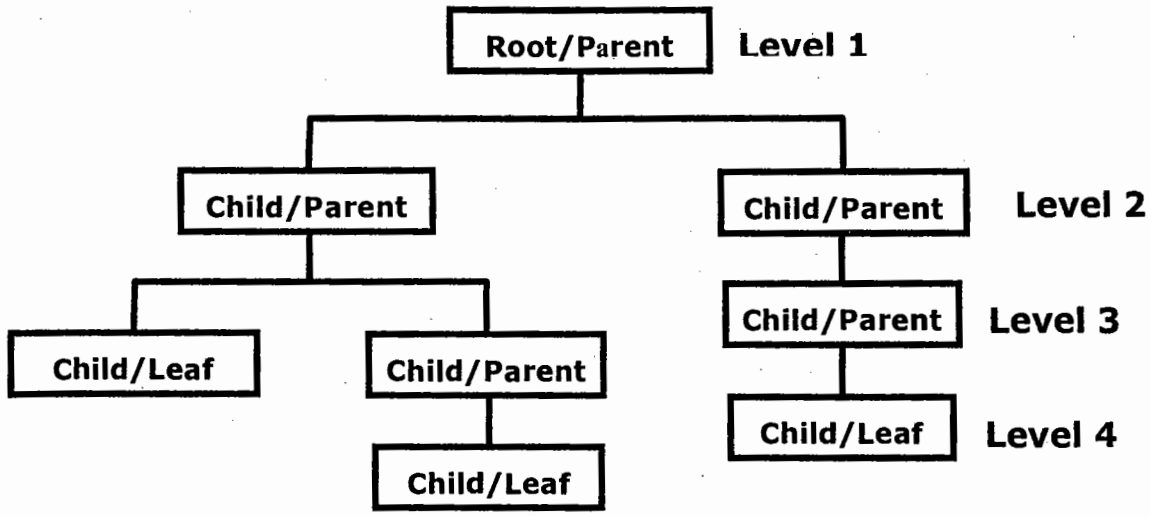
START WITH Clause
- It Identifies The Row(s) To Be Used As The ROOT(s) of a Hierarchical Query.
- It Specifies a Condition That The Roots Must Specify.
- If START WITH is Omitted, Oracle Uses All Rows in The Table As ROOT Rows.
- A START WITH Condition Can Contain a Subquery.

CONNECT By Clause
- This Clause Specifies The Relationship Between Parent and Child Rows, in a Hierarchical Query.
- This Clause Contains a Condition That Defines a Relationship.
- This Condition Can Be Any Condition As Defined By The Syntax Description.
- Within The Condition, Some Part of The Condition Must Use The PRIOR Operator, Which Refers to The Parent Row.
- The Format of PRIOR Operators Is

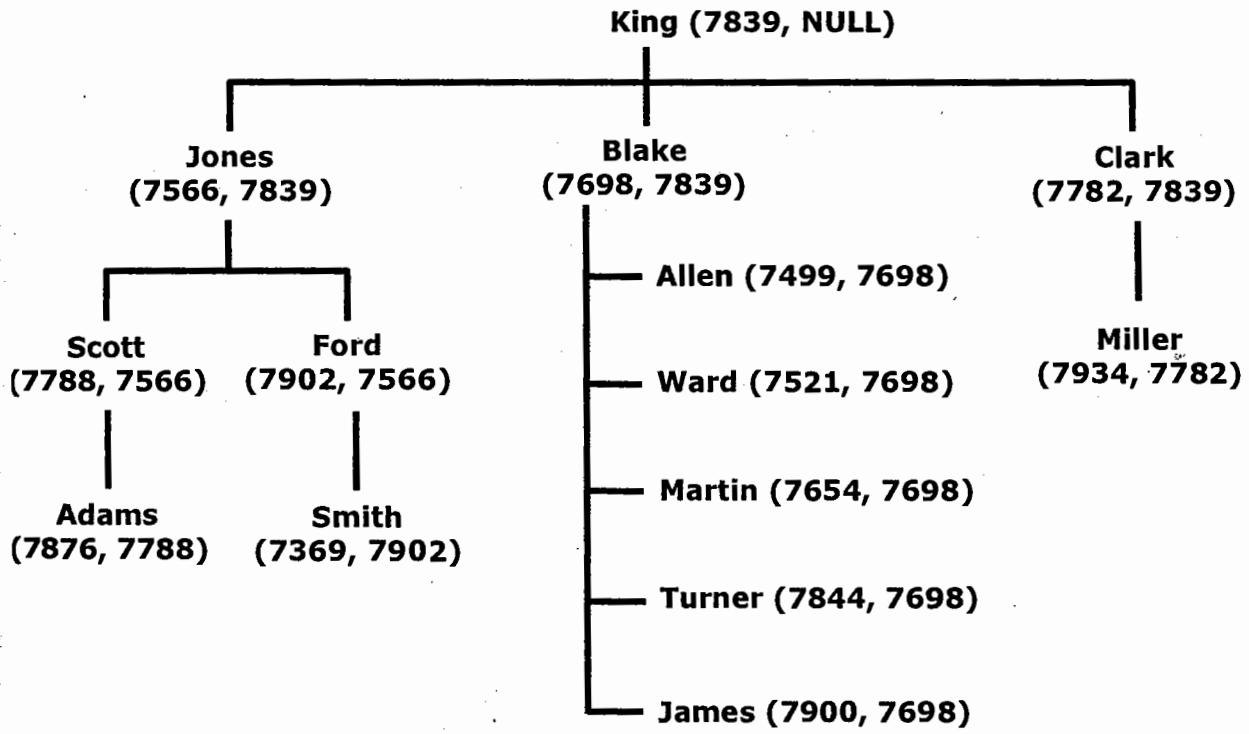
- PRIOR Expr ComparisonOperator Expr.
- Expr ComparisonOperator PRIOR Expr.

- The Clause Can Contain Other Conditions To Further Filter The Rows Selected By The Query.
- It Cannot Contain a Sub Query.

General Structure of Hierarchical Tree



Hierarchical Representation of Emp Table




```
SQL> SELECT Ename, Empno, Mgr, Job FROM Emp CONNECT BY PRIOR  Empno = MGR;
SQL> SELECT Ename, Empno, Mgr, Job FROM Emp START WITH Job = 'PRESIDENT'
                                        CONNECT BY  PRIOR  Empno = MGR;
SQL> SELECT Ename, Empno, Mgr , Job FROM Emp START WITH Ename = 'KING'
                                          CONNECT BY PRIOR Empno = MGR;
SQL> SELECT Ename, Empno, Mgr, Job, Sal FROM Emp START WITH Sal = 5000
                                          CONNECT BY PRIOR Empno = MGR;
SQL> SELECT Ename, Empno, Mgr, Job, Sal FROM Emp START WITH Sal =
                        (SELECT MAX(Sal) FROM Emp) CONNECT BY PRIOR Empno = MGR;
SQL> SELECT Ename, Empno, Mgr, Job, Sal FROM Emp START WITH  Sal =
            (SELECT MAX(Sal)  FROM Emp WHERE Deptno =
          (SELECT Deptno  FROM Dept WHERE Dname = 'ACCOUNTING' ) ) CONNECT BY
```

PRIOR Empno = MGR;

SQL> SELECT Ename, Empno, Mgr, Job, Sal FROM Emp START WITH Ename = 'KING'
CONNECT BY PRIOR Empno = MGR AND Job = 'MANAGER';

New Features in Hierarchical Queries Oracle 10g

1. New Operator
   - CONNECT_BY_ROOT
2. New Pseudo Columns
   - CONNECT_BY_ISCYCLE
   - CONNECT_BY_ISLEAF
3. New Function
   - SYS_CONNECT_BY_PATH (Oracle9i)
4. New Keywords
   - NOCYCLE
   - SIBLINGS (Oracle9i)

CONNECT_BY_ROOT Operator

- CONNECT_BY_ROOT is a UNARY Operator That is Valid Only in Hierarchical Queries.
- We Should Qualify a Column With This Operator, Then Oracle Returns The Column Value Using Data From The ROOT Row.
- It Extends The Functionality of The CONNECT BY [PRIOR] Condition of Hierarchical Queries.

Restriction

- We Cannot Specify This Operator in The START WITH Condition or The CONNECT BY Condition.

SQL> SELECT ENAME Name,
CONNECT_BY_ROOT(Ename) Boss FROM EMP START WITH EMPNO = 7839
CONNECT BY PRIOR EMPNO = MGR

SYS_CONNECT_BY_PATH Function

- The Function Returns The Path of a Column Value From Root To Node, With Column Values Separated By 'Char' For Each Row Returned By CONNECT BY Condition.
- Can Work on Any Datatype CHAR, VARCHAR2, NCHAR, or NVARCHAR2.

SQL> SELECT Ename, SYS_CONNECT_BY_PATH(Ename, '/') "Path"FROM Emp
START WITH Ename = 'KING' CONNECT BY PRIOR Empno = MGR;

NOCYCLE Keyword

- Cycles Are Not Allowed in a True Tree Structure. But Some Hierarchical Data May Contain Cycles.
- In a Hierarchical Structure, if a Descendant is Also an Ancestor, It is Called a CYCLE.
- To Allow The "START WITH ... CONNECT BY ... PRIOR" Construct To Work Properly Even if Cycles Are Present in The Data NOCYCLE is Used.
- The NOCYCLE Parameter in The CONNECT BY Condition Causes ORACLE to Return the Rows in Spite of The Recursive Loop.

SQL> SELECT Ename, SYS_CONNECT_BY_PATH(Sal, '/') "Path" FROM Emp
START WITH Ename = 'KING' CONNECT BY NOCYCLE PRIOR Empno = MGR;

SIBLINGS Keyword

- The Keyword is Valid Only When We Specify The Hierarchical Query Using CONNECT BY Clause.
- ORDER SIBLINGS BY Clause Preserves Any Ordering Specified in The Hierarchical Query Clause.
- The ORDER BY Clause Finally Gets Applied on The Query.
- ORDER SIBLINGS BY Clause is Generally Used When We Want To Order Rows of Siblings of The Same Parent.

SQL> SELECT Ename, Empno, MGR FROM Emp START WITH Empno = 7839
CONNECT BY PRIOR Empno = MGR;

SQL> SELECT Ename, Empno, MGR FROM Emp START WITH Empno = 7839
CONNECT BY PRIOR Empno = MGR ORDER BY Ename;

SQL> SELECT Ename, Empno, MGR FROM Emp START WITH Empno = 7839
CONNECT BY PRIOR Empno = MGR ORDER SIBLINGS BY Ename;

Taking The Advantage of Pseudo Columns in Oracle

Pseudo Column

- Pseudo Columns Behave Like a Table Column, But is Not Actually Stored in a Table.
- Upon Pseudo Columns Only SELECT Statements Can Be Implemented, But INSERT, UPDATE or DELETE Cannot be Implemented.
- The Available PSEUDO COLUMNS Are...
  - CURRVAL
  - NEXTVAL
  - LEVEL
  - ROWID
  - ROWNUM

CURRVAL And NEXTVAL Pseudo Columns

- These Pseudo Columns Are Applied Upon The SEQUENCE Schema Object.
- CURRVAL Returns The CURRENT Value of a Sequence.
- NEXTVAL INCREMENTS The Sequence And Returns the NEXT VALUE.
- The CURRVAL And NEXTVAL can be used Only in ...
  - The SELECT List of a SELECT Statement.
  - The VALUES Clause of an INSERT Statement.
  - The SET Clause of an UPDATE Statement.

Restrictions

- The CURRVAL and NEXTVAL Cannot Be Used in...
  - A Sub Query.
  - A View's Query or SNAPSHOT's Query.
  - A SELECT with the DISTINCT Option.
  - A SELECT with a GROUP BY or ORDER BY Clause.
  - A SELECT Statement with UNION, INERSECT, MINUS SET Operators.
  - The WHERE Clause of a SELECT.
  - The DEFAULT Option in CREATE TABLE or ALTER TABLE Statement.
  - The Condition of a CHECK Constraint.

Syntax

- SEQUENCENAME.CURRVAL → Returns the Current Value of the Sequence.
- SEQUENCENAME.NEXTVAL → Increments the Sequence Value by the Declared Specification.

SEQUENCE Schema Object

- A SEQUENCE is a Schema Object That Can Generate UNIQUE Sequential Values.
- The SEQUENCE Values Are Often Used For PRIMARY KEY's and UNIQUE KEY's.
- To Refer To The CURRENT or NEXT Value of a SEQUENCE in The SCHEMA of Another User, The Following Privileges Should Be Available...
  - SELECT OBJECT PRIVILEGE
  - SELECT ANY SEQUENCE
- For SEQUENCES in Other Schema The QUALIFYING SYNTAX is
  - SCHEMANAME.SEQUENCENAME.CURRVAL
  - SCHEMANAME.SEQUENCENAME.NEXTVAL
- To Refer to The Value of a SEQUENCE on a REMOTE Database, The SEQUENCE Should Be Qualified With a Complete or Partial Name of the Database Link.
  - SCHEMANAME.SEQUENCENAME.CURRVAL@DBLINK
  - SCHEMANAME.SEQUENCENAME.NEXTVAL@DBLINK
- In a Single SELECT Statement, All Referenced Sequences, LONG Columns, Updated Tables, and Locked Tables, Must Be Located on The Same Database.
- WHEN a SEQUENCE IS Created, We Can Define its INITIAL VALUE And The INCREMENT Between Its Values.
- The First Reference To The NEXTVAL Returns The SEQUENCES Initial Value.

- Before The CURRVAL Can Be Used For a SEQUENCE in a Session, First The SEQUENCE Should Be Incremented With NEXTVAL.
- A SEQUENCE Can Be Incremented Only Once in a Single SQL Statement.
- A SEQUENCE Can Be Accessed By Many Users Concurrently With No WAITING, No LOCKING.
- CURRVAL and NEXTVAL Should Be Qualified With The Name of The Sequence.

Creating Sequences

Purpose

- An Object From Which Multiple Users May Generate Unique Integers.
- CAN Be Used to Generate PRIMARY KEY Values Automatically.

Syntax

```
CREATE SEQUENCE SequenceName
INCREMENT BY Integer
START WITH Integer
MAXVAL Integer/NOMAXVALUE
MINVAL Integer/NOMINVALUE
CYCLE/NOCYCLE
CACHE Integer/NOCACHE
ORDER/NOORDER;
```

- SEQUENCE Can be Either Incremented Sequence OR Decremented Sequence

INCREMENT BY Clause

- Specifies The Interval Between The Sequence Numbers.
- Value Can Be Positive or Negative, But Cannot Be 0.
- If The Value is Positive It Is Incremented Sequence Else it is Decremented Sequence.
- If Omitted Defaults To Increment By 1.

MINVALUE Clause

- Specifies The Sequence's Minimum Value.

NOMINVALUE Clause

- Specifies a Minimum Value of 1 For An Ascending Sequence OR –(10)26 For Descending Sequence.

MAXVALUE Clause

- Specifies The Maximum Value That Can Be Generated.

NOMAXVAULE Clause

- Specifies a Maximum Value of 1027 For Ascending Sequence OR –1 For Descending Sequence.

CYCLE Clause

- Specifies The Sequence Will Continue To Generate Values After Reaching Either Maximum or Minimum Value.

NOCYCLE Clause

- Specifies The SEQUENCE Cannot Generate More Values After The Targeted Limit.

CACHE Clause

- Specifies The Pre-Allocation of SEQUENCE Numbers, The Minimum is 2.

NOCACHE Clause

- Specifies The Values of a SEQUENCE Are Not Pre-Allocated.

ORDER Clause

- Guarantees The Sequence Numbers To Be Generated in The Order of Request.

NO ORDER Clause

- Does Not Guarantee The Sequence Number With Order.

Note

- If The Above Parameters Are Not Specified By Default
  - START WITH Will Be 1.
  - INCREMENT BY WILL BE POSITIVE 1.
  - SEQUENCE IS NOCYCLE.
  - The CACHE Value Will Be 20.

- SEQUENCE IS ORDER Sequence.

Illustrations

```
SQL> CREATE TABLE Sample( SampID  NUMBER(4) Constraint SampID_PK PRIMARY  KEY,
                          SampName VARCHAR2(25), SampDate DATE );
```

Creation of Incremental Sequence

```
SQL> CREATE SEQUENCE SampleSeq
        INCREMENT By 1
        START WITH 0
        MINVALUE 0
        MAXVALUE 5
        NOCACHE
        NOCYCLE;
```

Activating And Attaching The Sequence To a Table

```
SQL> INSERT INTO
        Sample(SampID, SampName,  SampDate)
        VALUES(SampleSeq.NEXTVAL, 'SAMPLE', '31-AUG-05');
```

Creating A Sequence With CYCLE

```
SQL> CREATE SEQUENCE SampleSeq
        INCREMENT By 1
        START WITH 0
        MINVALUE 0 MAXVALUE 5
        NOCACHE
        CYCLE;
```

Creation of Decremental Sequence

```
SQL> CREATE SEQUENCE SampleSeq
        INCREMENT BY -1
        START WITH 5
        MAXVALUE 5
        MINVALUE 0
        NOCACHE
        NOCYCLE;
```

Modifying a Sequence

- The ALTER Command Can Be Used To Change The Present Status of a SEQUENCE.
- The ALTER SEQUENCE Command Can Be Used To Change…
  - Increment Value
  - Maximum Value.
  - Minimum Value
  - Cycle Option
  - Cache Option

Syntax

```
SQL> ALTER SEQUENCE SequenceName
        [INCREMENT BY n]
        [{MAXVALUE n/NOMAXVALUE}]
        [{MINVALUE n/NOMINVALUE}]
        [{CYCLE/NOCYCLE}]
        [{CACHE n/NOCACHE}];
```

Illustration

```
SQL> ALTER SEQUENCE SampleSeq
        MAXVALUE 10
        CACHE
        NOCYCLE;
```

Guidelines For Altering A Sequence

- The ALTER Privilege Should Be Available.

- Only The Future Sequence Numbers Are Affected By The ALTER SEQUENCE Statement.
- The START WITH Option Cannot Be Changed Using ALTER SEQUENCE.
- To Change The START WITH. Option, Drop The SEQUENCE And Then Recreate The SEQUENCE.
- Some Validation Performed, i.e., A NEW MAXVALUE Cannot Be Imposed That is Less Than The Current SEQUENCE Number.

Viewing the Current Value of a Sequence

```
SQL> SELECT SampleSeq.CURRVAL
        FROM DUAL;
```

Dropping An Existing Sequence

- A SEQUENCE Can Be DROPPED At Any Time.
- Once Removed, The SEQUENCE Can No Longer Be Referenced.

```
SQL> DROP SEQUENCE SampleSeq;
```

Confirming Sequences

- All SEQUENCES That Have Been Created Are Documented in The Data Dictionary.
- The Data Dictionary in Which The Information of SEQUENCES Are Stored is USER_OBJECTS.
- The Settings of The SEQUENCE Can Be Confirmed By SELECTING on USER_SEQUENCES Catalog.

```
SQL> SELECT SEQUENCE_NAME,
        MIN_VALUE, MAX_VALUE,
        INCREMENT_BY, LAST_NUMBER
        FROM User_Sequences;
```

Level Pseudo Column

This Pseudo Column Return 1 For ROOT Node, 2 For a Child of a ROOT And So on.

- Child → Any Non Root Node.
- Root → Highest Node within an Inverted Tree.
- Parent → Any Node / Row That Has Children.
- Leaf → Any Node Without Children.

- To Establish The Hierarchical Relationship With LEVEL We Need.
  - START WITH Clause.
  - CONNECY BY Clause.

```
SQL> SELECT Ename, Job, MGR, Level FROM Emp;
SQL> SELECT LPAD ( ' ', 2 * (LEVEL – 1)) Org_Level, Ename, Empno, Mgr, Job  FROM Emp
                    START WITH Job = 'PRESIDENT'  CONNECT BY PRIOR Empno = MGR;
SQL> SELECT  LPAD(' ' ,2 * (LEVEL – 1))||Ename Org_Chart,  Empno, MGR, Job
        FROM Emp START WITH Job = 'PRESIDENT' CONNECT BY PRIOR Empno = MGR;
SQL> SELECT  LPAD (' ', 2 * (LEVEL – 1))||Ename Org_Chart, Empno, MGR, Job , Sal
        FROM Emp WHERE Job != 'ANALYST' START WITH Job = 'PRESIDENT'
                                        CONNECT BY PRIOR Empno = MGR;
SQL> SELECT LPAD(' ', 2 * (LEVEL – 1))||Ename Org_Chart, Empno, MGR, Job, Sal
        FROM Emp START WITH Job = 'PRESIDENT' CONNECT BY PRIOR Empno = MGR
        AND  LEVEL <= 2;
```

Selecting Nth Highest Value From Table

Syntax

```
SQL> SELECT LEVEL, MAX(ColName) FROM TableName  WHERE LEVEL = &LEVELNO
                        CONNECT BY PRIOR ColName > ColName GROUP BY LEVEL;
```

Illustration

```
SQL> SELECT LEVEL, MAX(Sal) FROM EMP WHERE LEVEL = &LEVELNO
                        CONNECT BY PRIOR Sal > Sal GROUP BY LEVEL;
```

Selecting Nth Lowest Value From Table

Syntax

```
SQL> SELECT LEVEL, MIN(ColName) FROM TableName WHERE LEVEL = &LEVELNO
        CONNECT BY PRIOR ColName < ColName GROUP BY LEVEL;
```

Illustration

```
SQL> SELECT LEVEL, MIN(Sal) FROM EMP WHERE LEVEL = &LEVELNO
        CONNECT BY PRIOR Sal < Sal  GROUP BY LEVEL;
```

ROWNUM Pseudo Column

- For Each Row Returned By a Query, The ROWNUM Pseudo Column Returns a Number Indicating The Order in Which Oracle Selects The Rows From a Set of Joined Rows or Non Joined Rows.
- The First Row Selected Has a ROWNUM of 1, The Second Has 2. And So On...
- The ROWNUM Can Be Used To Limit The Number of Rows Returned By The Query.
- When ORDER BY Clause Follows a ROWNUM, The Rows Will Be Re-Ordered By ORDER BY Clause .
- If ORDER BY Clause is Embedded in a Sub Query And ROWNUM Condition is Placed in The TOP_LEVEL Query, Then The ROWNUM Condition Can Be Forced To Get Applied After The Ordering of The Rows.
- Conditions Testing For ROWNUM Values Greater Than a Positive Integer Are Always FALSE.

```
SQL> SELECT LPAD(' ' , ROWNUM, '*' ) FROM Emp;
SQL> SELECT ROWNUM, Ename, Sal FROM Emp;
```

Querying For Top 'N' Records

- We Can Ask For Nth Largest OR Smallest Values of a Column.
- Never Use ROWNUM And ORDER BY Clause Together As Oracle First Fetches The Rows According to ROWNUM And Then Sort's The Found Rows.
- From Oracle 8i , ORDER BY Clause Can Be Used in INLINE VIEWS.

```
SQL> SELECT ROWNUM, Ename, Sal  FROM Emp WHERE ROWNUM < 6
        ORDER BY Sal DESC; --Wrong Way
SQL> SELECT *  FROM (SELECT *  FROM Emp  ORDER BY Sal DESC )
        WHERE ROWNUM < 6; --Proper Way
```

ROWID Pseudo Column

- This Pseudo Column Returns a ROW's Address For Each Row Stored in The Database.
- ROWID Values Contain Information Necessary To Locate a The Physical Area of The Data Base Row.
  - The Row Belongs To Which Data Block in the Data File.
  - The Row Belongs To Which Row in The Data Block (First Row is 0)
  - The Row Belongs To Which Data File (First File is 1)
- The Rows in Different Tables That Are Stored Together in The Same Cluster Can Have The Same ROWID.
- The Date Type of The Values Belonging to The ROWID Are of ROWID Data Type.

Uses of ROWID Values

- ROWID is The Fastest Means of Accessing a Single Row From Data Base.
- ROWID Can Show How a Tables Row's Are Physically Stored.
- ROWID's Are UNIQUE Identifiers For a Row in a Table.
- A RowID Can Never Change During The Life Time of Its Row.
- ROWID's Should Not Be Assigned As PRIMARY KEY's As There is a Chance of ROWID To Change When The Database is EXPORTED or IMPORTED.
- When a Row is DELETED, ORACLE May Reassign Its ROWID To a New Row That is Inserted.
- The ROWID Can Never Be INSERTED, UPDATED and DELETED Manually.
- The ROWID Pseudo Column Can Be Used in SELECT and WHERE Clauses.

```
SQL> SELECT ROWID, Ename, Job FROM EmpWHERE Empno = 20;
SQL> SELECT Ename, Sal, Job FROM Emp  WHERE ROWID = 'AACQQAACAAAAEHAAA';
SQL> SELECT Ename, Sal, Job FROM Emp WHERE ROWID < 'AACQQAACAAAAEHAAA';
SQL> SELECT B.Sal, Sum(A.Sal) "Cum Sal"  FROM Emp A, Emp B  WHERE
                                    A.ROWID < = B.ROWIDGROUP By B.ROWID, B.Sal;
SQL> SELECT B.Sal, Sum(A.Sal) "Cum Sal"  FROM Emp A, Emp B  WHERE
                        A.ROWID < = B.ROWID GROUP BY B.ROWID, B.Sal;
```

```
SQL> SELECT B.Ename, B.Job, B.Sal, Sum(A.Sal) "Cum Sal"  FROM Emp A, Emp B
                    WHERE A.ROWID < = B.ROWID GROUP By B.ROWID, B.Sal;
```

New Pseudo Columns(Oracle 10g)

- CONNECT_BY_ISCYCLE
- CONNECT_BY_ISLEAF

CONNECT_BY_ISCYCLE Pseudocolumn

- The CONNECT_BY_ISCYCLE Pseudo Column Returns 1 If The Current Row Has A Child Which is Also Its Ancestor Otherwise It Returns 0.
- We Can Specify CONNECT_BY_ISCYCLE Only If We Have Specified The NOCYCLE Parameter of The CONNECT BY Clause.
- NOCYCLE Enables Oracle To Return The Results of A Query That Would Otherwise Fail Because of A CONNECT BY LOOP in The Data.

```
SQL> SELECT Ename, CONNECT_BY_ISCYCLE "Cycle", LEVEL,
                SYS_CONNECT_BY_PATH(Sal, '/') SalPath FROM Emp START WITH
                        Ename = 'KING' CONNECT BY NOCYCLE  PRIOR Empno = MGR;
```

CONNECT_BY_ISLEAF Pseudocolumn

- The CONNECT_BY_ISLEAF Pseudo Column Returns 1, if The Current Row is A Leaf of The Tree Defined By The CONNECT BY Condition, Else it Returns 0
- This Information Indicates Whether A Given Row Can Be Further Expanded To Show More of The Hierarchy.

```
SQL> SELECT Ename "Employee", CONNECT_BY_ISLEAF "IsLeaf",
          SYS_CONNECT_BY_PATH(Ename, '/') Path FROM Emp START WITH Empno = 7839
                                  CONNECT BY PRIOR Empno = MGR;
```

Working With SET Operators

SET Operators

- These Operators Are Used to Combine Information of Similar DATA Type From One or More Than One Table.
- DATA Type of The Corresponding Columns in All The SELECT Statements Should Be Same.
- The Different Types of SET Operators Are
  - UNION Operator.
  - UNION ALL Operator.
  - INTERSECT Operator.
  - MINUS Operator.
- SET Operators Can Combine Two or More Queries Into One Result.
- The Result of Each SELECT Statement Can Be Treated As a Set, and SQL Set Operations Can Be Applied on Those Sets To Arrive At a Final Result.
- SQL Statements Containing SET Operators Are Referred To As Compound Queries, And Each SELECT Statement in a Compound Query is Referred To As a Component Query.
- Set Operations Are Often Called Vertical Joins, As The Result Combines Data From Two or More SELECTS Based on Columns Instead of Rows.

The Generic Syntax

```
<component query>
{UNION | UNION ALL | MINUS |  INTERSECT}
<component query>;
```

UNION

- Combines The Results of Two SELECT Statements Into One Result Set, And Then Eliminates Any Duplicate Rows From That Result Set.

UNION ALL

- Combines The Results of Two SELECT Statements Into One Result Set Including The Duplicates.

INTERSECT

- Returns Only Those Rows That Are Returned By Each of Two SELECT Statements.

MINUS

- Takes The Result Set of One SELECT Statement, And Removes Those Rows That Are Also Returned By a Second SELECT Statement.

Point of Concentration

- The Queries Are All Executed Independently But Their Output is Merged.
- Only Final Query Ends With a Semicolon.

Rules and Restrictions

- The Result Sets of Both The Queries Must Have The Same Number of Columns.
- The Data Type of Each Column in The Second Result Set Must Match The Data Type of Its Corresponding Column in The First Result Set.
- The Two SELECT Statements May Not Contain An ORDER BY Clause, The Final Result of The Entire SET Operation Can Be Ordered.
- The Columns Used For Ordering Must Be Defined Through The Column Number.

Illustrations

```
SQL> SELECT Empno, Ename  FROM Emp WHERE Deptno = 10
                          UNION
     SELECT Empno, Ename  FROM Emp WHERE Deptno = 30 ORDER BY 1;

SQL> SELECT Empno, Ename, Job  FROM Emp WHERE Deptno = (SELECT Deptno
                                FROM Dept  WHERE Dname = 'SALES')
                          UNION
     SELECT Empno, Ename, Job  FROM Emp WHERE Deptno = (SELECT Deptno
                       FROM Dept  WHERE Dname = 'ACCOUNTING') ORDER BY 1;
SQL> SELECT Empno, Ename  FROM Emp  WHERE Deptno = 10
                     UNION ALL
     SELECT Empno, Ename  FROM Emp  WHERE Deptno = 30 ORDER BY 1;
```

```
SQL> SELECT Empno, Ename FROM Emp WHERE Deptno = 10
                        INTERSECT
     SELECT Empno, Ename FROM Emp WHERE Deptno = 30 ORDER BY 1;

SQL> SELECT Empno, Ename FROM Emp WHERE Deptno = 10
                        MINUS
     SELECT Empno, Ename FROM Emp WHERE Deptno = 30 ORDER BY 1;

SQL> SELECT Job FROM Emp WHERE Deptno = 20
                        UNION
     SELECT Job FROM EmpWHERE Deptno = 30;

SQL> SELECT Job FROM Emp WHERE Deptno = 20
                        UNION ALL
     SELECT Job FROM EmpWHERE Deptno = 30;

SQL> SELECT Job FROM Emp WHERE Deptno = 20
                        INTERSECT
     SELECT Job FROM EmpWHERE Deptno = 30;

SQL> SELECT Job FROM Emp WHERE Deptno = 20
                        MINUS
     SELECT Job FROM Emp WHERE Deptno = 10;

SQL> SELECT ROWNUM, Ename  FROM Emp  WHERE ROWNUM < 7
                        MINUS
     SELECT ROWNUM, Ename FROM Emp  WHERE ROWNUM < 6;
```

Working With Views in Oracle

View

- It is A Logical Table Based on One OR More Tables OR Views.
- A View in Practicality Contains No Data By Itself.
- The Tables Upon Which A View is Based Are Called As BASE TABLES.
- Views Can Be Created As Object Views OR Relational Views.
- The Object Views Support
  - LOB's
  - Object Types
  - Ref's
  - Nested Tables
  - Varrays
- Object View is A View of A User Defined Type, Where Each Row Contains Objects, And Each Object With An Object Identifier.

Prerequisites

- Should Have CREATE VIEW OR CREATE ANY VIEW System Privilege.
- To Create A Subview, We Need Under Any View System Privilege OR Under Object Privilege on The Superview.
- The Owner of The Schema Should Have SELECT, INSERT, UPDATE Or DELETE Rows From All Tables OR Views on Which The View is Based.
- The Above Privileges Must Be Granted Through Privileges Directly, Rather Than A Role.

Syntax

```
SQL> CREATE [OR REPLACE] [{FORCE/NOFORCE}] VIEW        ViewName
        [(AliasName[,AliasName...])] AS SubQuery [WITH{CHECK OPTION/READ ONLY}]
        [CONSTRAINT ConstraintName];
```

OR REPLACE

- Specifies The View Has To Be Replaced If Already Existing.

FORCE

- Specifies The View Has To Be Created Even If The Base Tables Dows Not Exist.

NOFORCE

- Specifies The View Should Not Be Created If The Base Table Does Not Exist, Which is Default.

ALIAS_NAME

- Specifies The Name of An Alias For An Expression in The Subquery.

WITH CHECK OPTION

- Specifies That Only Rows That Would Be Retrieved By The Subquery Can Only Be INSERTED, UPDATED OR DELETED.

CONSTRAINT

- Constraint_Name Specifies The Name Of The Constraint As WITH CHECK OPTION Or READ ONLY Constraint.

WITH READ ONLY

- Specifies That Rows Must Only Be Read From The Base Tables.

Restrictions

- If A View Has INSERTED Of Trigger, Then All Sub Views Created on It Must Have INSTEAD OF Triggers, Even If The Views Are Inherently Updatable.
- An Alias Cannot Be Specified When Creating An Object View.

Types of Views

- Simple Views
- Complex Views

Simple Views:

- Which Contains A Subquery That Retrieves Data From Only One Base Table.

Complex Views

- Contains A Subquery That Can Perform Any Of These Actions.
  - Retrieving From Multiple Base Tables.
  - Groups Rows Using A Group By or Distinct Clause.

- Contains A Function Call.

Simple Views

```
SQL> CREATE VIEW Employees AS SELECT  Empno "ID Number",  Ename Name,
        Sal "Basic Salary",  Job Designation FROM Emp;
```

Selecting Data From A View

```
SQL> SELECT Name, Designation  FROM Employees;
SQL> SELECT "ID Number", Name, "Basic Salary" * 12 FROM Employees;
SQL> SELECT "ID Number",  Name,  TO_CHAR("Basic Salary", '99,99,999.99') Monthly,
        "Basic Salary" * 12 Annual FROM Employees WHERE "Basic Salary" > 2500;
```

Complex Views

```
SQL> CREATE VIEW EmpInfo AS SELECT  E.Empno  EmployeeID, E.Ename  Name,
        D.Deptno  DepartmentID, D.Dname    DepartmentName  FROM Emp E, Dept D
                                    WHERE D.Deptno = E.Deptno ORDER BY D.Deptno;
SQL> CREATE VIEW EmpGrades  AS SELECT  E.Ename Name, E.Sal Basic, S.Grade Grade
              FROM Emp E , Salgrade S WHERE E.Sal BETWEEN S.LoSal AND S.HiSal
                                                    ORDER BY S.Grade;
SQL> CREATE OR REPLACE VIEW EmpManagers  AS SELECT RowNum SerialNo,
        INITCAP(E.Ename)||' Works Under '|| M.Ename  "Employee And Managers"
              FROM Emp E, Emp M  WHERE E.Mgr = M.Empno;

SQL> CREATE OR REPLACE VIEW EmpAccounts  AS SELECT  Ename,  Deptno,  Sal Monthly,
        Sal * 12 Annual  FROM  Emp  WHERE Deptno = (SELECT Deptno FROM Dept
              WHERE Dname = 'ACCOUNTING')  ORDER BY Annual;

SQL> CREATE OR REPLACE VIEW CumSum  AS SELECT B.Sal, SUM(A.Sal) Cum_Sal
        FROM Emp A, Emp B WHERE A.RowID < = B.RowID GROUP BY B.RowID, B.Sal;

SQL> CREATE OR REPLACE VIEW OrgDesignations  AS SELECT Job  FROM Emp
        WHERE Deptno = 10 UNION  SELECT Job  FROM Emp   WHERE Deptno IN(20, 30);
```

Data Access Using Views

- The Steps OR Operations Performed By The Oracle Server, When Data Is Accessed Using A VIEW Are
  - Retrieves The VIEW Definition From The Data Dictionary Table USER_VIEWS.
  - Checks The Access Privileges For The Views Base Table.
  - Converts The View Query into An Equivalent Operation on The Underlying Base Table OR Tables.

VIEWS in Data Dictionary

- Once The View Has Been Created, We Can Query Upon The DATA DICTIONARY Table Called USER_VIEWS To See The Name And Definition of The View.
- The Text of The SELECT Statement That Constitutes The VIEW is Stored in A Long Column.

```
SQL> SELECT VIEW_NAME, TEXT
        FROM USER_VIEWS;
```

Modifying A View

- OR REPLACE Option is Used To Modify An Existing VIEW With A New Definition.
- A View Can Be Altered Without Dropping, Recreating, And Regranting Object Privileges.
- The Assigned Column Aliases in The CREATE VIEW Clause, Are Listed in The Same Order As The Columns in The Subquery.

Creating Views With Columns Declarations

- When A VIEW is Being Created, We Can Specify The Names of The Columns, That It Can Project, Along With The View's Definition.
- The View in This Case Totally Hides The Original Names From The Base Table.

```
SQL> CREATE VIEW  DeptSalSummary( DepartmentName,  MinimumSalary,  MaxSalary,
              AverageSalary,  SalarySum )  AS  SELECT D.Dname,  MIN(E.Sal), MAX(E.Sal),
                    AVG(E.Sal),  SUM(E.Sal)  FROM Emp E, Dept D WHERE E.Deptno =
```

D.Deptno GROUP BY D.Dname;

Using Views To Create On The Fly Tables

SQL> CREATE VIEW InsertDept10 AS SELECT * FROM Emp WHERE Deptno = 10;

CREATING a TABLE With Data Using a VIEW

SQL> CREATE TABLE Dept10 AS SELECT * FROM InsertDept10;

SQL> CREATE VIEW EmpGradeIns AS SELECT Ename, Job, Sal, Grade FROM Emp E, SalGrade S WHERE E.Sal BETWEEN S.LoSal AND S.HiSal;

SQL> Create Table EmpGrades (Employee, Designation, BasicSalary, Grade) AS SELECT * FROM EmpGradeIns;

Dropping A View

- The DROP VIEW Statement is Used To Remove A View Permanently.
- Dropping A View Has No Affect on The Tables Upon Which The View is Created.
- Views OR Applications Based on Deleted Views Become Invalid.
- We Need DROP ANY VIEW Privilege To Remove The Views From Data Dictionary.

Syntax

SQL> DROP VIEW ViewName;

Example

SQL> DROP VIEW InsertDept;

Inline Views

- An INLINE VIEW is A Subquery With An ALIAS (Called As CORRELATION NAME), That Can Be Used Within A SQL Statement.
- An Inline View is Similar To Using A Named Subquery in The FROM Clause Of The Main Query.
- An Inline View is Not A Schema Object.
- An Inline View in The FROM Clause of A SELECT Statement Defines A Data Source For The Select Statement.

SQL> SELECT E1.Ename, E1.Sal, E1.Deptno, E2.Maxsal FROM Emp E1, (SELECT Deptno, MAX(Sal) Maxsal FROM Emp GROUP BY Deptno) E2 WHERE E1.Deptno = E2.Deptno AND E1.Sal < E2.Maxsal;

Performing DML Operations On A View:

- DML Operatons Can Be Perfomed Upon A Table Through VIEW.

Rules To Follow

- A Row Can Be Removed From A View Unless It Contains.
  - Group Function.
  - A GROUP BY Clause.
  - The ROWNUM PESUDO COLUMN.
  - The DISTINCT Key Word.
  - The Columns Defined By Expressions.
- Data Can Be Added Through A View, Unless It Contains Any of The Above Rules And There Does Not Exist Not Null Columns, And Without Default Value.

SQL> CREATE VIEW InsertDept(DeptID, DeptName , Place) AS SELECT Deptno, Dname, Loc FROM Dept;

Inserting Data Into Dept Table Using The InsertDept View

SQL> INSERT INTO InsertDept(DeptID, DeptName, Place) VALUES(50, 'ADMINISTRATION', 'DELHI');

Updating Data in Dept Table Using The InsertDept View

SQL> UPDATE InsertDept SET PLACE = 'MUMBAI' WHERE DeptID = 50;

Deleting Data From Dept Table Using The InsertDept View

SQL> DELETE FROM InsertDept WHERE DeptID = 50;

- Many Times These Transactions Look Consistent But It is Not TRUE, Hence Care Should Be Taken When Implementing Them.

Using with CHECK Option Clause

- To Ensure That DML on The View Stays Within The Domain of The VIEW We Use The WITH CHECK Option Clause.
- Views Make It Possible To Perform Referential Integrity Checks.
- Using Views We Can Enforce Constraints At Database Level.
- Using Views We Can Protect The Data Integrity, But The Use is Very Limited.
- The WITH CHECK OPTION Clause Specifies That INSERTS, & UPDATES Performed Through The VIEW Are Not Allowed To CREATE Rows, That The VIEW Cannot SELECT.
- Views Allow Integrity Constraints And Data Validation Checks To Be Enforced On Data Being INSERTED OR UPDATED.

```
SQL> CREATE OR REPLACE VIEW EDept30  AS SELECT *  FROM Emp  WHERE Deptno = 30
                                    WITH CHECK OPTION CONSTRAINT EDept30ChkView ;

SQL> CREATE OR REPLACE VIEW Emanage AS SELECT *  FROM Emp  WHERE
                Job = 'MANAGER' WITH CHECK OPTION CONSTRAINT EmanagerView;
```

Applying With READ ONLY OPTION

- By Adding The WITH READ ONLY Option We Can Ensure That No DML Operations Are Executed Through VIEW.
- An Attempt To Perform A DML Operation Results in Oracle Server Error.

```
SQL> CREATE OR REPLACE VIEW  EDptRead( EmpID,  Name,  Designation )
        AS SELECT Empno, Ename, Job  FROM Emp WHERE Deptno = 20 WITH READ ONLY;
```

Understanding Constraint State

- As Part of Constraint Definition, We Can Specify How And When Oracle Should Enforce The Constraint.

CONSTRAINT_STATE

- We Can Use The CONSTRAINT_STATE With Both INLINE And OUT-OF-LINE Specification.
- We Can Specify The Clauses of CONSTRAINT_STATE in Any Order, But We Can Specify Each Clause Only Once.

Constraint Clauses

Deferrable And Not Deferrable Clauses

- The Deferrable And Not Deferrable Parameters Indicate Whether OR Not, in Subsequent Transactions, Constraint Checking Can Be Deferred Until The End of The Transaction.
- The Constraint State Can Be Changed Using The SET CONSTRAINT(S) Statement.
- If We Omit This Clause While Creating The Table, Then The Default is NOT DEFERRABLE.
- We Specify NOT DEFERRABLE To Indicate That In Subsequent Transactions We Cannot Use The SET CONSTRAINT[S] Clause To Defer Checking of The Constraint Until The Transaction is Committed.
- We Specify DEFERRABLE To Indicate That in Subsequent Transactions We Can Use The SET CONSTRAINT[S] Clause To Defer Checking of The Constraint Until After The Transaction is Committed.
- The DEFERRABLE Setting in Effect Lets Us Disable The Constraint Temporarily While Making Changes To The Database That Might Violate The Constraint Until All The Changes Are Complete.

Restriction on [NOT] DEFERRABLE

- We Cannot Specify Either of The Above Clauses For A View Constraint.

INITIALLY Clause

- The INITIALLY Clause Establishes The Default Checking Behavior For Constraints That Are DEFERRABLE.
- The INITIALLY Setting Can Be Overridden By A SET CONSTRAINT(S) Statement in A Subsequent Transaction.
- Specify Initially Immediate To Indicate That Oracle Should Check The Constraint At The End of Each Subsequent SQL Statement.

- If We Do Not Specify INITIALLY At All, Then The Default is INITIALLY IMMEDIATE.
- Specify INITIALLY DEFERRED To Indicate That Oracle Should Check This Constraint At The End of Subsequent Transactions.

VALIDATE OR NOVALIDATE

- The Behavior of VALIDATE And NOVALIDATE Always Depends on Whether The Constraint is Enabled OR Disabled.

ENABLE Clause

- Specify ENABLE if We Want The Constraint To Be Applied To The Data in The Table.
- ENABLE VALIDATE Specifies That All Old And New Data Also Complies With The Constraint.
- ENABLE NOVALIDATE Ensures That All New DML Operations on The Constrained Data Comply With The Constraint.

Restriction On The Enable Clause

- We Cannot ENABLE A FOREIGN KEY That References A DISABLED UNIQUE OR PRIMARY KEY.

DISABLE Clause

- Specify DISABLE To DISABLE The INTEGRITY Constraint.
- Disabled Integrity Constraints Appear in The Data Dictionary Along With Enabled Constraints.
- If We Do Not Specify This Clause When Creating A Constraint, Oracle Automatically Enables The Constraint.
- DISABLE VALIDATE Disables The Constraint And Drops The Index on The Constraint, But Keeps The Constraint Valid.
- This Feature is Most Useful in Data Warehousing Situations, Because It Lets Us Load Large Amounts of Data While Also Saving Space By Not Having An Index.
- DISABLE NOVALIDATE Signifies That Oracle Makes No Effort To Maintain The Constraint.
- If We Specify Neither VALIDATE Nor NOVALIDATE, Then The Default is NOVALIDATE.

RELY Clause

- RELY And NORELY Are Valid Only When We Are Modifying An Existing Constraint.
- These Parameters Specify Whether A Constraint in NOVALIDATE Mode is To Be Taken Into Account For Query Rewrite.
- Specify RELY To Activate An Existing Constraint in NOVALIDATE Mode For Query Rewrite in An Unenforced Query Rewrite Integrity Mode.
- The Constraint is in NOVALIDATE Mode, So Oracle Does Not Enforce It.
- The Default is NORELY.

Restriction on The RELY Clause

- We Cannot Set A NONDEFERRABLE NOT NULL Constraint To RELY.

DEFERRABLE Constraints Illustration

Table With Default ENABLE State

```
SQL> CREATE TABLE MyMaster
        ( MastID      NUMBER(2) CONSTRAINT MastIDPK PRIMARY KEY
          MastName   VARCHAR2(10) CONSTRAINT MastNameCHK
                              CHECK(MastName = UPPER(MastName), MastDate    DATE
          CONSTRAINT MastDateNN NOT NULL  );

SQL> CREATE TABLE MyMasterDetail
        ( DetailID NUMBER(2), MastID NUMBER(2), DetailName VARCHAR2(10)
              CONSTRAINT DetailNameCHK  CHECK(DetailName = UPPER(DetailName)),
          DetailDate  DATE CONSTRAINT DetailDateNN NOT NULL,
                    CONSTRAINT DetailIDPK PRIMARY KEY(DetailID),
                        CONSTRAINT MastIDFK FOREIGN KEY(MastID)
                                        REFERENCES MyMaster(MastID) );
```

View Constraint

- In Practicality Oracle Does Not Enforce View Constraints, But Operations on VIEWS Are Subject To The INTEGRITY CONSTRAINTS Defined on The Underlying Base Tables.

- CONSTRAINTS on VIEWS Can Be Enforced Through CONSTRAINTS on Base Tables.

Restrictions on VIEW CONSTRAINTS

- The VIEW CONSTRAINTS Are Subset Of TABLE CONSTRAINTS.
- Only UNIQUE, PRIMARY KEY And FOREIGN KEY Constraints Can Be Specified on Views.
- The Check Constraint is Imposed Using WITH CHECK OPTION.
- As View Constraints Are Not Enforced Directly, We Cannot Specify INTIALLY DEFERRED OR DEFERRABLE Clauses.
- View Constraints Are Supported Only in DISABLE NOVALIDATE Mode.
- The RELY And NO RELY Key Words Will Instruct The Oracle Server When The View Constraint Should Be Enforced.
- As View is A Logical Table The Constraints on View Are Not Highly Considered in Practice.
- The View Constraints Are Adopted For Maintenance Easiness.

```
SQL> CREATE VIEW EmpSalary ( EmpID, Ename, Email UNIQUE RELY DISABLE
        NOVALIDATE, CONSTRAINT ID_PK PRIMARY KEY(EmpID) RELY DISABLE
                NOVALIDATE ) AS SELECT Empno, Ename, Email FROM Emp;
```

Let Us Increase The Speed and Performance of the Database Using Indexes

Working With Indexes

Index

- It Is A Schema Object Which Contains An Entry For Each Value That Appears In The Indexed Column(s) Of The Table Or Cluster.
- Index Provides Direct, Fast Access To Rows.

Types of Indexes

Normal Indexes

- They Are Default Indexes.
- They Are Created With B-tree Principle.

Bitmap Indexes

- They Store ROWID'S Associated With A Key Value As A Bitmap.

Partitioned Indexes:

- They Contain Partitions Containing An Entry For Each Value That Appears In The Indexed Columns of The Table.

Function Based Indexes

- They Are Based on Expressions.
- Enable Query To Evaluate Value Retuned By An Expression.

Domain Indexes

- They Are Indexes Which Are Instances of An Application Specific Index of Type Indextype.

Pre Requisites

- The TABLE OR CLUSTER To Be Indexed Must Be in The Own Schema.
- INDEX OBJECT Privilege Should Be Available on The Table To Be Indexed.
- CREATE ANY INDEX System Privilege Must Be Available.
- Unlimited Tablespace System Privilege OR Space Quota On Table Spaces Must Be Available.
- For DOMAIN Indexes, EXECUTE OBJECT Privilege on The Indextype Should Be Available.
- For FUCTION Based Indexes, The Function Used For Indexing Must Be Marked As Deterministic.

Restrictions

- If INDEX is Locally Partitioned Then The TABLE Must Be Partitioned.
- If The TABLE is INDEX ORAGANIZED, Then A Secondary INDEX is Created.
- If The TABLE is TEMPORARY TABLE, Then INDEX is Also Temporary With The Same Scope, As That of The Table.

Syntax

```
SQL> CREATE {[UNIQUE]/[BITMAP]} INDEX IndexName ON TableName(ColumnName
                [,ColumnName..]) TABLESPACE TableSpaceName;
```

Simple Index OR Normal Index

- A Simple Index OR Normal Index is Created Upon A Table By Considering Only One Column.
- These Indexes Are Generally Created Using The Algorithm of B-Tree Index.
- We Can Create Index On Columns Containing NULL OR Repeated Data if UNIQUE Key Word is Not Used.
- These Are Default Indexes in Oracle.

```
SQL> CREATE INDEX EmpEmpnoIDX ON Emp(Empno);
SQL> CREATE INDEX DeptDeptnoIDX ON Dept(Deptno);
```

Creating Composite Indexes

- COMPOSITE Index is An INDEX on Multiple Columns of A Table.
- COMPOSITE Index Can Be Created Upon A Table To The Maximum Collection of 32 Columns.

```
SQL> CREATE INDEX Stud_Fname_Lname_Idx ON Student(Fname, Lname);
SQL> CREATE INDEX Emp_Name_Place_IDX ON Emp(Ename , StreetName);
```

Creating Unique Indexes

- Specify UNIQUE To Indicate That The Value of The Column OR Columns Upon Which The INDEX is Based Must Be UNIQUE.

Restrictions

- We Cannot Specify Both UNIQUE and BITMAP Indexes at a Time.
- UNIQUE Key Word Cannot Be Specified For A DOMAIN INDEX.

SQL> CREATE UNIQUE INDEX EmpEmailIDxUNQ ON Emp(Email);
SQL> CREATE UNIQUE INDEXStuPhnoIDxUNQ ON Student(PhoneNo);

BITMAP Indexes

- Specify BITMAP To Indicate That INDEX Has To Be Created With A BITMAP For Each DISTINCT KEY in The Table.
- BITMAP Indexes Store The ROWID's Associated With A KEY Value As A BITMAP.
- Each BIT in The BITMAP Corresponds To A Possible ROWID.
- These Indexes Are Used To Tune Queries That Use Non Selective Columns in Their Limiting Conditions
- BITMAP INDEXES Should Be Used Only When The Data is Infrequently Updated.
- BITMAP INDEXES Add To The Cost of All Data Manipulation Transactions Against The Tables They INDEX.
- The Oracle Optimizer Can Dynamically Convert BITMAP INDEX Entries To ROWID's During The Query Processing.

Restrictions

- BITMAP Cannot Be Specified When Creating A Global Partitioned Index.
- Bitmap Secondary Index Cannot Be Created on An INDEX ORGANIZED TABLE Unless The Index Organized Table Has A Mapping Table Associated With It.
- We Cannot Specify Both UNIQUE And BITMAP Indexes At A Time.
- BITMAP Cannot Be Specified For A DOMAIN INDEX.
- BITMAP INDEXES Should Not Be Used For Tables Involved in OLTP.
- BITMAP Indexes Increase The Load Factor on The INTERNAL Mechanisms of Oracle To Maintain Them.
- Restricted With Usage To Tables Involved in BATCH Transactions.

SQL> CREATE BITMAP INDEX EmpBitMapDeptno ON Emp(Deptno);

Bitmap Index Structure

Dept no	Start ROWID	End ROWID	Bit Pattern
10	AAANsYAAEAAABSGAAA	AAANsYAAEAAABSGAAN	10100000000001
20	AAANsYAAEAAABSGAAA	AAANsYAAEAAABSGAAN	00010000011110
30	AAANsYAAEAAABSGAAA	AAANsYAAEAAABSGAAN	01001111100000

- Each Row in The Table Being Indexed Adds Only One Bit to The Size of The Bitmap Pattern Column For The Bitmap Index.
- Each Distinct Value Adds Another Row to The Bitmap Index.

Creating Function Based Indexes

- These Are Indexes Based on Expressions in SELECT Statements.
- The INDEX Expressions Are Built From Table Columns, Containing SQL Functions OR User Defined Functions.
- FUNCTION BASED INDEXES Defined With The UPPER(ColumnName) OR LOWER(ColumnName) Allow Case Insensitive Searches.
- We Should Have CREATE INDEX And QUERY REWRITE Privileges.
- We Should Have EXECUTE Object Privilege on The Functions Used in The Function Based Indexes.

- Function Based Indexes Are Designed To Improve Query Performance When The Function is Used in WHERE Clause.
- To Ensure That Oracle Uses The INDEX Rather Than Performing A Full Table Scan, We Should Be Sure That The Value of The Function is NOT NULL in The Subsequent Queries.
- Oracle Treats INDEXES With Columns Marked DESC As Function Based Indexes.
- The Function Based Indexes Are Used Only When The Query Statement is Executed Through The Specified Function.

```
SQL> CREATE INDEX EmpFuncAnnSalIDX  ON Emp(Sal * 12);
```

INDEX Creation is of Two Types

- Automatic.
- Manual.

Specifications of An INDEX

- INDEX is A Schema Object.
- Index is Used By The Oracle Server To Speed Up Retrieval of Rows By Using A Pointer.
- INDEX Reduces The Disk I/O By Using Rapid Path Access Method To Locate The Data Quickly.
- Index's Are Independent of The Table it Indexes, Both Logically And Physically.
- Index is Used And Maintained Automatically By The Oracle Server.
- Index's Can Be Created OR Dropped At Any Time And Have No Effect on The Base Tables OR Other Indexes.
- When A Table is Dropped, The Corresponding Indexes Are Also Dropped Automatically.
- On One Table More Than One Index Can Be Created, But This Does Not Mean That, More The Indexes, Lead To More Faster Performance.
- Each DML Operation That is Committed on A Table With Index, Means That The Index Must Be Updated.

Dropping An Index

Syntax

```
SQL> DROP INDEX INDEX_NAME;
```

When To Create An Index

- The Column is Used Frequently in The WHERE Clause OR in A Join Condition.
- The Column Contains A Wide Range Of Values.
- The Column Contains A Large Number Of NULL Values.
- Two OR More Columns Are Frequently Used Together in A Where Clause OR Join Condition.
- The Table is Large And Most Queries Are Expected To Retrieve Less Than 2 To 4 % of The Rows.

When Not To Create An Index

- The Table is Too Small.
- The Columns Are Not Often Used As Condition in The Query.
- Most Queries Are Expected To Retrieve More than 2 To 4 % of The Rows.
- The Table is Updated Frequently.

Querying For Indexes

- The Indexes Can Be Confirmed From The USER_INDEXES Data Dictionary.
- The Column That Are Involved in An Index By Querying USER_IND_COLUMNS.

```
SQL> SELECT UC.TABLE_NAME TabName,  UC.COLUMN_NAME ColName,
                I.INDEX_NAME IDXName  FROM USER_IND_COLUMNS UC,
                        USER_INDEXES UI WHERE UC.INDEX_NAME = UI.INDEX_NAME;
```

Fundamentals of Data Base Security

Database Security

- The SECURITY Upon Databases is Applied By Defining And Describing Separate SCHEMA OBJECTS And GRANTING Required PRIVILEGES Upon Them.
- In Oracle The Privileges Can Be Granted As Well As Revoked.

GRANT Command

Syntax

SQL> GRANT <PrivilageName1>, <PrivilageName2>, ON <ObjectName> TO <UserName>;

- GRANT Command is Used When We Want The Database To Be Shared With Other Users.
- The Other Users Are GRANTED With Certain Type of RIGHTS.
- GRANT Command Can Be issued Not Only on TABLE OBJECT, But Also on VIEWS, SYNONYMS, INDEXES, SEQUENCES Etc.

SQL> GRANT SELECT ON EMP TO ENDUSERS;

SQL> GRANT INSERT, SELECT, DELETE ON EMP TO OPERATORS ;

SQL> GRANT INSERT (Empno, Ename, Job) ON Emp To EndUsers ;

REVOKE Command

Syntax

SQL> REVOKE <PrivilegeName1>, <PrivilegeName2>, ON <ObjectName> FROM <UserName>;

- REVOKE is Used When We Want One Database To Stop Sharing The Information With Other Users.
- Revoke Privileges is Assigned Not Only On TABLE Object, But Also on VIEWS, SYNONYMS, INDEXES Etc.

SQL> REVOKE INSERT, DELETE ON EMP FROM Operators;

Types of Privileges

System Privileges

- They Allow A User To Perform Certain Actions Within The Database.

Object Privileges

- An Object Privilege Allows A User To Perform Certain Actions on Database Objects.

Checking The Object Privileges Granted

- The Schema Object That Stores The Information About The Privileges Granted is USER_TAB_PRIVS_MADE
- The Columns of USER_TAB_PRIVS_MADE
  - GRANTEE
  - TABLE_NAME
  - GRANTOR
  - PRIVILEGE
  - GRANTABLE
  - HIERARCHY

SQL> SELECT GRANTEE, TABLE_NAME "Table", GRANTOR, PRIVILEGE FROM USER_TAB_PRIVS_MADE;

Checking Object Privileges Received

- The Schema Object That Stores The Information About The PRIVILEGES That Are Received is USER_TAB_PRIVS_RECD.
- The Columns of USER_TAB_PRIVS_RECD
  - OWNER
  - TABLE_NAME
  - GRANTOR
  - PRIVILEGE
  - GRANTABLE
  - HIERARCHY

SQL> SELECT OWNER, TABLE_NAME "Table", GRANTOR, PRIVILEGE FROM USER_TAB_PRIVS_RECD;

Making Use Of Object Privileges

- Once A Particular USER Has Been Granted An Object Privilege, The Specific USER Can Perform The Tasks As Granted By The Privilege.

Steps To Be Performed

- Connect To The Required User Using The USER Name and PASSWORD.
- Execute The Required SQL Statement Using The Object Hierarchy.

SQL> SELECT * FROM SCOTT.EMP;

Working With Roles

- A ROLE is A Group of PRIVILEGES That Can Be Assigned To A USER OR Another ROLE.

Advantages

- Rather Than Assigning Privileges One At A Time Directly To A USER, We Can CREATE A ROLE, Assign PRIVILEGES To That ROLE, And Then GRANT That ROLE To Multiple USERS And ROLES.
- When You Add OR Delete A Privilege From A Role, All Users And Roles Assigned That ROLE Automatically Receive OR Lose Those Privileges.
- We Can Assign Multiple Roles To A Single USER OR One ROLE To Another ROLE.
- A Role Can Be Assigned With A Password.

ROLE's Creation

- To Create A Role We Should Have The CREATE ROLE SYSTEM Privilege.
- The Steps in Implementing The Roles
  - Role Creation
  - Granting Privileges To Roles
  - Granting ROLES TO USERS OR OBJECTS

Syntax

SQL> CREATE ROLE <RoleName> [IDENTIFIED BY <Password>];

SQL> CREATE ROLE Sales_Manger IDENTIFIED BY SalesAudit;

Granting Privileges to ROLE

- The ROLES Are Granted Privileges Using The GRANT Statement.
- A Role Can Be Granted Both System As Well As Object Privileges At a Time.

SQL> GRANT SELECT, INSERT, UPDATE, DELETE ON EMP TO Sales_Manager;

Granting A ROLE To a USER

SQL> GRANT Sales_Manager TO SCOTT;

Granting Multiple ROLES to Another ROLE

SQL> GRANT ROLE1, ROLE2, ... TO <TARGET_ROLE_NAME>;

Checking ROLES Granted To A USER

- The Schema Object USER_ROLE_PRIVS Specifies The ROLES Granted To A USER.
- The Columns of USER_ROLE_PRIVS
  - USERNAME
  - GRANTED_ROLE
  - ADMIN_OPTION
  - DEFAULT_ROLE
  - OS_GRANTED

SQL> SELECT USERNAME, GRANTED_ROLE FROM USER_ROLE_PRIVS;

SYSTEM Privileges Granted to A ROLE

- The Schema Object ROLE_SYS_PRIVS Specifies The SYSTEM PRIVILEGES Granted To A ROLE.
- The Columns of ROLE_SYS_PRIVS
  - ROLE
  - PRIVILEGE
  - ADMIN_OPTION

SQL> SELECT ROLE, PRIVILEGE FROM ROLE_SYS_PRIVS;

OBJECT Privileges Granted to A ROLE

- The Schema Object ROLE_TAB_PRIVS Specifies The OBJECT PRIVILEGES Granted To A ROLE.
- The Columns of ROLE_TAB_PRIVS
  - ROLE
  - OWNER
  - TABLE_NAME
  - COLUMN_NAME
  - PRIVILEGE
  - GRANTABLE

SQL> SELECT ROLE, PRIVILEGE FROM ROLE_TAB_PRIVS;

Revoking a ROLE

SQL> REVOKE Sales_Manager FROM SCOTT;

Revoking All Privileges From A ROLE

SQL> REVOKE ALL ON Emp FROM Sales_Manager;

Dropping A ROLE

Syntax

DROP ROLE <RoleName>;

SQL> DROP ROLE Sales_Manager;

Working With SYNONYMS

- A Synonym is Schema Object, Which Acts As An Alternate Name For An Existing Schema Object.
- By Using A Synonym, We Can Avoid The Entry of The SCHEMA Name, When Referencing Upon OBJECTS That Belong To Other SCHEMA.
- The CREATE SYNONYM Privilege is Necessary To Execute The Creation of A SYNONYM.

Syntax

SQL> CREATE [PUBLIC] SYNONYM <SynonymName> FOR <SchemaName>. <ObjectName>;

Illustration

SQL> CREATE SYNONYM EmpInfo FOR SCOTT.Emp;

SYNONYM Types

- The Synonyms Are Practically of Two Types
  - PRIVATE SYNONYM
  - PUBLIC SYNONYM
- We Should Have CREATE PUBLIC SYNONYM Privilege, Which Can Be Accessed By All USERS in The Data Base.

SQL> CREATE PUBLIC SYNONYM EmpInfo FOR SCOTT.Emp;

Dropping A Synonym

Syntax

DROP [PUBLIC] SYNONYM <SynonymName>;

Illustration

SQL> DROP SYNONYM EmpInfo;

SQL> DROP PUBLIC SYNONYM EmpInfo;

OLAP Features in Oracle

Some Features For Query Processing in ORACLE Include The Use Of ONLINE ANALYTICAL PROCESSING(OLAP) Upon The Data Base.

- OLAP Features Are Useful For Data Warehousing And Data Mart Applications.
- The OLAP Operations Are Performance Enhancements.
  - TOP_N QUERIES.
  - GROUP BY.
  - CUBE.
  - ROLLUP.

ROLLUP Option

- It is A GROUP BY Operation And is Used To Produce Subtotals At Any Level Of The Aggregation.
- The Generated Sub Totals "Rolled Up" To Produce Grant Total.
- The Totaling is Based On A One Dimensional Data Hierarchy of Grouped Information.

Syntax

GROUP BY ROLLUP (Column1, Column2,..)

Illustrations

SQL> SELECT Deptno, SUM(SAL) FROM Emp GROUP BY ROLLUP(Deptno);

SQL> SELECT Job, SUM(Sal) FROM Emp GROUP BY ROLLUP(Job);

SQL> SELECT Job, AVG(Sal) FROM Emp GROUP BY ROLLUP(Job);

Passing Multiple Columns To ROLLUP

- When Multiple Columns Are Passed To ROLLUP, The ROLLUP, Groups The Rows Into Blocks With The Same Column Values.

SQL> SELECT Deptno, Job, SUM(Sal) Salary FROM Emp GROUP BY ROLLUP(Deptno, Job);

SQL> SELECT Job, Deptno, SUM(Sal) Salary FROM Emp GROUP BY ROLLUP(Job, Deptno);

SQL> SELECT Job, Deptno, AVG(Sal) Average FROM Emp GROUP BY ROLLUP(Job, Deptno);

- NULL Values in The Output Of ROLLUP Operations Typically Mean That The Row Contains Subtotal Or Grant Total Information.
- Use NVL() Function For Proper Meaning.

CUBE Option

- It is An Extension Similar To ROLLUP.
- Cube Allows To Take A Specified Set of Grouping Columns And Crate Sub Totals For All Possible Combinations of Them.
- The Result of Cube is A Summary That Shows Subtotals For Every Combination of Columns OR Expressions in The Group By Clause.
- The Implementation of Cube is Also Called As N-Dimensional Cross Tabulation.

SQL> SELECT Deptno, Job, SUM(Sal) Salary FROM Emp GROUP BY CUBE(Deptno, Job);

SQL> SELECT Job, Deptno , SUM(Sal) Salary FROM Emp GROUP BY CUBE(Job, Deptno);

Applying GROUPING() Function

- The GROUPING() Function Accepts A Column And Returns 0 OR 1.
- GROUPING() Function Returns 1 When The Column Value is NULL, And Returns 0 When The Column Value is NOT NULL.
- GROUPING() Function is Used Only Upon Queries That Use ROLLUP OR CUBE.
- GROUPING() Function is Useful When We Want To Display A Value When A NULL Would Otherwise Be Returned in OLAP Queries.

SQL> SELECT GROUPING(Deptno), Deptno, SUM(Sal) FROM Emp GROUP BY ROLLUP(Deptno);

SQL> SELECT GROUPING(Job), Job, SUM(Sal) FROM Emp GROUP BY ROLLUP(Job);

DECODE Function

- It Is A Single Row Function.
- The Function Works On The Same Principle As The If – Then – Else.
- We Can Pass A Variable Number Of Values Into The Call Of The DECODE() Function.
- The First Item is Always The Name Of The Column That Need To Be Decoded.
- Once All Value-Substitute Pairs Have Been Defined, We Can Optionally Specify A Default Value.

Syntax

SQL> SELECT DECODE(ColumnName, Value 1, Substitute1, Value 2, Substitute2, ...
ReturnDefault)
FROM TableName;

- The Function Has No Restriction on The INPUT And OUTPUT Data Type.
- It is The Most Power full Function in Oracle.
- The Function Can Work For only an Analysis That Considers an Equality Operator in The Logical Comparision.

SQL> SELECT Ename, Job, Sal, DECODE(Deptno, 10, 'ACCOUNTING', 20, 'RESEARCH',
30 , 'SALES', 40 , 'OPERATIONS', 'OTHER') Departments
FROM Emp ORDER BY Departments;

GROUPING() With DECODE()

- The DECODE() Function Can Be Used To Convert 1 And 0 Returned Through GROUPING() into A Meaningful Output.

SQL> SELECT
DECODE(GROUPING(Deptno), 1, 'All Departments', Deptno) Departments, SUM(Sal)
FROM Emp GROUP BY ROLLUP(Deptno);

SQL> SELECT DECODE(GROUPING(Job), 1, 'All Designations', Job) Designations , SUM(Sal)
FROM Emp GROUP BY ROLLUP(Job);

DECODE() and GROUPING() To Converting Multiple Column Values

SQL> SELECT DECODE(GROUPING(Deptno), 1, 'All Departments', Deptno) Departments,
DECODE(GROUPING(Job), 1, 'All Designations', Job) Designations, SUM(Sal)
FROM Emp GROUP BY ROLLUP(Deptno, Job);

GROUPING() With DECODE() and CUBE

SQL> SELECT DECODE(GROUPING(Deptno), 1 , 'All Departments', Deptno) Departments,
DECODE(GROUPING(Job), 1, 'All Designations', Job) Designations, SUM(Sal)
FROM Emp GROUP BY CUBE(Deptno, Job);

Applying GROUPING SETS Clause

- The GROUPING SETS Clause is Used To Get The SUBTOTAL Rows.

SQL> SELECT Deptno, Job, SUM(Sal) FROM Emp GROUP BY GROUPING SETS(Deptno, Job);

Working with CASE Expressions

- The CASE Expression Can Be Used To Perform If–Then–Else Logic in SQL.
- CASE is Similar To DECODE But It is ANSI – Compliant.
- It Can be Used Even For Executing Conditions on range Based Comparison.
- Case Expressions Are of Two Types
  - SIMPLE CASE Expressions
  - SEARCHED CASE Expressions.

Simple CASE Expressions

- These Expressions Are Used To Determine The Returned Value.
- They Work With Equality Comparison Only, Almost All Similar To DECODE.
- It Has A Selector Which Associates To The Compared Value Either From The Column or Constant.
- The Value in The Selector is Used For Comparison With The Expressions Used in The WHEN Clause.

Syntax

```
SQL> CASE Search_Expr  WHEN Expr 1 THEN Result 1 WHEN Expr 2 THEN Result 2
                                    ELSE Default_Result END
```

Illustration

```
SQL> SELECT Ename, Deptno, CASE Deptno WHEN 10 THEN 'ACCOUNTS'
       WHEN 20 THEN 'RESEARCH' WHEN 30 THEN 'SALES'
                                    WHEN 40 THEN 'OPERATIONS' ELSE 'NOT FOUND'
                                          END FROM Emp;
```

Searched CASE Expressions

- The Statement Uses Conditions To Determine The Returned Value.
- It Helps in Writing Multiple Conditions For Evaluation.
- Helps in Range Analysis of Values Also.

Syntax

```
SQL> CASE  WHEN Condition 1 THEN Result 1 WHEN Condition 2 THEN Result 2
                    WHEN Condition n  THEN Resultn ELSE DefaultResult  END
```

Illustration

```
SQL> SELECT Ename, Deptno, CASE   WHEN Deptno = 10 THEN 'ACCOUNTING'
                WHEN Deptno = 20 THEN 'RESEARCH' WHEN Deptno = 30 THEN 'SALES'
             WHEN Deptno = 40 THEN 'OPERATIONS' ELSE 'Not Specified'  END  FROM Emp;
SQL> SELECT Ename, Sal,  CASE WHEN Sal >= 800 AND Sal <= 2000 THEN 'LOWEST PAY'
                WHEN Sal >= 2001 AND Sal <= 4000 THEN 'MODERATE PAY'
                                    ELSE 'HIGH PAY'  END  FROM Emp;
```

Materialized Views

- Materialized Views Are Used in DATA WAREHOUSES AND DATA MARTS.
- They Are Used To Increase The Speed of Queries on Very Large Scale Databases.
- Queries Making Use of Materialized Views Are...
  - Aggregations on A Single Table.
  - Joins Between Tables.
  - Aggregations And Joins Together.
- Materialized Views Can Be Used To Replicate Data.
- Prior To Materialized Views, The Concept of Snapshot Was Implemented.

Query Rewrite

- Materialized Views Improve Query Performance By Pre Calculating Expensive Join And Aggregation Operations on The Database Prior To Execution Time And Stores The Results in The Database.
- The Query Optimizer Can Make Use of Materialized Views By Automatically Recognizing When An Existing Materialized View Can And Should Be Used To Satisfy A Request.
- After Above Process is Completed Then The Query Optimizer Transparently Rewrites The Request To Use The Materialized View.
- Queries Are Then Directed To The Materialized View And Not To The Underlying Detail Tables OR Views.
- Rewriting Queries To Use Materialized Views Rather Than Detail Relations, Results in A Significant Performance Gain.

Prerequisites For Materialized Views

Privileges

```
SQL> GRANT  QUERY REWRITE  TO SCOTT;
```

```
SQL> GRANT CREATE MATERIALIZED VIEW  TO SCOTT;
SQL> ALTER SESSION  SET QUERY_REWRITE_ENABLED = TRUE;
```

Set the InitSid.ORA File

```
OPTIMIZER_MODE = CHOOSE
JOB_QUEUE_INTERVAL = 3600
JOB_QUEUE_PROCESSES = 1
QUERY_REWRITE_ENABLED = TRUE
QUERY_REWRITE_INTEGRITY = ENFORCED
```

Materialized View With Aggregation

```
SQL> CREATE MATERIALIZED VIEW EMP_SUM  ENABLE QUERY REWRITE
            AS SELECT Deptno , Job, SUM(Sal)  FROM Emp GROUP BY Deptno, Job;
```

Creating Optimizer Statistics and Refreshing Materialized Views

```
SQL> EXECUTE DBMS_UTILITY.ANALYZE_SCHEMA ('SCOTT', 'ESTIMATE');
SQL> EXECUTE DBMS_MVIEW.REFRESH('Emp_Sum');
```

Testing Materialized View

```
SQL> SET AUTOTRACE ON EXPLAIN;
SQL> SELECT Deptno, SUM(Sal)  FROM Emp  GROUP BY Deptno, Job;
```

Materialized View with Join/Aggregation

```
SQL> CREATE MATERIALIZED VIEW Emp_Dept_Sum  ENABLE QUERY REWRITE
        AS SELECT Dname, Job, SUM(Sal)  FROM Emp E , Dept D
                                WHERE E.Deptno = D.Deptno GROUP BY Dname, Job;
```

Creating Optimizer Statistics and Refreshing Materialized Views

```
SQL> EXECUTE DBMS_UTILITY.ANALYZE_SCHEMA ('SCOTT', 'ESTIMATE');
SQL> EXECUTEDBMS_MVIEW.REFRESH('Emp_Dept_Sum');
```

Testing Materialized View

```
SQL> SET AUTOTRACE ON EXPLAIN;
SQL> SELECT Dname, Job, SUM(Sal)  FROM Emp E , Dept D  WHERE E.Deptno = D.Deptno
                                                        GROUP BY Dname, Job;
```

Putting the Things with ROLLUP

```
SQL> CREATE MATERIALIZED VIEW Emp_Dept_Agg ENABLE QUERY REWRITE
        AS SELECT Deptno, Job, COUNT(*), SUM(Sal) FROM Emp GROUP BY
                                                        ROLLUP(Deptno, Job);
```

GROUPING_ID() Function

- The Function is Used To FILTER ROWS Using A HAVING Clause To Exclude Rows That Do Not Contain A Subtotal OR Grand Total .
- The Function Accepts One OR More Columns And Returns The Decimal Equivalent of The GROUPING BIT VECTOR.
- The GROUPING BIT VECTOR is Computed By Combining The Results Of A Call To The GROUPING() Function For Each Column in Order.

Computing The GROUPING BIT VECTOR

- GROUPING() Function Returns 1 When The Column Value is NULL, Else Returns 0, Based On This Concept.
- GROUPING_ID() Returns 0, When Deptno And Job Are NOT NULL'S.
- GROUPING_ID() Returns 1, If Deptno is NOT NULL And Job is NULL.
- GROUPING_ID() Returns 2, If Deptno is NULL And Job is NOT NULL.
- GROUPING_ID() Returns 3, If Deptno is NULL And Job is NULL.

```
SQL> SELECT Deptno, Job,  GROUPING(Deptno) GDPT,  GROUPING(Job) GJOB ,
```

```
GROUPING_ID(Deptno, Job) GRPID, SUM(Sal) FROM Emp GROUP BY
                                                    ROLLUP(Deptno, Job);
SQL> SELECT Deptno, Job, GROUPING(Deptno) GDPT, GROUPING(Job) GJOB ,
GROUPING_ID(Deptno, Job) GRPID, SUM(Sal) FROM Emp GROUP BY
                                                    CUBE(Deptno, Job);
```

GROUPING_ID() and HAVING Clause

```
SQL> SELECT Deptno, Job, GROUPING_ID(Deptno, Job) GRPID, SUM(Sal) FROM Emp
GROUP BY CUBE(Deptno , Job) HAVING GROUPING_ID(Deptno, Job) > 0;
```

Representing Column Multiple Times in a GROUP BY Clause

- A Column Can Be Represented Multiple Times in A GROUP BY Clause.

```
SQL> SELECT Deptno, Job, SUM(Sal) FROM Emp GROUP BY Deptno, ROLLUP(Deptno, Job);
SQL> SELECT Deptno, Job, SUM(Sal) FROM Emp GROUP BY Deptno, CUBE(Deptno, Job);
```

Applying GROUP_ID Function

- The GROUP_ID() Function is Used To Remove The Duplicate Rows Returned By GROUP BY Clause.
- The GROUP_ID() Does Not Accept Any Parameters.
- If 'N' Duplicates Exist For A Particular Grouping, GROUP_ID() Returns Numbers in The Range 0 To N – 1.

```
SQL> SELECT Deptno, Job, GROUP_ID(), SUM(Sal) FROM Emp GROUP BY Deptno,
                                                    ROLLUP(Deptno, Job);
SQL> SELECT Deptno, Job, GROUP_ID(), SUM(Sal) FROM Emp GROUP BY Deptno,
                                                    CUBE(Deptno, Job);
SQL> SELECT Deptno, Job, GROUP_ID(), SUM(Sal) FROM Emp GROUP BY Deptno,
                                    ROLLUP(Deptno, Job) HAVING GROUP_ID() = 0;
```

Enhancing The Power of OLAP Using Analytic Functions

- Analytic Functions Are Designed To Address Problems Such As
  - Calculate A Running Total.
  - Find Percentages With in A Group.
  - Top "N" Queries.
  - Compute "Moving Averages".
- Analytic Functions Add Greater Performance To The Standard Query Processing.

How Analytic Functions Work?

- Analytic Functions Compute An Aggregate Value Based on A Group of Rows.
- The Differences From Ordinary Aggregate Functions To Analytic Functions Are, They Return Multiple Rows For Each Group.
- The Group of Rows Are Called As "WINDOW" And is Defined By The Analytic Clause.
- For Each Row, A Sliding Window of Rows Are Defined.
- The Window Define The Range of Rows Used To Perform The Calculation For The Current Row.
- Window Sizes Can Be Based Upon Either A Physical Number of Rows OR A Logical Interval Such As Time.
- Analytic Functions Are The Last Set Of Operations Performed in A Query, Except For The Final ORDER BY Clause.
- All JOINS, WHERE Clause, GROUP BY, And HAVING Clauses Are Completed Before The Analytic Functions Are Processed.
- Analytic Functions Can Appear Only in The SELECT List OR ORDER BY Clause.

Syntax

```
AnalyticFunction(Arg 1, Arg2, Ar3)
OVER(Partition Clause
        ORDER BY Clause
        Windowing Clause)
```

- Analytic Function Takes 0 To 3 Arguments.
- The PARTITION BY Clause Logically Breaks A Single Result Set Into 'N' Groups.
- The Analytic Functions Are Applied For Each Group Independently, And They Are Reset For Each Group.
- The ORDER BY Clause Specifies How Data is Stored Within Each GROUP (Partition).
- The Output of Analytic Function is Affected By ORDER BY Clause.
- The Windowing Clause Gives Us A Way To Define A Sliding (OR) Analytic Function Will Operate, Within A Group.

Analytic Functions Categories

Ranking Functions

- They Enable US To Calculate Ranks, Percentiles and N-Tiles.

Inverse Percentile Functions

- Enable To Calculate The Value Corresponding To A Percentile.

Window Functions

- Enable To Calculate Cumulative And Moving Aggregates.

Reporting Functions

- Enable To Calculate Area Like Market Shares.

LAG and LEAD Functions

- Enable To Get A Value in A Row Where That Row is A Certain Number of Rows Away From The Current Row.

First and Last Functions

- Enable To Get The First And Last Values in An Ordered Group.

Linear Regression Functions

- Enable To Fit An Ordinary–Least– Squares Regression Line To A Set of Number Pairs.

Hypothetical Rank and Distribution Functions

- Enable To Calculate The Rank And Percentile That A New Row Would Have If A Value is Inserted Into A Table.

Normal Ranking

```
SQL> SELECT EName, Deptno, Sal, RANK() OVER(ORDER BY Sal) EmpRank FROM Emp
                                GROUP BY Deptno, EName, Sal ORDER By Emprank;
SQL> SELECT EName, Deptno, Sal, DENSE_RANK() OVER(ORDER BY Sal DESC) EmpRank
          FROM Emp GROUP BY Deptno, EName, Sal ORDER BY EmpRank;
```

Ranking With Partition

```
SQL> SELECT EName, Deptno, Sal, RANK() OVER(PARTITION BY DeptNo
          ORDER BY Sal DESC) "TOP Sal" FROM Emp ORDER BY Deptno, Sal DESC;
SQL> SELECT EName, DeptNo, Sal, DENSE_RANK() OVER(PARTITION BY Deptno
        ORDER BY Sal DESC) "TOP Sal" FROM Emp ORDER BY DeptNo, Sal DESC;
```

Ranking With Partition And Filters

```
SQL> SELECT * FROM(SELECT Ename, Deptno, Sal, RANK() OVER(PARTITION BY DeptNo
          ORDER BY Sal DESC) "TOP Sal" FROM Emp) WHERE "TOP Sal" <=3
                                                ORDER BY DeptNo, Sal DESC;
```

Applying Windows

- The Windowing Clause Gives A Way To Define A SLIDING OR ANCHORED WINDOW of Data, Upon Which The ANALYTIC FUNCTION Will Operate, Within A GROUP.
- The Default Window is An ANCHORED WINDOW That Simply Starts At The FIRST ROW of A GROUP And Continues To The CURRENT ROW.
- Window Can Be Based on
  - RANGES of Data Values.
  - ROWS OFFSET From The Current Row.

- Existence of An ORDER BY in An Analytic Function Will Add A DEFAULT WINDOW Clause of "RANGE UNBOUNDED PRECEDING", Which Gets All Rows in The Partition That Came Before The ORDER BY Clause.

Row WINDOW

```
SQL> SELECT DeptNo, Ename, Sal,  SUM(SAL)  OVER(PARTITION BY DeptNo
                ORDER BY EName   ROWS 2 PRECEDING) "Sliding Total" FROM Emp
                                                        ORDER BY DeptNo, EName;
```

Range WINDOW

- RANGE WINDOWS Collect ROWS Together Based on A WHERE Clause.
- The RANGE UNITS Can Either Be Numeric Comparisons OR Date Comparisons.
- Range Units Are Not Valid if Data Type is Other Than Number OR Dates.

```
SQL> SELECT Ename,  HireDate, HireDate – 100, Count (*) OVER (ORDER BY HireDate ASC
                        RANGE 100 PRECEDING) HireCnt FROM Emp ORDER BY HireDate ASC;
SQL> SELECT Ename, HireDate, Sal,  AVG(Sal) OVER(ORDER BY HireDate ASC
                        RANGE 100 PRECEDING) AvgSal FROM Emp ORDER BY Hire Data ASC;
```

Accessing ROWS Around Current Row

LAG Function

- Lag Provides Access To More Than One Row of A Table At The Same Time Without The Use of SELF JOIN.
- Given A Series of Rows Returned From A Query And A Position of The Cursor, LAG Function Provides Access To A Row At A Given Offset Prior To That Position.
- If offset is Not Provided Than Default Offset is Considered As 1.
- The Optional Default Value is Returned if The Offset Goes Beyond The Scope of The Window.
- If Default is Not Specified, Than The Default is Considered As NULL.

Syntax

```
LAG(ValueExpr(, Offset)(, DEFAULT))
OVER((Query-Partition-Clause)
ORDER BY CLAUSE)
```

LEAD Function

- LEAD Provides Access To More Than One Row of A Table At The Same Time Without The Use of A SELF JOIN.
- Given A Series of Rows Returned From A Query And A Position of The Cursor, LEAD Function Provides Access To A Row At A Given Physical Offset Beyond That Position.

Syntax

```
LEAD(ValueExpr(, Offset)(, Default))
OVER((Query-Partition_Clause)
ORDER BY Clause)
```

Illustration

```
SQL> SELECT  Ename, HireDate, Sal,  LAG(Sal, 1, 0) OVER(ORDER BY HireDate) PreSal
                                                        FROM  Emp;
SQL> SELECT  Ename, HireDate, Sal, LEAD(Sal, 1, 0) OVER(ORDER BY HireDate) NextSal
                                                        FROM  Emp;
```

FIRST_VALUE Function

- FIRST_VALUE Returns The First Value in An Ordered Set of Values.
- If The First Value in The Set is NULL, Then The Function Returns NULL Unless IGNORE NULLS is Specified.
- IGNORE NULLS Setting is Useful For Data Densification. If Specified Then FIRST_VALUE Returns The Fist Non-Null Value in The Set, Else NULL is Returned.

Syntax

```
FIRST_VALUE(EXPR IGNORE NULLS)
```

OVER(Analytic_Clause)

LAST_VALUE Function

- LAST_VALUE Returns The LAST Value in An Ordered Set of Values.
- If The Last Value in The Set is NULL, Then The Function Returns NULL Unless IGNORE NULLS is Specified.
- IGNORE NULLS Setting is Useful For Data Densification. If Specified Then LAST_VALUE Returns The Fist Non-Null Value in The Set, Else NULL is Returned.

Syntax

```
LAST_VALUE(EXPR IGNORE NULLS)
OVER(Analytic_Clause )
```

Illustration

```
SQL> SELECT Ename, Deptno, Sal, FIRST_VALUE(Ename) OVER(PARTITION BY DeptNo
       ORDER BY Sal DESC)  Max_Sal_Name FROM   Emp ORDER BY DeptNo,  Ename;

SQL> SELECT Ename, Deptno, Sal, FIRST_VALUE(Ename) OVER(ORDER BY Sal ASC)
                    Min_Sal_Name FROM (SELECT  * FROM  Emp WHERE Deptno = 30);
```

ROW_NUMBER Function

- The ROW_NUMBER Function Assigns a Unique Number Sequentially, Starting From 1, As Defined By ORDER BY To Each Row Within The Partition.

Syntax

```
ROW_NUMBER() OVER([Query_Partition_Clause]  ORDER_BY_Clause )

SQL> SELECT  ROW_NUMBER() OVER(PARTITION BY Deptno
         ORDER BY Sal DESC NULLS LAST) RowNo, Ename, Sal, Deptno,  FROM  Emp;
```

CROSSTAB OR PIVOT Queries

- A CROSSTAB Query Can Be Used To Get A Result With Rows And Columns in A Matrix Form.

```
SQL> SELECT Deptno, MAX(DECODE(Seqno, 1, Ename, NULL)) First,
       MAX(DECODE(Seqno, 2, Ename, NULL)) Second,
            MAX(DECODE(Seqno, 3, Ename, NULL) Third FROM  (SELECT Deptno,
                 Ename, ROW_NUMBER ( ) OVER (PARTITION BY Deptno
                      ORDER BY Sal DESC NULLS LAST)  SeqNo
                           FROM  Emp) WHERE SeqNo <= 3 GROUP BY Deptno;
```

Centered SUM And Centered AVERAGE

```
SQL> SELECT Ename, SUM(Sal) SalAmt, ROUND(SUM(SUM(Sal)) OVER(ORDER BY Deptno
            ROWS BETWEEN  1 PRECEDING AND 1 FOLLOWING ), 2) CSum,
                 ROUND(AVG(SUM(Sal)) OVER(ORDER BY Deptno  ROWS BETWEEN
                      1 PRECEDING AND 1 FOLLOWING ), 2) CAvg
                           FROM Emp GROUP BY Deptno, Ename ORDER BY Deptno;
```

Extra Stuff For Practice on Analytic Functions

Create The Following Tables in The Same Order

```
SQL> CREATE TABLE ItemTypes (  ItemTypeID INTEGER  CONSTRAINT ItemTypePK
                                    PRIMARY KEY,  ItemTName VARCHAR2(10)
       CONSTRAINT ItemTNameNN NOT NULL );

SQL> CREATE TABLE Items (  ItemID INTEGER  CONSTRAINT ItemIDPK PRIMARY KEY,
          ItemTypeID INTEGER  CONSTRAINT ItemTypeIDFK  REFERENCES
           ItemTypes(ItemTypeID),  ItemName VARCHAR2(10)  CONSTRAINT
            ItemNameNN NOT NULL,  Description VARCHAR2(100), Price NUMBER(5, 2) );

SQL> CREATE TABLE Divisions (  DivID CHAR(3)  CONSTRAINT DivIDPK PRIMARY KEY,
            DivName  VARCHAR2(15)  CONSTRAINT DivNameNN NOT NULL);
```

```
SQL> CREATE TABLE JobMaster ( JobID  CHAR(3)  CONSTRAINT JobIDPK PRIMARY KEY,
                JobName VARCHAR2(20)  CONSTRAINT JobNameNN NOT NULL );

SQL> CREATE TABLE EmpStores(EmpID INTEGER CONSTRAINT EmpSoresPK
          PRIMARY KEY, DivID  CHAR(3)  CONSTRAINT DivIDFK REFERENCES
          Divisions(DivID), JobID CHAR(3)  CONSTRAINT JobIDFK REFERENCES
          JobMaster(JobID), FirstName  VARCHAR2(10)  CONSTRAINT FNameNN NOT NULL,
        LastName VARCHAR2(10)  CONSTRAINT LNameNN NOT NULL, Sal NUMBER(6, 0));

SQL> CREATE TABLE AllSales ( Year INTEGER CONSTRAINT YearNN NOT NULL,
          Month INTEGER CONSTRAINT MonthNN NOT NULL,  ItemTypeID INTEGER
              CONSTRAINT ItemTIDAS  REFERENCES ItemTypes(ItemTypeID),
                EmpID  INTEGER  CONSTRAINT EmpIDAS REFERENCES
                    EmpStores(EmpID),  SaleAmt  NUMBER(8, 2), CONSTRAINT AllSalesPk
                        PRIMARY KEY (Year, Month, ItemTypeID, EmpID));
```

Insert Sample Data Into Itemtypes Table

```
SQL> INSERT INTO ItemTypes (ItemTypeID, ItemTName) VALUES (1, 'Book');
SQL> INSERT INTO ItemTypes (ItemTypeID, ItemTName) VALUES (2, 'Video');
SQL> INSERT INTO ItemTypes (ItemTypeID, ItemTName) VALUES (3, 'DVD');
SQL> INSERT INTO ItemTypes (ItemTypeID, ItemTname) VALUES (4, 'CD');
SQL> INSERT INTO ItemTypes (ItemTypeID, ItemTname)VALUES (5, 'Magazine');
SQL> COMMIT;
```

Insert Sample Data Into Products Table

```
SQL> INSERT INTO  Items (ItemID, ItemTypeID, itemName, Description, Price) VALUES
                        (1, 1, 'Modern Science', 'A Desc of modern science', 19.95);
SQL> INSERT INTO  Items (ItemID, ItemTypeID, itemName, Description, Price) VALUES
                        (2, 1, 'Chemistry', 'Introduction to Chemistry', 30.00);
SQL> INSERT INTO Items (ItemID, ItemTypeID, itemName, Description, Price) VALUES
                        (3, 2, 'Supernova', 'A star explodes', 25.99);
SQL> INSERT INTO  Items (ItemID, ItemTypeID, itemName, Description, Price) VALUES
                        (4, 2, 'Tank War', 'Action movie about a future war', 13.95);
SQL> INSERT INTO  Items (ItemID, ItemTypeID, itemName, Description, Price) VALUES
                        (5, 2, 'Z Files', 'Series on mysterious activities', 49.99);
SQL> INSERT INTO  Items (ItemID, ItemTypeID, itemName, Description, Price) VALUES
                        (6, 2, '2412: The Return', 'Aliens return', 14.95);
SQL> INSERT INTO  Items (ItemID, ItemTypeID, itemName, Description, Price) VALUES
                       (7, 3, 'Space Force 9', 'Adventures of heroes', 13.49);
SQL> INSERT INTO  Items (ItemID, ItemTypeID, itemName, Description, Price) VALUES
                  (8, 3, 'From Another Planet', 'Alien from another planet lands on Earth', 12.99);
SQL> INSERT INTO  Items (ItemID, ItemTypeID, itemName, Description, Price) VALUES
                        (9, 4, 'Classical Music', 'The best classical music', 10.99);
SQL> INSERT INTO  Items (ItemID, ItemTypeID, itemName, Description, Price) VALUES
                        (10, 4, 'Pop 3', 'The best popular music', 15.99);
SQL> INSERT INTO  Items (ItemID, ItemTypeID, itemName, Description, Price) VALUES
                        (11, 4, 'Creative Yell', 'Debut album', 14.99);
SQL> INSERT INTO  Items (ItemID, ItemTypeID, itemName, Description, Price) VALUES
                         (12, NULL, 'My Front Line', 'Their greatest hits', 13.49);
SQL> COMMIT;
```

Insert Sample Data Into Divisions Table

```
SQL> INSERT INTO Divisions(DivID, DivName)  VALUES('SAL', 'Sales');
SQL> INSERT INTO Divisions(DivID, DivName) VALUES ('OPE', 'Operations');
SQL> INSERT INTO Divisions(DivID, DivName) VALUES ('SUP', 'Support');
SQL> INSERT INTO Divisions(DivID, DivName) VALUES ('BUS', 'Business');
```

```
SQL> COMMIT;
```

Insert Sample Data Into Jobs Table

```
SQL> INSERT INTO JobMaster(JobID, JobName)  VALUES ('WOR', 'Worker');
SQL> INSERT INTO JobMaster(JobID, JobName) VALUES ('MGR', 'Manager');
SQL> INSERT INTO JobMaster(JobID, JobName) VALUES ('ENG', 'Engineer');
SQL> INSERT INTO JobMaster(JobID, JobName) VALUES ('TEC', 'Technologist');
SQL> INSERT INTO JobMaster(JobID, JobName) VALUES ('PRE', 'President');
SQL> COMMIT;
```

Insert Sample Data Into Empstores Table

```
SQL> INSERT INTO  EmpStores  (EmpID, DivID, JobID, FirstName, LastName, Sal)  VALUES
                              (1, 'BUS', 'PRE', 'James', 'Smith', 800000);
SQL> INSERT INTO  EmpStores  (EmpID, DivID, JobID, FirstName, LastName, Sal) VALUES
                              (2, 'SAL', 'MGR', 'Ron', 'Johnson', 350000);
SQL> INSERT INTO EmpStores  (EmpID, DivID, JobID, FirstName, LastName, Sal) VALUES
                              (3, 'SAL', 'WOR', 'Fred', 'Hobbs', 140000);
SQL> INSERT INTO  EmpStores  (EmpID, DivID, JobID, FirstName, LastName, Sal) VALUES
                              ( 4, 'SUP', 'MGR', 'Susan', 'Jones', 200000);
SQL> INSERT INTO EmpStores (EmpID, DivID, JobID, FirstName, LastName, Sal) VALUES
                             (5, 'SAL', 'WOR', 'Rob', 'Green', 350000);
SQL> INSERT INTO  EmpStores  (EmpID, DivID, JobID, FirstName, LastName, Sal) VALUES
                              (6, 'SUP', 'WOR', 'Jane', 'Brown', 200000);
SQL> INSERT INTO  EmpStores  (EmpID, DivID, JobID, FirstName, LastName, Sal) VALUES
                              (7, 'SUP', 'MGR', 'John', 'Grey', 265000);
SQL> INSERT INTO  EmpStores  (EmpID, DivID, JobID, FirstName, LastName, Sal) VALUES
                              (8, 'SUP', 'WOR', 'Jean', 'Blue', 110000);
SQL> INSERT INTO  EmpStores  (EmpID, DivID, JobID, FirstName, LastName, Sal) VALUES
                              (9, 'SUP', 'WOR', 'Henry', 'Heyson', 125000);
SQL> INSERT INTO  EmpStores  (EmpID, DivID, JobID, FirstName, LastName, Sal) VALUES
                              (10, 'OPE', 'MGR', 'Kevin', 'Black', 225000);
SQL> INSERT INTO  EmpStores  (EmpID, DivID, JobID, FirstName, LastName, Sal) VALUES
                             (11, 'OPE', 'MGR', 'Keith', 'Long', 165000);
SQL> INSERT INTO  EmpStores  (EmpID, DivID, JobID, FirstName, LastName, Sal)  VALUES
                             (12, 'OPE', 'WOR', 'Frank', 'Howard', 125000);
SQL> INSERT INTO  EmpStores  (EmpID, DivID, JobID, FirstName, LastName, Sal) VALUES
                             (13, 'OPE', 'WOR', 'Doreen', 'Penn', 145000);
SQL> INSERT INTO  EmpStores  (EmpID, DivID, JobID, FirstName, LastName, Sal) VALUES
                             (14, 'BUS', 'MGR', 'Mark', 'Smith', 155000);
SQL> INSERT INTO  EmpStores  (EmpID, DivID, JobID, FirstName, LastName, Sal) VALUES
                             (15, 'BUS', 'MGR', 'Jill', 'Jones', 175000);
SQL> INSERT INTO  EmpStores  (EmpID, DivID, JobID, FirstName, LastName, Sal) VALUES
                             (16, 'OPE', 'ENG', 'Megan', 'Craig', 245000);
SQL> INSERT INTO EmpStores  (EmpID, DivID, JobID, FirstName, LastName, Sal)VALUES
                             (17, 'SUP', 'TEC', 'Matthew', 'Brant', 115000);

SQL> INSERT INTO  EmpStores  (EmpID, DivID, JobID, FirstName, LastName, Sal) VALUES
                             (18, 'OPE', 'MGR', 'Tony', 'Clerke', 200000);
SQL> INSERT INTO  EmpStores  (EmpID, DivID, JobID, FirstName, LastName, Sal) VALUES
                             (19, 'BUS', 'MGR', 'Tanya', 'Conway', 200000);
SQL> INSERT INTO  EmpStores  (EmpID, DivID, JobID, FirstName, LastName, Sal)VALUES
                             (20, 'OPE', 'MGR', 'Terry', 'Cliff', 215000);
SQL> INSERT INTO EmpStores (EmpID, DivID, JobID, FirstName, LastName, Sal) VALUES
                             (21, 'SAL', 'MGR', 'Steve', 'Green', 275000);
SQL> INSERT INTO EmpStores (EmpID, DivID, JobID, FirstName, LastName, Sal) VALUES
```

```
                    (22, 'SAL', 'MGR', 'Roy', 'Red', 375000);
SQL> INSERT INTO EmpStores  (EmpID, DivID, JobID, FirstName, LastName, Sal) VALUES
                    (23, 'SAL', 'MGR', 'Sandra', 'Smith', 335000);
SQL> INSERT INTO EmpStores  (EmpID, DivID, JobID, FirstName, LastName, Sal) VALUES
                    (24, 'SAL', 'MGR', 'Gail', 'Silver', 225000);
SQL> INSERT INTO EmpStores (EmpID, DivID, JobID, FirstName, LastName, Sal) VALUES
                    (25, 'SAL', 'MGR', 'Gerald', 'Gold', 245000);
SQL> INSERT INTO EmpStores (EmpID, DivID, JobID, FirstName, LastName, Sal) VALUES
                    (26, 'SAL', 'MGR', 'Eileen', 'Lane', 235000);
SQL> INSERT INTO EmpStores  (EmpID, DivID, JobID, FirstName, LastName, Sal) VALUES
                    (27, 'SAL', 'MGR', 'Doreen', 'Upton', 235000);
SQL> INSERT INTO EmpStores (EmpID, DivID, JobID, FirstName, LastName, Sal) VALUES
                    (28, 'SAL', 'MGR', 'Jack', 'Ewing', 235000);
SQL> INSERT INTO EmpStores (EmpID, DivID, JobID, FirstName, LastName, Sal) VALUES
                    (29, 'SAL', 'MGR', 'Paul', 'Owens', 245000);
SQL> INSERT INTO EmpStores (EmpID, DivID, JobID, FirstName, LastName, Sal) VALUES
                    (30, 'SAL', 'MGR', 'Melanie', 'York', 255000);
SQL> INSERT INTO EmpStores (EmpID, DivID, JobID, FirstName, LastName, Sal) VALUES
                    (31, 'SAL', 'MGR', 'Tracy', 'Yellow', 225000);
SQL> INSERT INTO EmpStores (EmpID, DivID, JobID, FirstName, LastName, Sal) VALUES
                    (32, 'SAL', 'MGR', 'Sarah', 'White', 235000);
SQL> INSERT INTO EmpStores  (EmpID, DivID, JobID, FirstName, LastName, Sal) VALUES
                    (33, 'SAL', 'MGR', 'Terry', 'Iron', 225000);
SQL> INSERT INTO EmpStores (EmpID, DivID, JobID, FirstName, LastName, Sal) VALUES
                    (34, 'SAL', 'MGR', 'Christine', 'Brown', 247000);
SQL> INSERT INTO EmpStores (EmpID, DivID, JobID, FirstName, LastName, Sal) VALUES
                    (35, 'SAL', 'MGR', 'John', 'Brown', 249000);
SQL> INSERT INTO EmpStores (EmpID, DivID, JobID, FirstName, LastName, Sal) VALUES
                    (36, 'SAL', 'MGR', 'Kelvin', 'Trenton', 255000);
SQL> INSERT INTO EmpStores (EmpID, DivID, JobID, FirstName, LastName, Sal) VALUES
                    (37, 'BUS', 'WOR', 'Damon', 'Jones', 280000);
SQL> COMMIT;
```

Insert Sample Data Into Allsales Table

```
SQL> INSERT INTO  AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)  VALUES
                    (21, 1, 2003, 1, 10034.84);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)  VALUES
                    (21, 1, 2003, 2, 15144.65);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)  VALUES
                    (21, 1, 2003, 3, 20137.83);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)  VALUES
                    (21, 1, 2003, 4, 25057.45);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)  VALUES
                    (21, 1, 2003, 5, 17214.564);
SQL> INSERT INTO  AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                    (21, 1, 2003, 6, 15564.64);
SQL> INSERT INTO  AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                    (21, 1, 2003, 7, 12654.84);
SQL> INSERT INTO  AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)  VALUES
                    (21, 1, 2003, 8, 17434.82);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)  VALUES
                    (21, 1, 2003, 9, 19854.57);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)  VALUES
                    (21, 1, 2003, 10, 21754.19);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
```

```
                (21, 1, 2003, 11, 13029.73);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (21, 1, 2003, 12, 10034.84);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (21, 2, 2003, 1, 1034.84);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)  VALUES
                (21, 2, 2003, 2, 1544.65);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (21, 2, 2003, 3, 2037.83);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (21, 2, 2003, 4, 2557.45);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (21, 2, 2003, 5, 1714.564);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (21, 2, 2003, 6, 1564.64);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)  VALUES
                (21, 2, 2003, 7, 1264.84);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)  VALUES
                (21, 2, 2003, 8, 1734.82);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)  VALUES
                (21, 2, 2003, 9, 1854.57);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (21, 2, 2003, 10, 2754.19);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)  VALUES
                (21, 2, 2003, 11, 1329.73);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)  VALUES
                (21, 2, 2003, 12, 1034.84);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (21, 3, 2003, 1, 6034.84);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (21, 3, 2003, 2, 1944.65);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)  VALUES
                (21, 3, 2003, 3, 2537.83);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)  VALUES
                (21, 3, 2003, 4, 4557.45);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)  VALUES
                (21, 3, 2003, 5, 3714.564);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)  VALUES
                (21, 3, 2003, 6, 3564.64);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (21, 3, 2003, 7, 21264.84);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)  VALUES
                (21, 3, 2003, 8, 21734.82);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (21, 3, 2003, 9, 12854.57);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (21, 3, 2003, 10, 32754.19);
SQL> INSERT INTO  AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)  VALUES
                ( 21, 3, 2003, 11, 15329.73);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)  VALUES
                (21, 3, 2003, 12, 14034.84);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)  VALUES
                (21, 4, 2003, 1, 3034.84);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)  VALUES
                (21, 4, 2003, 2, 2944.65);
```

```
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)  VALUES
                (21, 4, 2003, 3, 5537.83);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (21, 4, 2003, 4, 3557.45);
SQL> INSERT INTO  AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (21, 4, 2003, 5, 2714.564);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (21, 4, 2003, 6, 7564.64);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (21, 4, 2003, 7, 1264.84);

SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (21, 4, 2003, 8, 21734.82);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (21, 4, 2003, 9, 14854.57);
SQL> INSERT INTO  AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (21, 4, 2003, 10, 22754.19);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (21, 4, 2003, 11, 11329.73);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (21, 4, 2003, 12, 11034.84);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (22, 1, 2003, 1, 11034.84);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (22, 1, 2003, 2, 16144.65);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (22, 1, 2003, 3, 24137.83);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (22, 1, 2003, 4, 29057.45);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (22, 1, 2003, 5, 19214.564);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (22, 1, 2003, 6, 16564.64);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)  VALUES
                (22, 1, 2003, 7, 13654.84);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)  VALUES
                (22, 1, 2003, 8, 17834.82);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)  VALUES
                (22, 1, 2003, 9, 21854.57);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (22, 1, 2003, 10, 18754.19);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (22, 1, 2003, 11, 16529.73);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (22, 1, 2003, 12, 9434.84);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (22, 2, 2003, 1, 1234.84);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (22, 2, 2003, 2, 1044.65);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (22, 2, 2003, 3, 2537.83);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (22, 2, 2003, 4, 2657.45);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (22, 2, 2003, 5, 1314.56);
```

```
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (22, 2, 2003, 6, 1264.64);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (22, 2, 2003, 7, 1964.84);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (22, 2, 2003, 8, 1234.82);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (22, 2, 2003, 9, 1954.57);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (22, 2, 2003, 10, 2254.19);

SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (22, 2, 2003, 11, 1229.73);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (22, 2, 2003, 12, 1134.84);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (22, 3, 2003, 1, 6334.84);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (22, 3, 2003, 2, 1544.65);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (22, 3, 2003, 3, 2737.83);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (22, 3, 2003, 4, 4657.45);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (22, 3, 2003, 5, 3714.56);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (22, 3, 2003, 6, 3864.64);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (22, 3, 2003, 7, 27264.84);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (22, 3, 2003, 8, 17734.82);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (22, 3, 2003, 9, 10854.57);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (22, 3, 2003, 10, 15754.19);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (22, 3, 2003, 11, 10329.73);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (22, 3, 2003, 12, 12034.84);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (22, 4, 2003, 1, 3334.84);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (22, 4, 2003, 2, 2344.65);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (22, 4, 2003, 3, 5137.83);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (22, 4, 2003, 4, 3157.45);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (22, 4, 2003, 5, 2114.564);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (22, 4, 2003, 6, 7064.64);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (22, 4, 2003, 7, 1564.84);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (22, 4, 2003, 8, 12734.82);
```

```
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (22, 4, 2003, 9, 10854.57);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (22, 4, 2003, 10, 20754.19);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (22, 4, 2003, 11, 10329.73);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (22, 4, 2003, 12, 2034.84);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (23, 1, 2003, 1, 4034.84);

SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (23, 1, 2003, 2, 7144.65);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (23, 1, 2003, 3, 12137.83);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (23, 1, 2003, 4, 16057.45);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (23, 1, 2003, 5, 13214.564);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (23, 1, 2003, 6, 3564.64);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (23, 1, 2003, 7, 7654.84);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (23, 1, 2003, 8, 5834.82);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (23, 1, 2003, 9, 6754.57);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (23, 1, 2003, 10, 12534.19);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (23, 1, 2003, 11, 2529.73);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (23, 1, 2003, 12, 7434.84);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (23, 2, 2003, 1, 1234.84);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (23, 2, 2003, 2, 2244.65);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (23, 2, 2003, 3, 2137.83);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (23, 2, 2003, 4, 2357.45);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (23, 2, 2003, 5, 1314.56);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (23, 2, 2003, 6, 1364.64);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (23, 2, 2003, 7, 1364.84);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (23, 2, 2003, 8, 1334.82);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (23, 2, 2003, 9, 1354.57);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (23, 2, 2003, 10, 2354.19);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (23, 2, 2003, 11, 1329.73);
```

```
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (23, 2, 2003, 12, 1334.84);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (23, 3, 2003, 1, 6334.84);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (23, 3, 2003, 2, 1344.65);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (23, 3, 2003, 3, 2337.83);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (23, 3, 2003, 4, 4357.45);

SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (23, 3, 2003, 5, 3314.56);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (23, 3, 2003, 6, 3364.64);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (23, 3, 2003, 7, 23264.84);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (23, 3, 2003, 8, 13734.82);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (23, 3, 2003, 9, 13854.57);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (23, 3, 2003, 10, 13754.19);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (23, 3, 2003, 11, 13329.73);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (23, 3, 2003, 12, 13034.84);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (23, 4, 2003, 1, 3334.84);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (23, 4, 2003, 2, 2344.65);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (23, 4, 2003, 3, 5337.83);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (23, 4, 2003, 4, 3357.45);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (23, 4, 2003, 5, 2314.564);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (23, 4, 2003, 6, 7364.64);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (23, 4, 2003, 7, 1364.84);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (23, 4, 2003, 8, 13734.82);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (23, 4, 2003, 9, 13854.57);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (23, 4, 2003, 10, 23754.19);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (23, 4, 2003, 11, 13329.73);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (23, 4, 2003, 12, 2334.84);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (24, 1, 2003, 1, 7034.84);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (24, 1, 2003, 2, 17144.65);
```

```
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (24, 1, 2003, 3, 22137.83);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (24, 1, 2003, 4, 24057.45);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (24, 1, 2003, 5, 25214.564);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (24, 1, 2003, 6, 14564.64);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (24, 1, 2003, 7, 17654.84);

SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (24, 1, 2003, 8, 15834.82);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (24, 1, 2003, 9, 15854.57);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (24, 1, 2003, 10, 22754.19);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (24, 1, 2003, 11, 14529.73);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (24, 1, 2003, 12, 10434.84);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (24, 2, 2003, 1, 1934.84);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (24, 2, 2003, 2, 2844.65);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (24, 2, 2003, 3, 2837.83);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (24, 2, 2003, 4, 2697.45);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (24, 2, 2003, 5, 7314.56);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (24, 2, 2003, 6, 1864.64);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (24, 2, 2003, 7, 2364.84);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (24, 2, 2003, 8, 4334.82);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (24, 2, 2003, 9, 6654.57);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (24, 2, 2003, 10, 2254.19);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (24, 2, 2003, 11, 5429.73);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (24, 2, 2003, 12, 3334.84);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (24, 3, 2003, 1, 2334.84);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (24, 3, 2003, 2, 4544.65);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (24, 3, 2003, 3, 6337.83);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (24, 3, 2003, 4, 3357.45);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (24, 3, 2003, 5, 2314.56);
```

```
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (24, 3, 2003, 6, 1364.64);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (24, 3, 2003, 7, 5264.84);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (24, 3, 2003, 8, 1734.82);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (24, 3, 2003, 9, 1854.57);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (24, 3, 2003, 10, 1354.19);

SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (24, 3, 2003, 11, 1332.73);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (24, 3, 2003, 12, 3034.84);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (24, 4, 2003, 1, 3364.84);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)VALUES
                (24, 4, 2003, 2, 4344.65);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (24, 4, 2003, 3, 4337.83);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)VALUES
                (24, 4, 2003, 4, 2357.45);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (24, 4, 2003, 5, 6314.564);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (24, 4, 2003, 6, 4364.64);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (24, 4, 2003, 7, 2364.84);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (24, 4, 2003, 8, 3734.82);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (24, 4, 2003, 9, 3854.57);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (24, 4, 2003, 10, 3754.19);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (24, 4, 2003, 11, 1329.73);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (24, 4, 2003, 12, 7334.84);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (25, 1, 2003, 1, 1234.84);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (25, 1, 2003, 2, 6144.65);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (25, 1, 2003, 3, 8137.83);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (25, 1, 2003, 4, 14057.45);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (25, 1, 2003, 5, 12214.564);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (25, 1, 2003, 6, 4564.64);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (25, 1, 2003, 7, 5654.84);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (25, 1, 2003, 8, 8834.82);
```

```
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (25, 1, 2003, 9, 10854.57);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (25, 1, 2003, 10, 12754.19);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (25, 1, 2003, 11, 3529.73);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (25, 1, 2003, 12, 5434.84);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (25, 2, 2003, 1, 3434.84);

SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (25, 2, 2003, 2, 1844.65);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (25, 2, 2003, 3, 2137.83);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (25, 2, 2003, 4, 1697.45);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (25, 2, 2003, 5, 4314.56);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (25, 2, 2003, 6, 3264.64);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (25, 2, 2003, 7, 5364.84);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (25, 2, 2003, 8, 3334.82);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (25, 2, 2003, 9, 2654.57);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (25, 2, 2003, 10, 454.19);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (25, 2, 2003, 11, 2429.73);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (25, 2, 2003, 12, 1334.84);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (25, 3, 2003, 1, 1434.84);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (25, 3, 2003, 2, 3544.65);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (25, 3, 2003, 3, 1337.83);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (25, 3, 2003, 4, 1457.45);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (25, 3, 2003, 5, 1314.56);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (25, 3, 2003, 6, 4364.64);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (25, 3, 2003, 7, 1264.84);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (25, 3, 2003, 8, 2734.82);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (25, 3, 2003, 9, 4354.57);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (25, 3, 2003, 10, 2354.19);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (25, 3, 2003, 11, 3432.73);
```

```
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (25, 3, 2003, 12, 1534.84);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (25, 4, 2003, 1, 1164.84);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (25, 4, 2003, 2, 2144.65);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (25, 4, 2003, 3, 4337.83);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (25, 4, 2003, 4, 1357.45);

SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (25, 4, 2003, 5, 2314.564);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (25, 4, 2003, 6, 2364.64);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (25, 4, 2003, 7, 3264.84);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (25, 4, 2003, 8, 2234.82);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (25, 4, 2003, 9, 3454.57);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (25, 4, 2003, 10, 2754.19);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (25, 4, 2003, 11, 3429.73);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (25, 4, 2003, 12, 4334.84);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (26, 1, 2003, 1, 5534.84);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (26, 1, 2003, 2, 8844.65);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (26, 1, 2003, 3, 5137.83);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (26, 1, 2003, 4, 12057.45);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (26, 1, 2003, 5, 10214.564);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (26, 1, 2003, 6, 2564.64);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (26, 1, 2003, 7, 3654.84);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (26, 1, 2003, 8, 9834.82);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (26, 1, 2003, 9, 9854.57);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (26, 1, 2003, 10, 16754.27);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (26, 1, 2003, 11, 5529.73);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (26, 1, 2003, 12, 3434.84);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (26, 2, 2003, 1, 5434.84);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (26, 2, 2003, 2, 3844.65);
```

```
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (26, 2, 2003, 3, 5137.83);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (26, 2, 2003, 4, 3697.45);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (26, 2, 2003, 5, 2314.56);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (26, 2, 2003, 6, 5264.64);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt) VALUES
                (26, 2, 2003, 7, 3364.84);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (26, 2, 2003, 8, 4334.82);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (26, 2, 2003, 9, 4654.57);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (26, 2, 2003, 10, 3454.19);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (26, 2, 2003, 11, 4429.19);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (26, 2, 2003, 12, 4334.84);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (26, 3, 2003, 1, 2434.84);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (26, 3, 2003, 2, 2544.65);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (26, 3, 2003, 3, 5337.83);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (26, 3, 2003, 4, 5457.45);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (26, 3, 2003, 5, 4314.56);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (26, 3, 2003, 6, 3364.64);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (26, 3, 2003, 7, 3264.84);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (26, 3, 2003, 8, 4734.82);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (26, 3, 2003, 9, 2354.57);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (26, 3, 2003, 10, 4354.34);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (26, 3, 2003, 11, 2432.73);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (26, 3, 2003, 12, 4534.84);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (26, 4, 2003, 1, 3164.23);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (26, 4, 2003, 2, 3144.65);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (26, 4, 2003, 3, 6337.83);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (26, 4, 2003, 4, 2357.45);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (26, 4, 2003, 5, 4314.564);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
```

```
        VALUES (26, 4, 2003, 6, 4364.64);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (26, 4, 2003, 7, 2264.84);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (26, 4, 2003, 8, 4234.82);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (26, 4, 2003, 9, 2454.57);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (26, 4, 2003, 10, 1554.19);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (26, 4, 2003, 11, 2429.73);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (26, 4, 2003, 12, 3334.85);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (21, 5, 2003, 1, NULL);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (21, 5, 2003, 2, NULL);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (21, 5, 2003, 3, NULL);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (21, 5, 2003, 4, NULL);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (21, 5, 2003, 5, NULL);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (21, 5, 2003, 6, NULL);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (21, 5, 2003, 7, NULL);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (21, 5, 2003, 8, NULL);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (21, 5, 2003, 9, NULL);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (21, 5, 2003, 10, NULL);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (21, 5, 2003, 11, NULL);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (21, 5, 2003, 12, NULL);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (22, 5, 2003, 1, NULL);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (22, 5, 2003, 2, NULL);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (22, 5, 2003, 3, NULL);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (22, 5, 2003, 4, NULL);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (22, 5, 2003, 5, NULL);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (22, 5, 2003, 6, NULL);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (22, 5, 2003, 7, NULL);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (22, 5, 2003, 8, NULL);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (22, 5, 2003, 9, NULL);
```

```
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (22, 5, 2003, 10, NULL);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (22, 5, 2003, 11, NULL);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (22, 5, 2003, 12, NULL);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (23, 5, 2003, 1, NULL);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (23, 5, 2003, 2, NULL);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (23, 5, 2003, 3, NULL);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (23, 5, 2003, 4, NULL);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (23, 5, 2003, 5, NULL);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (23, 5, 2003, 6, NULL);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (23, 5, 2003, 7, NULL);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (23, 5, 2003, 8, NULL);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (23, 5, 2003, 9, NULL);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (23, 5, 2003, 10, NULL);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (23, 5, 2003, 11, NULL);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (23, 5, 2003, 12, NULL);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (24, 5, 2003, 1, NULL);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (24, 5, 2003, 2, NULL);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (24, 5, 2003, 3, NULL);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (24, 5, 2003, 4, NULL);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (24, 5, 2003, 5, NULL);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (24, 5, 2003, 6, NULL);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (24, 5, 2003, 7, NULL);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (24, 5, 2003, 8, NULL);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (24, 5, 2003, 9, NULL);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (24, 5, 2003, 10, NULL);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (24, 5, 2003, 11, NULL);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (24, 5, 2003, 12, NULL);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
```

```
        VALUES (25, 5, 2003, 1, NULL);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (25, 5, 2003, 2, NULL);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (25, 5, 2003, 3, NULL);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (25, 5, 2003, 4, NULL);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (25, 5, 2003, 5, NULL);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (25, 5, 2003, 6, NULL);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (25, 5, 2003, 7, NULL);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (25, 5, 2003, 8, NULL);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (25, 5, 2003, 9, NULL);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (25, 5, 2003, 10, NULL);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (25, 5, 2003, 11, NULL);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (25, 5, 2003, 12, NULL);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (26, 5, 2003, 1, NULL);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (26, 5, 2003, 2, NULL);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (26, 5, 2003, 3, NULL);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (26, 5, 2003, 4, NULL);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (26, 5, 2003, 5, NULL);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (26, 5, 2003, 6, NULL);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (26, 5, 2003, 7, NULL);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (26, 5, 2003, 8, NULL);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (26, 5, 2003, 9, NULL);

SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (26, 5, 2003, 10, NULL);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (26, 5, 2003, 11, NULL);
SQL> INSERT INTO AllSales (EmpID, ItemTypeID, Year, Month, SaleAmt)
        VALUES (26, 5, 2003, 12, NULL);
SQL> COMMIT;
```

Illustrative Examples on The Above Database

Ranking Functions

- The Different types of Ranking Functins are

1. **RANK() Function**
   - It returns the rank of items in a group.
   - RANK() leaves a gap in the sequence of rankings in the event of a tie.
2. **DENSE_RANK() Function**
   - It returns the rank of items in a group.
   - DENSE_RANK() doesn't leave a gap in the sequence of rankings in the vent of tie.

```
SQL> SELECT ItemTypeID, SUM(SaleAmt),RANK() OVER(ORDER BY SUM(SaleAmt) DESC)
        Rank, DENSE_RANK() OVER(ORDER BY SUM(SaleAmt) DESC) Dense_Rank
           FROM AllSales WHERE Year = 2003 AND SaleAmt IS NOT NULL
              GROUP BY ItemTypeID ORDER BY ItemTypeID;
```

PARTITION BY Clause

- The Clause is used to divide groups into subgroups.

```
SQL> SELECT ItemTypeID, Month, SUM(SaleAmt), RANK() OVER(PARTITION BY Month
     ORDER BY SUM(SaleAmt) DESC) Rank  FROM AllSales WHERE Year = 2003
        AND SaleAmt IS NOT NULL  GROUP BY ItemTypeID, Month
                                    ORDER BY ItemTypeID, Month;
```

APPLYING ROLLUP, CUBE AND GROUPING SETS

```
SQL> SELECT ItemTypeID, SUM(SaleAmt), RANK() OVER(ORDER BY SUM(SaleAmt) DESC)
        Rank FROM AllSales WHERE Year = 2003 GROUP BY ROLLUP(ItemTypeID)
                                    ORDER BY ItemTypeID;
SQL> SELECT ItemTypeID, EmpID, SUM(SaleAmt), RANK()OVER(ORDER BY SUM(SaleAmt)
        DESC) Rank  FROM AllSales  WHERE Year = 2003  GROUP BY
           CUBE(ItemTypeID, EmpID)  ORDER BY ItemTypeID, EmpID;

SQL> SELECT ItemTypeID, EmpID, SUM(SaleAmt), RANK()OVER(ORDER BY SUM(SaleAmt)
        DESC) Rank FROM AllSales  WHERE Year = 2003  GROUP BY GROUPING
           SETS(ItemTypeID, EmpID)  ORDER BY ItemTypeID, EmpID;

SQL> SELECT ItemTypeID, SUM(SaleAmt), RANK()  OVER(ORDER BY SUM(SaleAmt) DESC)
        Rank , DENSE_RANK() OVER(ORDER BY SUM(SaleAmt) DESC) DENSE_RANK
           FROM AllSales  WHERE Year = 2003  GROUP BY ItemTypeID
                                    ORDER BY ItemTypeID;
```

NULLS FIRST AND NULLS LAST CLAUSE

```
SQL> SELECT ItemTypeID, SUM(SaleAmt),  RANK()  OVER(ORDER BY SUM(SaleAmt) DESC
        NULL LAST ) Rank, DENSE_RANK()  OVER(ORDER BY SUM(SaleAmt) DESC
           NULLS LAST) Dense_Rank FROM AllSales  WHERE Year = 2003
              GROUP BY ItemTypeID  ORDER BY ItemTypeID;
```

CUME_DIST() Function

- It returns the position of a specified value relative to a group of values.
- The function represents the Cumulative distribution of the data.

PERCENT_RANK() Function

- It returns the Percent Rank of a Value relative to a group by Values.

```
SQL> SELECT ItemTypeID, SUM(SaleAmt), CUME_DIST() OVER(ORDER BY SUM(SaleAmt)
        DESC) Cumulative, PERCENT_RANK()  OVER(ORDER BY SUM(SaleAmt)
           DESC) Percent FROM AllSales  WHERE Year = 2003
              GROUP BY ItemTypeID  ORDER BY ItemTypeID;
```

NTILE() Function

- The NTILE (Buckets) is used to calculate n–tiles.
- Bucket specifies the number of buckets into which groups of rows are placed.

```
SQL> SELECT ItemTypeID, SUM(SaleAmt), NTILE(4) OVER(ORDER BY Sum(SaleAmt)
     DESC ) AS Ntile FROM AllSales WHERE Year = 2003 AND SaleAmt IS NOT NULL
          GROUP BY ItemTypeID ORDER BY ItemTypeID;
```

ROW_NUMBER() Function

- The Function is used to return a number with each row in a group.
- The row number starts at 1.

```
SQL> SELECT ItemTypeID, SUM(SaleAmt), ROW_NUMBER() OVER(ORDER BY
          SUM(SaleAmt) DESC) Row_Number FROM AllSales WHERE Year = 2003
               GROUP BY ItemTypeID ORDER BY ItemTypeID;
```

INVERSE PERCENTAGE Function

- These functions are used to get the value corresponding to a percentile.
- The types of INVERSE percentile functions are
  - PERCENTILE_DIST()
    - It examines the cumulative distribution values in each group until it finds one that is greater than or equal to x.
  - PERCENTILE_CONT()
    - It examines the percent rank values in each group until it finds one that is greater than or Equal to x.

```
SQL> SELECT PERCENTILE_CONT(0.6) WITHIN GROUP(ORDER BY SUM(SaleAmt)
          DESC) Percentile_Cont ,PERCENTILE_DIST(0.6) WITHIN GROUP(ORDER BY
                    SUM(SaleAmt) DESC) Percentile_Dist FROM AllSales
                         WHERE Year = 2003 GROUP BY ItemTypeID;
```

WINDOW FUNCTIONS

- The WINDOW Functions are used to calculate Cumulative Sums and moving averages within a specified range of rows.

Calculating Cumulative Sum

```
SQL> SELECT Month, SUM(SaleAmt) MonthlyAmt ,SUM(SUM(SaleAmt)) OVER
       (ORDER BY Month ROWS BETWEEN UNBOUNDED PERCEDING AND
            CURRENT ROW) CumulativeAmount FROM AllSales WHERE Year = 2003
                 GROUP BY Month ORDER BY Month;
```

UNBOUNDED PRECEDING

- Specifies that the window starts at the first row of the partition.

UNBOUNDED FOLLOWING

- Specifies that the window starts at the last row of the partition.

CURRENT ROW

- Specifies that the window begins at the Current Row or Value.

```
SQL> SELECT Month, SUM(SaleAmt) MonthlyAmt ,SUM(SUM(SaleAmt))
       OVER(ORDER BY Month ROWS UNBOUNDED PERCEDING)
          CumulativeAmount FROM AllSales WHERE Year = 2003
             AND Month BETWEEN 6 AND 12 GROUP BY Month ORDER BY Month ;
```

Calculating Moving Averages

```
SQL> SELECT Month, SUM(SaleAmt) MonthlyAmt,AVG(SUM(SaleAmt))
          OVER(ORDER BY Month ROWS BETWEEN 3 AND CURRENT ROW)
               MovingAverage FROM AllSales WHERE Year = 2003
                    GROUP BY Month ORDER BY Month ;
```

Calculating Centered Average

```
SQL> SELECT Month, SUM(SaleAmt) MonthAmt, AVG(SUM(SaleAmt))
       OVER(ORDER BY Month ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING)
         MovAvg FROM AllSales WHERE Year = 2003 GROUP BY Month ORDER BY Month;
```

- Displaying First And Last Rows Using First_Value() and Last_Value()

```
SQL> SELECT Month, SUM(SaleAmt) MonthAmt, FIRST_VALUE(SUM(SaleAmt))
       OVER(ORDER BY Month ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING)
         PRVMonthAmt, LAST_VALUE(SUM(SaleAmt) OVER(ORDER BY Month
           ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING) NEXTMonthAmt
             FROM AllSales WHERE Year = 2003 GROUP BY Month ORDER BY Month;
```

- Working With REPORTING Functions
- The Reporting function can be used to perform calculations across groups and partitions within groups.

```
SQL> SELECT Month, ItemTypeID, SUM(SUM(SaleAmt)) OVER(PARTITION BY Month)
       Tot_Month_Amt, SUM(SUM(SaleAmt)) OVER(PARTITION BY ItemTypeID)
         Tot_Pesd_Type_Amt FROM AllSales WHERE Year = 2003 AND Month < = 3
           GROUP BY Month, ItemTypeID ORDER BY Month, ItemTypeID;
```

RATIO_TO_REPORT() Function

- The function is used to compute the ratio of a value to the sum of a set of values.

```
SQL> SELECT Month, ItemTypeID, SUM(SaleAmt) ItemTypeAmt,
       RATIO_TO_REPORT(SUM(SaleAmt)) OVER(PARTITION BY Month) ProTypeRatio
         FROM AllSales WHERE Year = 2003 AND Month < = 3 GROUP BY Month,
           ItemTypeID ORDER BY Month, ItemTypeID;
```

LAG() and LEAD() Functions

- They Are used to get a value in a row, where that row is a certain number of rows away from the current row.

```
SQL> SELECT Month , SUM(SaleAmt) MonthAmt ,AVG(SUM(SaleAmt), 1) OVER(ORDER BY
       Month) PrevMthAmt , LEAD(SUM(SaleAmt), 1) OVER(ORDER BY Month)
         NextMthAmt FROM AllSales WHERE Year = 2003 AND Month < = 3
           GROUP BY Month ORDER BY Month;

SQL> SELECT MIN(Month) KEEP(DENSE_RANK FIRST ORDER BY SUM(SaleAmt))
       LowestSaleMonth FROM AllSales WHERE Year = 2003 GROUP BY Month
                                                   ORDER BY Month;
```

Data Updating And Deletion

Updating The Data in A Table

- The UPDATE Statement is Used To Change The Existing Values in A Table OR in The Base Table of View.
- It Can Be Used To UPDATE The Master Table of Materialized View.

Prerequisites

- The Table Must Be in The Own Schema.
- UPDATE Object Privilege Should Be Available.

Syntax:

```
SQL> UPDATE <Table_Name> SET <Specification> WHERE Clause;
SQL> UPDATE EMP SET Comm = NULL WHERE Job = 'CLERK';

SQL> UPDATE Emp SET  (Job, Deptno) = (SELECT Job, Deptno FROM Emp
                                      WHERE Empno = 7499) WHERE Empno = 7698;
SQL> UPDATE Emp SET Deptno = (SELECT Deptno  FROM Emp WHERE Empno = 7788)
                             WHERE Job = (SELECT Job  FROM Emp  WHERE Empno = 7788);
```

Applying Default Values

- It is Used To UPDATE A Value in A Column With DEFAULT Value Set in The Constraints.
- DEFAULT Key Word is Introduced From Oracle 9i Onwards.

```
SQL> UPDATE Emp SET Sal = DEFAULT  WHERE Ename = 'SMITH';
SQL> UPDATE Emp SET HireDate = DEFAULT  WHERE Ename = 'TAYLOR';
```

Returning Clause

- The Returning Clause is Introduced From 10g.
- It is Used To Return A Value From A Aggregate Function.
- The Clause Can Be Specified For Tables And Materialized Views And For Views With A Single Base Table.

Restrictions:

- Each Expression Must Be A Simple Expression OR A Single Set Aggregate Function.
- It Cannot Be Specified For Multi Table Insert.
- It Cannot Be Used With Parallel DML OR With Remote Objects.
- It Cannot Be Used To Retrieve Long Types.
- It Cannot Be Specified Upon A View Upon Which A Instead of Trigger is Defined.

```
SQL> VARIABLE SumSal NUMBER
SQL> UPDATE Emp  SET Sal = Sal * 1.1 WHERE Deptno = 10 RETURNING SUM(Sal)
                                                        INTO :SumSal;
SQL> PRINT SumSal;
```

MERGE Statement

- The MERGE Statement is Used To SELECT Rows From One OR More Sources For UPDATE OR INSERT Into One OR More Tables.
- The MERGE Statement is Convenient To Combine Multiple Operations And Avoid Multiple INSERT, UPDATE, DELETE.
- MERGE is A Deterministic Statement, Using Which The Same Row of The Target Table Can Be Transacted Multiple Times in The Same MERGE Statement.

Illustrations

```
SQL> CREATE TABLE MyBonus( Empno NUMBER, Bonus NUMBER DEFAULT 100 );
SQL> INSERT INTO MyBonus(Empno) (SELECT E.Empno FROM Emp E WHERE
                                                    Job = 'SALESMAN');
SQL> MERGE INTO MyBonus B USING (SELECT Empno, Sal, Deptno  FROM Emp
        WHERE Deptno = 30) S ON (B.Empno = S.Empno)
        WHEN MATCHED THEN UPDATE  SET B.Bonus = B.Bonus + S.Sal * 0.1
                                  DELETE  WHERE (S.Sal > 4000)
```

```
WHEN NOT MATCHED THEN INSERT(B.Empno, B.Bonus) VALUES(S.Empno,
                                S.Sal * 0.1) WHERE(S.Sal <= 4000;

SQL> CREATE TABLE ExamTimeTable ( ExamName VARCHAR2(30),
       ExamTime  VARCHAR2(12), CONSTRAINT ExamNamePK
            PRIMARY KEY(ExamName) );

SQL> INSERT INTO ExamTimeTable  VALUES ('PHYSICAL SCIENCES' , '9:00 AM');
SQL> MERGE INTO ExamTimeTable E1
      USING ExamTimeTable E2
       ON ( E2.ExamName = E1.ExamName AND E1.ExamName = 'PHYSICAL SCIENCES' )
       WHEN MATCHED THEN
           UPDATE  SET E1.ExamTime = '10:30 AM'
      WHEN NOT MATCHED THEN
       INSERT(E1.ExamName, E1.ExamTime) VALUES('PHYSICAL SCIENCES' , '10:30 AM');

SQL> MERGE INTO ExamTimeTable E1 USING ExamTimeTable E2
       ON (E2.ExamName = E1.ExamName AND E1.ExamName = 'CHEMICAL SCIENCES')
       WHEN MATCHED THEN
            UPDATE SET E1.ExamTime = '12:30 PM'
       WHEN NOT MATCHED THEN
            INSERT(E1.ExamName, E1.ExamTime ) VALUES('CHEMICAL SCIENCES' ,
                                                     '12:30 PM');
```

DELETE Statement

- It is Used To Remove Rows From
- An Un Partitioned OR Partitioned Table.
- The Un Partitioned OR Partitioned Base Table of A View.
- The Un Partitioned OR Partitioned Container Table of Writable Materialized View.

Prerequisites

- To DELETE Rows From A Table, The TABLE Must Be in The USERS.SCHEMA.
- To DELETE Rows From A Materialized View, DELETE OBJECT Privilege is A Must.
- DELETE ANY TABLE System Privilege Allows To DELETE Rows From Any TABLE OR PARTITION TABLE OR Form The Base Table of Any VIEW.

Syntax

```
SQL> DELETE [FROM] <Table_Name> [WHERE Condition];
```

Using RETURNING Clause

```
SQL> DELETE FROM EmpWHERE Empno = 7864;
SQL> DELETE FROM Emp WHERE Deptno = 20;
SQL> DELETE FROM Emp WHERE Deptno = (SELECT Deptno FROM Dept
                                     WHERE Dname = 'SALES');
SQL> VARIABLE Salary NUMBER;
SQL> DELETE FROM Emp WHERE Job = 'SALESMAN' AND HireDate < SYSDATE
                                  RETURNING SUM(Sal) INTO :Salary;
SQL> PRINT :Salary;
```

Transaction Control

- Oracle Server Ensures Data Consistency Based Upon Transactions.
- Transactions Consist of DML Statements That Make Up One Consistent Change To The Data.

Transaction Start And End Cases

- A Transaction Begins When The First Executable SQL Statement is Encountered.
- The Transaction Terminates When The Following Specifications Occur.
  - A COMMIT OR ROLLBACK is Issued.
  - A DDL Statement Issued.

- A DCL Statement Issued.
- The User Exists The SQL * Plus
- Failure of Machine OR System Crashes.

- A DDL Statement OR A DCL Statement is Automatically Committed And Hence Implicitly Ends A Transaction.

Explicit Transaction Control Statements

- The Logic of Transaction Can Be Controlled By Using

COMMIT

- It Ends The Current Transaction By Making All Pending Data Changes Permanent.

SAVEPOINT <Name>

- It Marks A SAVEPOINT Within The Current Transaction.

ROLLBACK [To Savepoint Name]

- It Ends The Current Transaction By Discarding All Pending Data Changes.

State of Data Before Commit OR Rollback

- Every Data Change Made During The Transaction is Temporary Until The Transaction is Committed.
- Data Manipulation Operations Primarily Do Not Affect The State of The Data, Hence The Can Be Recovered.
- The Current User Can Review The Results of The Data Manipulation Operation By Querying The Tables.
- Other Users Cannot View The Results of The Data Manipulation Operations Made By The Current User.
- The Oracle Server Institutes Read Consistency To Ensure That Each User Sees Data As It Existed At The Time of Last Commit.

State of The Data After Commit is Issued

- Data Changes Are Written To The Database Permanently.
- The Previous State of The Data in The Database is Permanently Lost.
- All Users Can View The Results of The Recent Transactional Change.
- The Locks on The Affected Rows Are Automatically Released.
- All SAVEPOINTS Are Erased.

SQL> COMMIT;

State OF The Data After Rollback

- Rollback Statement is Used To Discard All Pending Changes.
- The Data Changes Are Undone.
- The Previous State of The Data is Returned OR Restored.
- The LOCKS on The Affected Rows Are Released Automatically.

SQL> ROLLBACK;

Rolling Back Changes To A Savepoint

- SAVEPOINT is Used To Create A Marker in The Current Transaction.
- Using SAVEPOINT The Transaction Can Be Discarded Up To The Marker By Using The ROLLBACK Statement.

SQL> ROLLBACK TO <SAVEPOINTName>;

- If a Second SAVEPOINT is Created With The Same Name As An Earlier SAVEPOINT, The Earlier SAVEPOINT is Deleted.

Alterting The Table Definition

Syntax For Adding Column

SQL> ALTER TABLE <Table_name> ADD (ColumnName DataType [DEFAULT Exp],
ColumnName DataType]...);

Syntax For Modifying Column

SQL> ALTER TABLE <TableName> MODIFY (ColumnName DataType [DEFAULT Exp],
Column DataType]...);

Adding A Column To A Table

- The ADD Clause is Used To Add Columns For An Existing Table.

SQL> ALTER TABLE Dept30 ADD (Job VARCHAR2(9));

Guidelines For ADDING Column

- A Column Can Be ADDED OR MODIFIED But Cannot Be Dropped From A Table.
- We Cannot Specify The Location Where The Column Can Appear, It By Default Becomes The Last Column.
- If The Table Contains Records, Before The Column is Added, The New Column Contains NULL Values.

Modifying A Column

- A Column Data Type, Size And Default Value Can Be Changed.
- A Change To The Default Value Affects Only Subsequent Insertions To The Table.

Guidelines To Modify A Column

- We Can Increase The Width OR Precision of A Numeric Column.
- We Can Decrease The Width of A Column If The Column Contains Only NULL Values And If The Table Has No Rows.
- We Can Change The Data Type If The Column Contains NULL's.
- We Can Convert A CHAR Column To The VARCHAR2 Data Type OR Convert A VARCHAR2 Column To The CHAR Data Type If The Column Contains NULL Values OR If The Size is Not Changed.

Dropping A Column

- A Column Can Be Dropped From A Table By Using The ALTER TABLE Statement.
- The DROP COLUMN Clause is Used For This Purpose And The Feature is Enabled From Oracle 8i Onwards.

Guidelines To Drop A Column

- The Column May OR May Not Contain Data.
- Only One Column Can Be Dropped At A Time.
- The Table Must Have At Least One Column Remaining in It After It is Altered.
- Once A Column is Dropped It Cannot Be Recovered.

SQL> ALTER TABLE Dept30 DROP Column Job;

SET UNUSED Option

- The SET UNUSED OPTION Marks One OR More Columns As Unused Such That They Can Be Dropped When The Demand on System Resources is Less(8i).
- The Response Time is Faster Than The DROP Clause.
- Unused Columns Are Treated As if They Were Dropped, Even Trough Their Column Data Remains in The Tables Rows.
- After A Column Has Been Marked Unused, We Cannot Have Access To That Column.
- The Names And Types of Columns Marked Unused Will Not Be Displayed During A Describe.
- We Can ADD To A TABLE A New Column With The Same Name As An Unused Column.

SQL> ALTER TABLE Dept30 SET UNUSED(Ename);

DROP UNUSED Columns Option

- This Option Removes From The Table All Columns Currently Marked As Unused.
- The Option is Used When We Want To Reclaim The Extra Disk Space From Unused Columns in The Table.
- If The Table Does Not Contain UNUSED Columns The Statement Returns With No Errors.

SQL> ALTER TABLE Dept30 DROP UNUSED COLUMNS;

Dropping A Table

- It Removes The Definition of The ORACLE TABLE.

- The Command Not Only Drops The TABLE But The ENTIRE DATABASE is Lost Along With The ASSOCIATED INDEXES.

Syntax

SQL> DROP TABLE <TableName> [CASCADE CONSTRAINTS];
SQL> DROP TABLE Dept30 CASCADE CONSTRAINTS;

Guidelines To Drop Table

- The Data is Totally Deleted From The Table.
- Any VIEWS And SYNONYMS Will Remain But Are Kept in Invalid State.
- Any Pending Transactions Are Committed.
- Only The Creator OR Owner of The Table OR A USER With DROP ANY TABLE Privilege Can Remove A Table From Database.
- The DROP TABLE Statement Once Executed is Irreversible.

Changing The Name of An Object

- The RENAME Command Can Be Used To Change The Name of A
  - TABLE
  - VIEW
  - SEQUENCE
  - SYNONYM
- To RENAME The OBJECT We Must Be The OWNER of The OBJECT.

Syntax

SQL> RENAME <OldName> TO <NewName>;

Illustration

SQL> RENAME Dept TO Department;

Truncating A Table

- It Is Used To Remove All Rows From A TABLE And To Release The STORAGE SPACE Used By The Specific TABLE.
- The TRUNCATE TABLE Will Not Facilitate For ROLLBACK.

Syntax

SQL> TRUNCATE TABLE <TableName>;

Illustration

SQL> TRUNCATE TABLE Department;
To TRUNCATE A TABLE We Must Be The OWNER of The TABLE.

Applying Comments Upon A Table

- The COMMENTS Command is Used To ADD Comments To A TABLE OR A COLUMN OR VIEW Etc.
- Each Comment Can Be Up To 2000 Bytes.
- The Data Dictionary In Which Comments Are Stored Are...
  - ALL_COL_COMMENTS
  - USER_COL_COMMENTS
  - ALL_TAB_COMMENTS
  - USER_TAB_COMMENTS

Syntax

SQL> COMMENT ON TABLE <TableName>/ COLUMN <Tablename.Column> is 'Text';

Illustration

SQL> COMMENT ON TABLE Emp IS 'The Table Storing Employee Information';
SQL> COMMENT ON COLUMN Emp.MGR IS
'This Column is Actually Storing The Registered Employee Numbers As Manager Numbers, With A Self Relation';

Dropping A Comment

- A Comment is Dropped From The Database By Setting It To An Empty String.

SQL> COMMENT ON TABLE Emp IS '';

Advanced Table Creation Strategies

Creating A Table From An Existing Table

ON The FLY Tables

- Oracle Allows The Creation of A New Table On-The-Fly, Depending on A SELECT Statement on An Already Existing Table.

Syntax

SQL> CREATE TABLE <TableName> AS SELECT Columns FROM TableName;
[WHERE Condition];

- The CREATE TABLE... AS SELECT... Command Will Not Work If one of The Selected Columns Use LONG Data Type.
- When The New Table is Described It Reveals That It Has "INHERITED" The Column Definition From The Existing Table.
- Using This Style We Can Include All Columns Using Asterisk OR A Subset of Columns From Table.
- The New Table Can Contain "INVENTED COLUMNS" Which Are The Product of Function of The Combination of Other Columns.
- The Column Definition Will Adjust To The Size Necessary To Contain The Data in The INVENTED COLUMNS.

Creating An Exact Copy

SQL> CREATE TABLE SampDept AS SELECT * FROM Dept;

Creating An Exact Copy With Different Column Names

SQL> CREATE TABLE SampDept1 (DeptID, DeptName, Place) AS SELECT * FROM Dept;

Creating A Copy With Required Columns

SQL> CREATE TABLE SampDept3 AS SELECT DeptNo, Dname FROM Dept;

Creating A Copy With Required Columns

SQL > CREATE TABLE SampDept3(DeptID, DeptName) AS SELECT DeptNo, Dname
FROM Dept;

Creating A Copy With Invented Columns

SQL> CREATE TABLE SampDept3 (DeptID, DeptName,DeptBudget) AS SELECT D.DeptNo,
D.Dname, SUM(E.Sal) FROM Emp E, Dept DWHERE E.Deptno = D.Deptno
GROUP BY D.Deptno, D.Dname;

Creating A Copy Without Data

SQL> CREATE TABLE Sampdept3 AS SELECT * FROM Dept WHERE 1 = 2;

Creating A TABLE Without Generating REDO LOG Entries

- REDO LOG Entries Are Chronological Records of Database Actions Used During Database Recoveries.
- The REDO LOG Entries Generation Can Be Avoided By Using The NOLOGGING Keyword.
- By Circumventing With NOLOGGING Key Word The Performance of The CREATE TABLE Command Will Improve As Less Work is Being Done.
- As The New Table Creation is Not Being Written To The REDO LOG Files ,The Table Will Not Be Able To Re-Create, Following A Database Failure.
- The REDO LOG Files Are Used To Recover The Database.

SQL> CREATE TABLE SampDept NOLOGGING AS SELECT * FROM Dept;

Creating Index Organized Table:

- An Index Organized Table Keeps Its Data Stored According To The PRIMARY Key Column Values in The Table.

- An Index Organized Table Stores Its Data As if The Entire Table Was Stored in An INDEX.
- To Create An Index Organized Table The ORGANIZATION INDEX Clause of The CREATE TABLE is Used.
- To Create A Table As An Index Organized Table We Must Create A PRIMARY KEY Constraint on It.

Illustration

```
SQL> CREATE TABLE Sample(SampID NUMBER(4), SampName VARCHAR2(20),
                SampDate DATE, CONSTRAINT SampIDSampNamePK
                        PRIMARY KEY(SampID, SampName)) ORGANIZATION INDEX;
```

- To Minimize The Amount of Active Management of The Index Organized Table, We Should Create it Only if The Table's Data is Very Static.
- An Index Organized Table is Most Effective When The Primary Key Constitutes A Large Part of The Tables Columns.

Working With Partitioned Tables

- Dividing The Rows of A Single Table Into Multiple Parts is Called Partitioning of A Table.
- The Table That is Partitioned is Called "PARTITIONED TABLE" And The Parts Are Called PARTITIONS.
- The Partitioning is Useful For Very Large Tables Only.

Important Goals Behind Partitioning

- The Performance Of Queries Against The Tables Can Improve Performance
- The Management Of The Table Becomes Easier
- As The Partitioned Table Data is Stored In Multiple Parts, It is Easier To Load And Delete Data in Partitions Than in The Large Table.
- The Backup And Recovery Operation Can Be Performed Better.

Normal Un Partitioned Table

```
SQL> CREATE TABLE SampleTable( SampleID NUMBER(4) CONSTRAINT SampIDPK
        PRIMARY KEY,SampName VARCHAR2(20),SampDate DATE, SampDesc LONG);
```

Creating Range Partition Table

```
SQL> CREATE TABLE SampleTablePart(SampID NUMBER(4) PRIMARY KEY ,
            SampName VARCHAR2(20), SampDate DATE, SampDesc VARCHAR2(4000))
            PARTITION BY RANGE(SampID)
            ( PARTITION SampIDPart1 VALUES LESS THAN(500),
              PARTITION SampIDPart2 VALUES LESS THAN(1000)
            PARTITION  SampIDPart3 VALUES LESS THAN(MAXVALUE));
```

- The Maximum Value Need Not Be Specified For The Last Partition, The MAXVALUE Keyword is Specified.
- The MAXVALUE Specifies Oracle To Use The Partition To Store Any Data That Could Not Be Stored In The Earlier Partitions.
- We Can Create Multiple Partitions Each With Its Own Upper Value Defined.
- The Minimum Value For The Range is Implicitly Determined By Oracle From The Definition of The Preceding Partition.

Hash Partitions Upon A Table

- A Hash Partition Upon A Table Determines The Physical Placement of Data.
- The Physical Placement of Data is Determined By Performed A Hash Function On The Values of The Partition Key.
- In Hash Partition Consecutive Values of The Partition Key Are Not Generally Stored in The Same Partition.
- Hash Partitioning Distributes A Set of Records Over A Greater Set of Partitions, Decreasing The Likelihood For I/O Contention.

- To Create A Hash Partition We Use The PARTITION BY HASH Clause.

```
SQL> CREATE TABLE Emptablehash( Empno NUMBER(6) CONSTRAINT Empnopk PRIMARY
        KEY, Ename VARCHAR2(30), Job VACHAR2(30), Deptno NUMBER(2)
          CHECK Deptno IN(10, 20, 30, 40, 50, 60, 70, 80, 90),Sal NUMBER(8, 2),
            CONSTRAINT Deptnofk_Hash FOREIGN KEY(Deptno)
              REFERENCES Dept(Deptno )) PARTITION BY
                HASH(Deptno) PARTITIONS 9;
```

Format Choices

- Specify The Number of Partitions And The Table Space To Use.
- Specify The Named Partitions.

```
SQL> CREATE TABLE Emptablehash (Empno NUMBER(6) CONSTRAINT Empnopk PRIMARY
          Key ..... )
     PARTITION BY HASH(Deptno)
     PARTITION 2 STRORE IN (Deptnopart1TS, Deptnopart2TS);
```

Working With List Partitioning

- In LIST PARTITIONING We Specify Oracle All The Possible Values And Designate The Partition Into Which The Corresponding Rows Should Be Inserted.

```
SQL> CREATE TABLE Empsamplelist (Emp  NUMBER(4) Constraint Empnopk PRIMARY KEY,
        Ename VARCHAR2(20),Sal NUMBER(8, 2), Hiredate Date, Deptno NUMBER(2),
          Job VARCHAR2(15),CONSTRAINT DeptnoFK FOREIGN KEY(Deptno)
            REFERENCES Dept(Deptno)) PARTITION BY LIST(Job)
          ( PARTITION Jpart1 VALUES ('PRESIDENT', 'ANALYST'),
            PARTITION Jpart2 VALUES('MANAGER', 'SALESMAN', 'CLERK'));
```

Generating Sub Partitions

- Sub Partitions Are Partitions of Partitions.
- Sub Partitions Can Be Used To Combine The Two Types of Partitions...
  - RANGE Partitions
  - HASH Partitions
- In Very Large Tables, The Composite Partition Strategy is An Effective Way of Separating The Data Into Manageable And Tunable Divisions.

```
SQL> CREATE TABLE Empsamplesubpart(Empno NUMBER(6) PRIMARY KEY,
        Ename VARCHAR2(30),Sal NUMBER(8, 2),Deptno NUMBER(2),
          Job VARCHAR2(30), CONSTRAINT Deptnofk FOREIGN KEY(Deptno)
                                    REFERENCES Dept(Deptno) )
     PARTITION BY RANGE(Ename)
     SUBPARTITION BY HASH(JOB)
     SUBPARTITIONS 5(
          PARTITION Namep1
          VALUES LESS THAN('M'),
          PARTITION Namep2
               VALUES LESS THAN(MAXVALUE) );
```

Splitting Table Partitions

```
SQL> ALTER TABLE SampleTablePart SPLIT PARTITION Sampidpart3 AT (2000)
          INTO ( PARTITION Sampidpart3, PARTITION Sampidpart4);
SQL> ALTER TABLE Empsamplelist SPLIT PARTITION Jpart2 VALUES('SALESMAN')
        INTO ( PARTITION JSalesMan, PARTITION SSalesMan);
```

Merging Table Partitions

```
SQL> ALTER TABLE Empsamplelist MERGE PARTITIONS Jpart1, Jsalesman
                                    INTO PARTITION Jpart1;
SQL> ALTER TABLE Sampletablepart MERGE PARTITIONS Sampidpart2, Samplepart3
                                    INTO PARTITION Sampidpart3;
```

Dropping A Table Partition

SQL> ALTER TABLE Sampletablepart DROP PARTITION Sampleidpart3;
SQL> ALTER TABLE Empsamplelist DROP PARTITION Jpart1;

Creating Indexes upon Partitions

- Once A Partitioned Table is Created, We Have To Create An Index Upon That Table.
- The Index May Be Partitioned According To The Same Range of Values As That Were Used To Partition The Table.
- The Indexed Partitions Can Be Placed Into Specific Table Spaces.

```
SQL> CREATE INDEX Empsamplelistinx ON Empsamplelist(Job) LOCAL(
                    PARTITION Jpart1, Jpart2 );
```

- The LOCAL Keyword Tells Oracle To Create A Separate Index For Each Partition on The Table.
- The GLOBAL Keyword Tells Oracle To Create A Non Partitioned Index.
- LOCAL Indexes Are Easier To Manage Than GLOBAL Indexes.
- GLOBAL Indexes Can Perform Uniqueness Checks Faster Than LOCAL Indexes.

Object Oriented Concepts in Oracle

Object Tables

- Object Tables Are Created By Using The User Defined Data Types.
- In An Object Table Each Row OR Record is Treated As An Object.
- Each Row in An Object Table Has An Object Identifier(OID), Which is Unique Through Out The Database.
- The OID is Generated And Assigned By Oracle When The Row OR Object is Created.
- The Rows OR Objects of An Object Table Can Be Referenced By Other Objects With in The Database.
- An Object Table is Created Using The CREATE TABLE Command.
- All Object Tables Automatically Inherit The Data Types From The User Defined Data Types.
- All Object Types Are Associated With Default Methods Applied Upon The Relational Tables i.e. INSERT, DELETE, UPDATE And SELECT.
- The Relational DML Operation Style is Accepted Only When The User Defined Data Type is A Collection of Built-in Data Types, And The Object Table Does Not Contain Any REF Constraints.

Creating An User Defined Object Type

Syntax

```
SQL> CREATE OR REPLACE TYPE <TypeName> AS OBJECT ( ColumnName1  DataType(Size),
        Column Name2 DataType(Size),ColumnNameN DataType(Size));
```

- All User Defined Data Types Are Schema Objects of The Database.
- The User Defined Object Data Type Can Be Used As Reference in Other Tables, OR Instantiated As Object Table Directly.
- All User Defined Data Types And Objects Are Stored Permanently in The Data Dictionaries
  - USER_TYPES
  - USER_OBJECTS
- We Can Query For The User Defined Data Types And Objects Using The Relational SELECT.

```
SQL> SELECT TYPE_NAME, TYPECODE,ATTRIBUTES, METHODS FROM USER_TYPES;
SQL> SELECT OBJECT_NAME, OBJECT_TYPE FROM USER_OBJECTS;
```

Creating User Defined Student Type

```
SQL> CREATE OR REPLACE TYPE Student AS OBJECT (Studid NUMBER(6),
        Sname VARCHAR2(20),DOB DATE ,DOA DATE,FEES NUMBER(7, 2));
```

- The Above Statement Creates The User Defined Object Data Type Called As Student And Stores it in The Data Dictionary Called USER_TYPES.
- This Data Type is Also Called As Collection in Oracle, And This Collection is Reusable Where Ever The Same Data Type Collection is Expected in Project Development.

Creating An object Table

Syntax

```
SQL> CREATE TABLE TableName OF  TypeName;
```

Illustration

```
SQL> CREATE TABLE McaStudent OF Student;
```

- The Above Statement Creates The Object Table McaStudent As An Abstract Data Type.
- Each Row in The Object Table Has An OID Value Generated By Oracle Server.
- The Rows in The Object Table Are Referenced As OBJECTS.

Inserting Rows Into OBJECT TABLES

- To INSERT A Record into An OBJECT TABLE We May Use The CONSTRUCTOR METHOD of The Actual Data Type OR Directly Implement The RELATIONAL INSERT Statement.
- The Normal INSERT OR RELATIONAL INSERT is Possible Only When The Table Does Not Contain Any Nested Data Types.

Relational INSERT

SQL> INSERT INTO McaStudent VALUES(1234, 'Kumar', '07-Oct-98', SYSDATE, 15000);

INSERT Records into Specific Columns

SQL> INSERT INTO McaStudent(StudID, SName)VALUES(1234, 'Krishna');

INSERT Using CONSTRUCTOR Method

SQL> INSERT INTO McaStudent VALUES(Student(1234, 'Satish','05-SEP-99',SYSDATE, 1300));

UPDATE Data From OBJECT TABLE

SQL> UPDATE McaStudent SET Sname = 'Sri Ram' WHERE StudID = 1234;

DELETE Data From OBJECT TABLE

SQL> DELETE FROM McaStudent WHERE StudID = 1234;
SQL> DELETE FROM McaStudent;

SELECT Data From OBJECT TABLE

- The Abstract Data Types Column Can Be Referred As A Part of The Table's Columns.

SQL> SELECT * FROM McaStudent;
SQL> SELECT StudID, Sname FROM McaStudent;
SQL> SELECT StudID, Sname FROM McaStudentWHERE StudID = 1234

Creating Table With User Defined Data Type

SQL> CREATE TABLE MBAStudent(StudentColl STUDENT,Semester VARCHAR2(10),
SemStartDate DATE,SemEndDate DATE, Specialization VARCHAR2(20));

- Once The User Defined Data Types Are Created We Can Instantiate Them in The Normal Relational Tables.
- These Instances Look As Normal Attributes With in The Table, But Can Be Operated Only With CONSTRUCTOR METHOD OR OBJECT VIEWS.
- In Any of The Operation We Have To Provide Reference To All Attributes With in The Instance, But Partial Association is Not Accepted.

SQL> INSERT INTO MBAStudentVALUES(STUDENT(1234, 'SAMPATH', '10-DEC-85',
SYSDATE, 25000), 'FIRST', '25-JUL-07', '25-OCT-07', 'MARKETING');

The REF() Function

- The REF() Function Allows To Reference Existing Row Objects.
- The OID Assigned To Each Row Can Be Seen By Using The REF Function

SQL> SELECT REF(A) FROM McaStudent A WHERE SName = 'SATISH';

- REF() Always Returns The OID of The Registered Objects in The Object Table.
- REF() Always Expects The Alias of The Object Table As Argument.
- The REF(A) Value Will Be Different Under Different Systems And May Be Wrapped Onto Multiple Lines.
- The REF() Function Can Only Reference Row Objects ,Hence We Cannot Use REF() For Referencing Column Objects.
- The Column Objects Can Be of
  - Abstract Data Types.
  - LOB's.
  - Collections.
- The REF() Function By Itself Does Not Give Any Useful Information.

DEREF() Function

- The DEREF() Function Takes A Reference Value i.e, The OID Generated For A Reference And Returns The Original Value of The Row Object.

```
SQL> CREATE TABLE StudentIncharges(InchargeName  VARCHAR2(30),
                        StudentIncharge REF STUDENT );
```

- The StudentIncharge Column References The Data That is Stored Elsewhere in The System.
- The REF() Function Points The StundentIncharge Column To A ROW OBJECT of The STUDENT Data Type.
- As McaStudent is An OBJECT TABLE of The STUDENT Data Type, The StudentIncharge Column Can Point To
- The ROW OBJECT Within The McaStudent OBJECT TABLE.
- The StudentIncharge Table Can Be Described As An Ordinary Table.

```
SQL> DESC StudentIncharge;
```

- Seeing Full Details of The Reference Column

```
SQL> SET DESCRIBE DEPTH 2;
SQL> DESC StudentIncharge;
```

INSERT Records Into REFERENCE TABLE

- To Insert Records Into Reference Tables We Have To Use The REF() Function.
- The REF Column Always Contains Only OID's

```
SQL> INSERT INTO StudentIncharges SELECT 'SUBRAMANYAM SHARMA',
                REF(A) FROM McaStudent A  WHERE SName = 'SATISH KUMAR';
```

In The Above Case

- The McaStudent Table is Queried First.
- The REF() Function Returns The OID For The ROW OBJECT Selected.
- The Selected OID is Stored in The StudentIncharge Table As A Pointer To That ROW OBJECT in The McaStudent OBJECT TABLE.
- The StudentIncharges Table Actually Contains The Name of The StudentIncharge And A REFERENCE To A ROW OBJECT in The McaStudent Table.
- The REFERENCE OID Can Be Seen By Querying Upon The StudentIncharge Table.

```
SQL> SELECT * FROM StudentIncharges;
```

- The Reference Value Cannot Be Seen Until We Use The DEREF() Function.
- The Parameter For The DEREF() Function is The Column Name of The REF Column But Not The Table Name.

```
SQL> SELECT  DEREF(X.StudentIncharge)  FROM StudentIncharges X WHERE
                                        InchargeName = 'SUBRAMANYAM SHARMA';
```

Points To Note

- The Query Uses A Reference To A Row Object To Travel From One Table To The Second.
- A Join is Performed in The Background Without Specifying The Join Criteria.
- The Object Table It Self is Not Mentioned in The Query.
- The Name of The Object Table Need Not Be Known To DEREF The Values.
- The Entire Referenced Object is Returned Not Just Part of The Row.

Implementing VALUE() Function

- The VALUE() Function is Useful When Debugging REFERENCES.
- The VALUE() Function Allows To Query The Formatted Values Directly From The Object Table.
- We Can Select The Values From Object Table Without Using The DEREF Query Upon The StudentIncharges, StudentIncharge Column.

```
SQL> SELECT VALUE(A) FROM MCAStudent A   WHERE SName = 'SESHU';
```

Invalid References

- We Can DELETE The Object To Which A REFERENCE Points.

- We Can DELETE A Row From MCAStudent Object Table To Which StudentIncharge Record Points.

SQL> DELETE FROM MCAStudent WHERE SName = 'SESHU';

DANGLING REF

- It is A Record Which Has An OID Pointing To A Record in An Object Table, For Which The Reference Record is Not Existing in The Original Object Table.
- As Oracle Generates A OID For The Row Object Which Can Be Referenced By Any Other Row From Other Object Table When The Row Object is DELETED The OID is Lost And The Oracle Doesn't Reuse The OID Numbers.
- Hence If A New Record For 'SESHU' is Inserted This is Given An OID Value, But The StudentIncharge Record Still Points To The OLD Value Only.
- In A Relational System The Join Between Two Tables is Dependent Only On The Current Data.
- In An OOP System The Join is Between Objects, Hence The Fact That Two Objects Having The Same Data Does Not Mean They Are Same.

OBJECT VIEWS With REF's

- OBJECT VIEWS Are Very Important To Super Impose OOPS Structures on An Exiting Relational Table.
- We Can Create Abstract Data Type And Use Them Within The Object View of An Existing Table.
- OBJECT VIEW Acts As A Bridge Between The Existing Relational Application To Object Relational Applications.

Integrating User Defined Data Types To Relational Tables
Relational Stage

```
SQL> Create Table Students(Studid NUMBER(6) CONSTRAINT StudIDPK PRIMARY KEY,
        SName VARCHAR2(30),Street VARCHAR2(40),CityName VARCHAR2(25),
            StateName VARCHAR2(40),Pincode NUMBER(6));
```

Creating Abstract Data Types

```
SQL> CREATE OR REPLACE TYPE MyAddress AS OBJECT( Street VARCHAR2(40),
        CityName VARCHAR2(25),StateName VARCHAR2(40),Pincode NUMBER(6));

SQL> CREATE OR REPLACE TYPE MyStudent AS OBJECT(SName VARCHAR2(30),
                                    Saddress Myaddress);
```

- As Student Table Was Created Without Using The MyAddress And MyStudent Data Types It Has To Be Accessed Via OBJECT VIEWS.
- The Above Principle of Accessing Data is Called As Object Based Access.
- An OBJECT VIEW Can Be Specified Upon The Abstract Data Types To Apply Onto The Relational Table.

Creating An Object View

```
SQL> CREATE OR REPLACE VIEW StudentOV(StudID, StudDEF) AS SELECT StudID,
        MyStudent(SName,Myaddress(Street, CityName , StateName, Pincode)) FROM Students;
```

- We Can Access The Students Table Directly As A Relational Table OR Via The CONSTRUCTOR Methods of The Abstract Data Types.

Relational SELECT on "Students" Table

```
SQL> SELECT * FROM Students;
```

Selecting Data From The Relational Table Using The OBJECT VIEW

```
SQL> SELECT * FROM StudentOV;
```

Object Views With References

- If The Students Table is Related To Another Table, Then The Object Views Can Be Created As A Reference Between These Tables.
- By Using The Above Concept ORACLE Uses The Existing PRIMARY KEY OR FOREIGN KEY Relationships To Simulate OID's For Use By REF's Between The Tables.
- By Implementing The Above Concept We Can Access The Table Either As Relational Table OR As OBJECTS.
- When The Tables Are Treated As OBJECTS We Can Use The REF's, To Automatically Perform JOINS Upon The Tables Using The DEREF() Function.

Create A Reference Table For Students

```
SQL> CREATE TABLE StudentBooks( LibTranNO NUMBER(6),StudID NUMBER(6),
        BookTitle VARCHAR2(50),LendingDate DATE,CONSTRAINT LibtreamNoPK
          PRIMARY KEY(LibTranNO, Studid),CONSTRAINT StudentBooksFK
            FOREIGN KEY(Studid) REFERENCES Students(StudID));
```

Inserting The Data Into Library Table

```
SQL> INSERT INTO StudentBooks VALUES(2000, 1200, 'THERMO DYNAMICS', SYSDATE);
SQL> SELECT *FROM StudentBooks;
SQL> SELECT Sname, BookTitle, LendingDateFROM Students S, StudentBooks SB
                                        WHERE S.StudID = SB.StudID;
```

GENERATING OID's

- We Can Use An Object View To Assign OID's To The Records in Students Table.
- OID's Are Assigned To Records in an OBJECT TABLE And An OBJECT TABLE in Turn is Based on An Abstract Data Type.
- As First Step Create An ABSTRACT DATA TYPE That Has The Same Structure As The Student Table i.e. The Relational Table.

```
SQL> CREATE OR REPLACE TYPE StudentType AS OBJECT(Studid NUMBER(6),
        Sname VARCHAR2(30), Street Varchar2(40),Cityname VARCHAR2(25),
          Statename VARCHAR2(40),Pincode.NUMBER(6));
```

- As The Next Step Create A View Based Upon The StudentType By Assigning OID Values To The Records in Student Table.

```
SQL> CREATE OR REPLACE VIEW STUDENTOV OF StudentType WITH OBJECT
        IDENTIFIER(StudID) AS SELECT StudID, Sname, Street, Cityname, Statename,
                                        Pincode FROM Students;
```

- The First Part of The Statement Tells The Database To CREATE A VIEW Based on The Structure Defined in Student Type.
- The Next Part of The CREATE VIEW Tells The Database How To Construct OID Values For The Rows in Students Table.
- With The Above Step The Rows of Students Are Now Accessible As ROW OBJECTS Via The StudentOV View.
- The OID Values Generated For The StudentOV Rows Are Called As PKOID's As They Are Based Upon Students Primary Key Values.
- The Relational Tables Can Be Accessed As Row OBJECTS If The Object Views Are Created Upon Them.

Generating References

- Actually The Rows of StudentBooks Reference Rows in Students Table.
- As Per Relational Concept The Relationship is Determined By The FOREIGN KEY Pointing From The StudentBooks.StudID Column To The Student.StudID Column.
- From The StudentOV Object View That Has Been Created, The Rows in Students Table Can Be Accessed Via OID's.
- We Have To CREATE REFERENCE VALUES in StudentBooks That Reference Students Table.

- Once The REF's Are Created We Can Use DEREF() Function To Access The Students Data From Student Books.

```
SQL> CREATE VIEW StudentBooksOV AS SELECT MAKE_REF(StudentOV, StudID) StudID,
                    LibTranNo, BookTitle, LendingDate FROM StudentBooks;
```

MAKE_REF() Function

- It Creates REFERENCES Which Are Called PKREF's From An Existing View To Another View.
- PKREF's Are Named So As They Are Based on PRIMARY KEYS.
- The Function Takes As An Argument(s) As The Name of The OBJECT VIEW Being REFERENCED And The Name of The Column OR Columns That Form The FOREIGN KEY in The Local Table.
- Since MAKE_REF() Creates A View The Result of An Operation Must Be Given A COLUMN ALIAS.

Querying Through OBJECT VIEWS

- We Use The DEREF() Function To SELECT The Value of The Referenced Data.
- The Concept is Almost All Same As That of OBJECT VIEWS.

```
SQL> SELECT DEREF(SB.StudID) FROM StudentBooksOV SB
                    WHERE BookTitle = 'THERMO DYNAMICS';
```

Steps Performed

- The Query Finds The Records in The StudentBooks Table For Which The LendingDate is The Current System Date.
- Taking The StudID Value From That Record Evaluates Its Reference.
- The Evaluated Reference StudID is Pointed To The PKOID Value in The StudentOV OBJECT VIEW Using MAKE_REF.
- The StudentOV Object View Returns The Record Whose PKOID Matched The Referenced Value.
- The DEREF() Function Then Gets Activated Returning The Value of The Referenced Row.
- The Query Returns Rows From Students Even Though The USER Actually Queried on StudentBooks.

Things To Note

- Object Views of Column Objects Enable To Work With Table As if They Were Both Relational Tables And Object Relational Tables.
- When OBJECT Views Are Extended To Row OBJECT They Enable To Generate OID Values Based on Established FOREIGN KEY OR PRIMARY KEY Relationship.
- OBJECT Views Allow Us To Continue To Use The Existing Constraints And Standard INSERT, DELETE UPDATE And SELECT Statements.
- They Help in Using OOP Features Such As REFERENCES Against The Object TABLES.
- They Provide A Technological Bridge For Migrating To An OOPS DB Architecture.
- Oracle Performs JOINS That Resolve The References Defined in The Database.
- When Referenced Data is Retrieved It Beings The Entire Row Object That Was Referenced.
- To Reference The Data We Need To Establish PKOID's in The Table That is 'PRIMARY KEY' Table Relationship.
- Use MAKE_REF() To Generate References In The Table That is 'FOREIGN KEY' Table Relationship.
- Once The Above Specifications Are Completed We Can Work With The Data As if it Was Stored in OBJECT TABLES.

Procedural Extensions To Structured Query Language Using PL/SQL

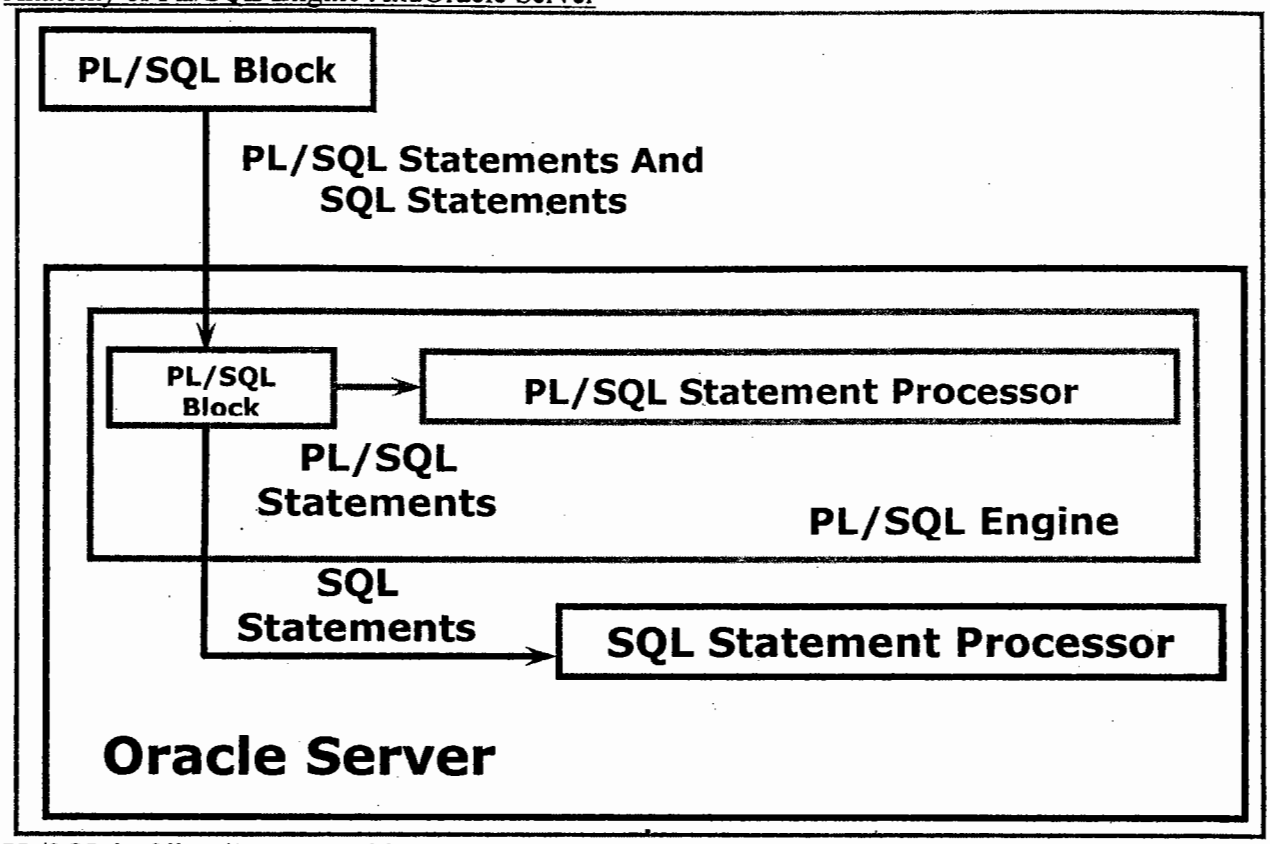
General Points To Ponder

- PL/SQL Stands For PROCEDURAL Language Extensions To SQL.
- PL/SQL Extends SQL By Adding Programming Structures And Subroutines Available in Any High Level Language.
- PL/SQL Can Be Used For Both Server-Side And Client-Side Development.
- PL/SQL Has Syntax And Rules That Determine How Programming Statements Work Together.
- PL/SQL is Not A Stand Alone Programming Language.
- Hence it Cannot Be Used Individually for Developing Applications.
- PL/SQL is A Part of The Oracle RDBMS, And Hence Can Reside in Two Environments, The Client And The Server.
- Any Module That is Developed Using PL/SQL An Be Moved Easily Between Server Side And Client Side Applications.
- Either in Client OR Server Environment Any PL/SQL Block OR Sub Routine is Processed By The PL/SQL Engine.
- PL/SQL Engine is A Special Component That Processes And Executes Any PL/SQL Statements And Sends Any SQL Statements To The SQL Statement Processor.
- The SQL Statement Processor is Always Located On The Oracle Server And Hence All SQL Statements Are Processed on The Server Only.
- As Per The Necessity The PL/SQL Engine Can Be Located Either At
  - SERVER Side.
  - CLIENT Side.
- When PL/SQL Engine is Located Upon The SERVER, The Whole PL/SQL Block is Passed To The PL/SQL Engine on The ORACLE SERVER For Processing Including Client Components.
- This Concept Gives Security, Less Load And Performance For Applications.
- When The PL/SQL Engine is Located Upon The Client, The PL/SQL Processing is Done on The Client Side. All SQL Statements That Are Embedded Within The PL/SQL Block, Are Sent To The ORACLE SERVER For Further Processing.
- If The PL/SQL Block Does Not Contain Any SQL Statements, The Entire Block is Executed on The Client Side it Self.
- Because of The Above Concept, The Application Load in The Network Will Highly Reduce And The System Becomes highly Loose Coupled in The Development Process, Giving High Accessibility For Easy Maintenance.

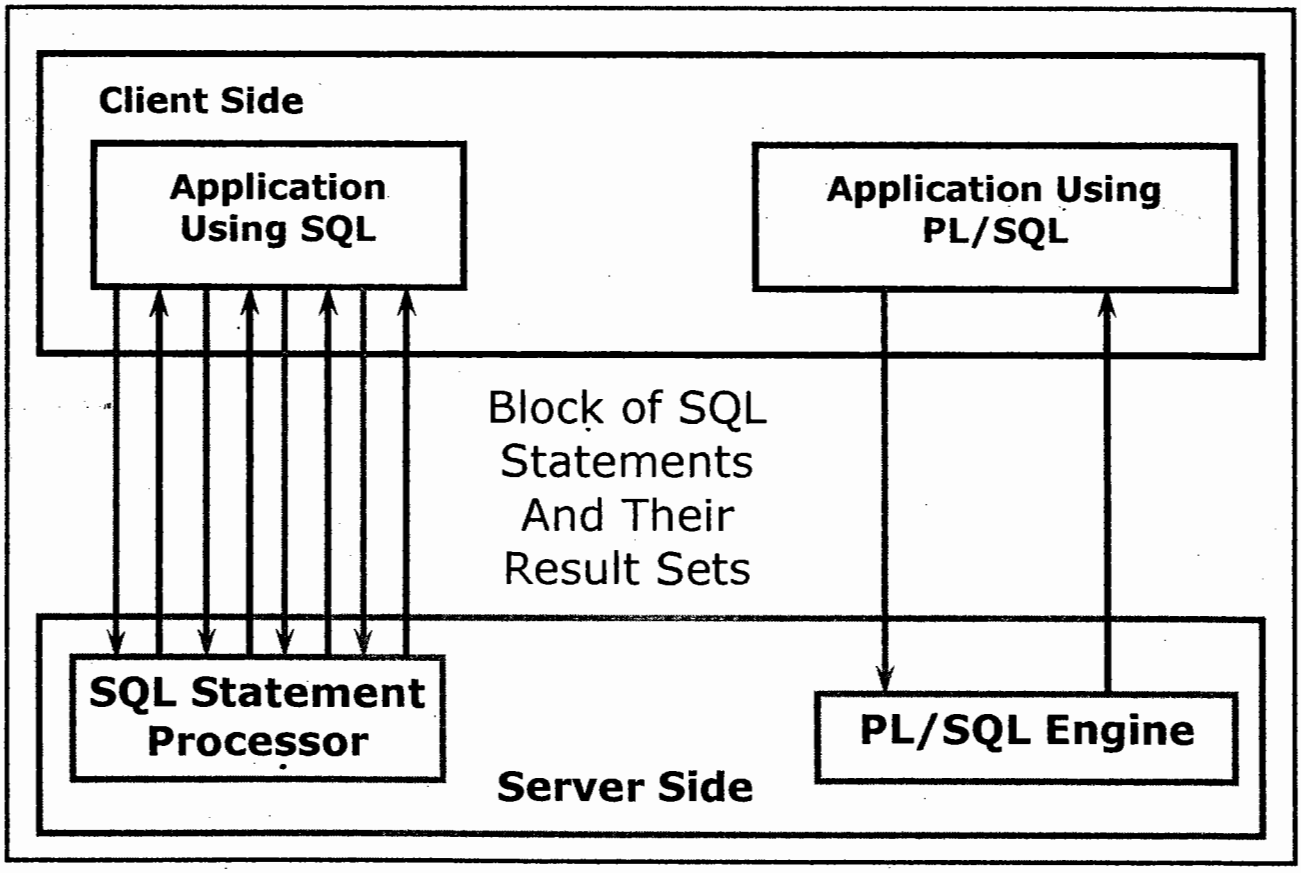
Advantages of PL/SQL

- Completely Portable.
- High Performance Transaction Processing Language.
- Support For SQL.
- Support For Object Oriented Programming.
- Better Performance.
- Higher Productivity.
- Tight Integration With Oracle Products.
- Tight Security.

Anatomy of PL/SQL Engine AndOracle Server

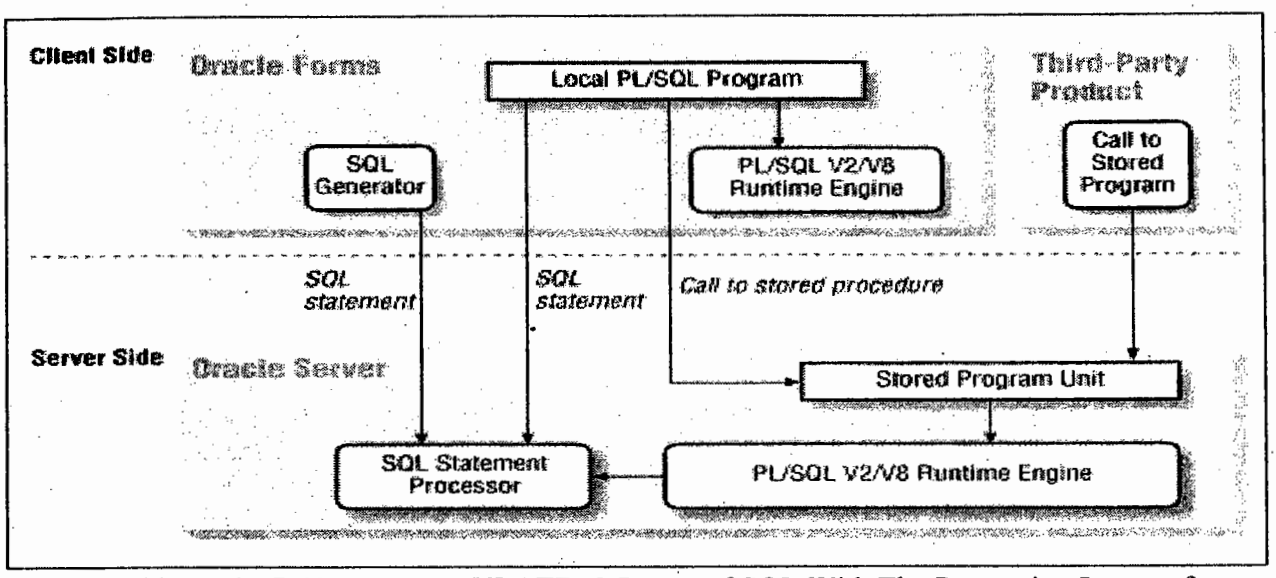


PL/SQL in Client/Server Architecture



Application Development Architecture Using PL/SQL

Main Features of PL/SQL



- It Combines The DATA MANIPULATING Power of SQL With The Processing Power of PROCEDURAL LANGUAGES.
- Program Flow Can Be Controlled Using The Statements Like "IF" And "LOOP".
- We Can Declare Variables, Define Procedures And Functions And Trap Runtime Errors.
- Can Be Used To Break Complex Problems Down Into Easily Understandable Procedural Code.
- The Code Can Be Reused For Multiple Applications.
- SQL Commands Can Be Directly Embedded Inside The PL/SQL Without Learning New API's.
- The PL/SQL Data Types Correspond With SQL's Column Types Making Learning Process Easier.

Block Structure Approach

- The Basic Units That Make Up A PL/SQL Program Are Logical Blocks.
- The Blocks in PL/SQL Can Be Nested With One Another.
- A Block Groups Related Declarations And Statements into One Single Unit.
- The Declarations Are Local To The Block And Cease To Exist OR Lost When The Block Completes.
- Block Structure Approach Avoids Cluttered Namespaces For Variable Declarations And Procedures.
- The Basic Parts of A PL/SQL Block Are
  - Declarative Part (Optional)
  - Executable Part (Mandatory)
  - Exception Handling Part (Optional.)
- The Order of The Parts of PL/SQL Block is Logical.

Declarative Part

- It is Used To Define User Defined Types, Variables And Similar Items.

Executable Part

- It Contains The Operational Code For The Program.
- The Items in Declarative Part Are Manipulated Here.

Exception Handling Part

- Any Exceptions That Are Raised During The Program Execution Are Handled Here.

Some Points To Note

- Block Can Be Nested in The Executable And Exception Handling Parts of A PL/SQL Block, OR A Sub Program.
- Local Sub Programs Can Be Defined in The Declarative Part of Any Block.
- Local Sub Programs Can Be Called From The Block in Which They Are Declared.

- A PL/SQL is Marked With Either A "DECLARE" OR "BEGIN" Keyword, And Ends With The Keyword "END".
- Only BEGIN and END Keywords Are Mandatory.
- A Semicolon(;) Has To be Placed After The "END" Keyword.

```
SQL> DECLARE
        Variable Declarations,
        Cursor Declaration,
        User_Defined Execption
        BEGIN
        SQL Statements
        PL/SQL Statement
        EXCEPTION
        Action To Perform When Errors Occur.
        END;
```

- Each Logical Block of PL/SQL Corresponds To A Problem OR Sub Problem To Be Solved in A Real Time System.
- PL/SQL Supports The Divide-And-Conquer Approach To Solve A Real Time Problem, Which is Also Called As "Stepwise Refinement".
- We Can Define Local Sub Programs in The Declarative Part of Any Block.
- However, We Can Call Local Subprograms Only From The Block in Which They Are Defined.

A Simple PL/SQL Block

```
SQL> BEGIN
        NULL;
        END;
```

NULL BLOCK

```
SQL> BEGIN
        RETURN;
        END;
```

BLOCK With RETURN Clause

```
SQL> DECLARE
        BEGIN
        NULL;
        END;
```

NULL BLOCK With DECLARE

```
SQL> DECLARE
        BEGIN
        NULL;
        EXCEPTION
        WHEN OTHERS THEN
        NULL;
        END;
```

NULL BLOCK With DECLARE And EXCEPTION Associated With A PRAGMA Clause

Executing Statements And PL/SQL Blocks From SQL*Plus

- Place A Semicolon(;) At The End of The SQL Statement OR PL/SQL Control Statement.
- Use A Forward Slash(/) To Run The Anonymous PL/SQL Block in SQL*Plus Buffer.
- Place A Period (.) To Close A SQL*Plus Buffer With Out Running The PL/SQL Program.
- A PL/SQL Block is Treated As One Continuous Statement in The SQL*Plus Buffer.
- Semicolon Within The PL/SQL Block Does Not Closes OR Runs The SQL Buffer.
- In PL/SQL An Error is Called As An Exception.
- Sections Keywords "DECLARE", "BEGIN", And "EXCEPTION" Should Not Contain A Semicolon.
- "END" And All Other PL/SQL Statements Should Be Applied With A Semicolon At The End.

Types of Blocks In PL/SQL

- A PL/SQL Program Can Be Written in Various Types of Blocks, They Are

Anonymous Blocks

- They Are Un-Named Blocks.
- They Are Declared At The Point in An Application, Where They Are To Be Executed And Are Passed To The PL/SQL Engine For Execution At Runtime.
- An Anonymous Block Can Be Embedded Within A Pre-Compiler Program And Within SQL*Plus OR Server Manager.

```
SQL> DECLARE
       BEGIN
       NULL;
       EXCEPTION
       WHEN OTHERS THEN
       NULL;
       END;
```

Anonymous NULL BLOCK With DECLARE And EXCEPTION Associated With A PRAGMA Clause

Named Blocks

- They Have All The Features As Specified For The Anonymous Blocks, But The Only Difference is That Each Block Can Be Named if Necessary.
- Named Blocks Help in Associating With The Scope And Resolution of Variables in Different Blocks.
- They Give The Specifications of The Named Spaces As Provided in High Level OO Languages Like C++ And JAVA.
- Named Blocks Are Conveniences For Variable Management.

```
SQL> <<FirstBlock>>
       DECLARE
       BEGIN
       NULL;
       EXCEPTION
       WHEN OTHERS THEN
       NULL;
       END FirstBlock;
```

Named NULL BLOCK With DECLARE And EXCEPTION Associated With A PRAGMA Clause

- Named Blocks Make The PL/SQL Blocks More Clear and Readable.
- Named Blocks Increase The Clarity of Programming When We Attempt The Nesting Process, And Control Structures.

Sub-Programmed Blocks

- These Are Named PL/SQL Blocks That Can Take Parameters And Can Be Invoked With in The Other Anonymous OR Sub-Programmed PL/SQL Blocks.
- These Blocks Are Either Declared As Procedures OR Functions.
- A Procedural Block is Used For Performing An Action, And A Functional Block is Used For Performing Computations in General.
- The Sub-Programmed Blocks Add The Concept of Code Reusability and Help The Developer To Achieve The Modular Approach.

Let Us Understand Variables in PL/SQL

Usage of Variables

Temporary Storage

- Data Can Be Temporarily Stored in One OR More Variables For Use When Validating Data Input For Processing The Data in The Data Flow Process.

Data Manipulation

- Variables Help in The Data Manipulation of Stored Values.

- Variables Can Be Used For Calculations And Other Data Manipulations Without Accessing The Database Each And Every Time, Hence They Help in Reducing The I/O Cycles.

Reusability

- Once Declared Can Be Used Repeatedly By Even in Other Blocks.

Ease of Maintenance

- Variables Can Be Declared Based on The Declarations of The Definitions of Database Columns.
- Provide Data Independence, Reduces Maintenance Costs And Allows Programs To Adapt The Changes As The Database Changes.

Handling Variables

- All Variables Are Declared And Initialized Only in The DECLARATIVE SECTION of PL/SQL Block.
- Declarations Allocate Storage Space For A Value According to Its Defined Size.
- Declarations Can Be Assigned With An Initial Value And Can Be Imposed With A NOT NULL Constraint.
- We can Reassign New Values To Variables in The Executable Section, In This Case The Existing Value of The Variable is Replaced With A New One.
- Forward References Are Not Allowed in PL/SQL.
- We Can Pass Values Into PL/SQL Subprograms Through Parameters With The Assistance of Variables.
- There Are Three Parameter Modes, They Are IN (The Default), OUT And IN OUT.
- View The Results From A PL/SQL Block Through Output of Variables.
- Reference Variables Can Be Used For Input OR Output in SQL Data Manipulation Statements.

Variable Types in PL/SQL

- All PL/SQL Variables Have A Data Type Specifying
  - Storage Format
  - Constraints
  - Valid Range
- PL/SQL Supports Four Data Type Categories
  - Scalar Data Types:
  - Composite Data Types
  - Reference Data Types
  - LOB Data Types

Scalar Data Types

- They Hold A Single Value.
- Main Data Types Are Those That Correspond To Column Types in Oracle Server Tables.
- Supports Boolean Variables.

Composite Data Types:

- Composite Data Types Are Similar To Structures in 'C' Language.
- They Help in Keeping All The Related Data Items Together As One Collection.
- They Are Applied in Three Ways
  - %ROWTYPE.
  - RECORD Type.
  - PL/SQL TABLE Type.
- Records Allows Groups of Fields To Be Defined And Manipulated In PL/SQL Blocks Very Easily.

Reference Data Types

- They Hold Values, Acting As Pointers, Which Designate Other Program Items.
- They Are Very Essential When Manipulating Collection of Data Items in Sub-Programs.

LOB Data Types

- They Hold Values Called Locators, Specifying The Location of Large Objects That Are Stored Out of Line in The PL/SQL Program.

Declaring PL/SQL Variables

Syntax

IdentifierName [CONSTANT] DataType [NOT NULL] [:= [DEFAULT] Expr];

IdentifierName

- Specifies The Name of The relevant Variable For That Block.

Constant

- Constraints The Variable Such That Its Value Cannot Change During The Program Process.
- Constants Must Be Initialized Else Raise An Exception.

DataType

- It Is A Scalar, Composite, Reference OR LOB Data Type As Applicable To The Required Situation.

NOT NULL

- Constraints A Variable Such That It Must Contain A Value And Raises If NULL is Identified.
- NOT NULL Variables Should Be Initialized Else Raise An Exception.

DEFAULT

- Sets The Default Value For The Value in The PL/SQL Program if Not Attended.

Expr

- It Is Any PL/SQL Expression That Can Be A Literal, Another Variable OR An Expression Involving Operations And Functions For Initialization.

Illustration

```
DECLARE
V_Sample1       NUMBER(2);
V_Sample2       CONSTANT NUMBER(2) := 99;
V_Sample3       NUMBER(2) NOT NULL := 0;
V_Sample4       NUMBER(2) := 50;
V_Sample5       NUMBER(2) DEFAULT 25;
```

Detailed Aspects of PL/SQL Language

Character Set

- PL/SQL Programs Are Written As Lines of Text Using A Specific Set of Characters.
- The Characters That Can Be Used Are
  - 'A' To 'Z' OR 'a' OR 'z'
  - 0 To 9
  - () + - */< > = ! ~ ^ : Etc.
  - Tabs, Space And Carriage Returns
- PL/SQL Key Words Are Not Case Sensitive, Hence Lower Case Letters Are Equivalent To Corresponding Upper Case Letters.

Lexical Units

- A Group of Characters That Are Contained Within A Line of PL/SQL Statement Are Called As Lexical Units.
- The Types of Lexical Units Are
  - Delimiters
  - Identifiers
  - Literals
  - Comments

Delimiters

- It is A Simple OR Compound Symbol That Has Special Meaning To PL/SQL.
- The Delimiters Are Used To Represent Arithmetic Operations Etc in PL/SQL Program.

Identifiers

- They Are Used To Name PL/SQL Program Items And Units.
- Identifiers Can Include
  - Constants
  - Variables
  - Exception
  - Cursor Variables
  - Subprograms
  - Packages
- An Identifiers Consists of A Letter Followed By A Set of Letters, Numbers, Dollar Signs Etc.
- Identifiers Are Not Case Sensitive, Until They Are Not Declared Within Quotes.
- An Identifiers Length Cannot Exceed 30 Characters.

Valid Identifier	Invalid Identifier
X	This&That
P2	You–Me
V_Phone#	Show/Off
V_Phone$	Number

Delimiters

- It is A Simple OR Compound Symbol That Has Special Meaning To PL/SQL.
- The Delimiters Are Used To Represent Arithmetic Operations Etc in PL/SQL Program.

Reserved Words

- Reserved Words Have Special Syntactic Meaning To PL/SQL.
- Reserved Words Can Be Used in Combination With Other Words For Declaration.
- It is Better To Represent Reserved Words in Upper Casing For Clarity.

Predefined Identifiers

- The Identifiers Globally Declared In Package STANDARD Can Be Re Declared.
- Declaring Predefined Identifiers Again is Error Prone As The Local Declaration Overrides The Global Declaration.

Quoted Identifiers

- Identifiers Declared in Double Quotes Are Called Quoted Identifiers.
- The Maximum Size of A Quoted Identifier Cannot Exceed 30 Characters.
- Using PL/SQL Reserved Words As Quoted Identifiers is Not Advisable.

Example : "Employee(s)"
"Status On/Off"

Literals

- A Literal is An Explicit Numeric, Character, String OR Boolean Value Not Represented By An Identifier.

Comments In PL/SQL

- PL/SQL Compiler Ignores Comments.
- Comments Promote Readability And Aids Understanding.
- PL/SQL Supports Two Styles.
  - Single Line Comments (--)
  - Multi Line Comments (/* ..... */)
- Comments Should Appear Within A Statements At The End of A Line, And Cannot Be Nested.
- It is Most Preferable To Use Multi Line Comments Rather Than Single Line Comments.

Declarations in PL/SQL

- The Program Stores Values in Variables.

- As The Program Executes, The Values of Variables Can Change But The Values of Constants Cannot Change.
- Declarations Allocate Storage Space For A Value, Hence Should Be Specified By A Data Type, And Name For Further Reference.
- Any Variable Can Be Declared Only in The Declarative Part of Any PL/SQL Block, Subprogram OR Package.

Illustration

```
DECLARE
AdmnDate        DATE;
PatientName     VARCHAR2(30);
BPCount         SMALLINT := 0;
```

Assigning Values To A Variable

- In PL/SQL A Variable Can Be Assigned In Three Ways.
  - By Using Assignment Operator (:=).
  - By Selecting OR Fetching Database Values.
  - By Passing It As An OUT OR IN OUT Parameter To A Subprogram.

Presenting Data On To The Screen

- For Producing Outputs On The Video Device We Need The Assistance of DBMS_OUTPUT Package.
- The Package Enables To Display Output From PL/SQL Blocks And Sub-Programs.
- The Procedure PUT_LINE Outputs Information To A Buffer in The SGA.
- The Information Can Be Displayed By Calling The Procedure GET_LINE OR By Setting SERVEROUTPUT in SQL*Plus.

Syntax

DBMS_OUTPUT.PUT_LINE('Message');

- DBMS_OUTPUT Has Limitation For Maximum of 255 Character Per Line.
- The DBMS_OUTPUT Package is Owned By The Oracle User SYS.
- It Writes Information To The Buffer For Storage.
- The Size of The Buffer Can Be Set Between 2000 To 10,00,000 Bytes.
- The Specifications To Set Buffers Are
  - SET SERVEROUTPUT ON;
  - SET SERVEROUTPUT ON SIZE 5000;
  - SET SERVEROUTPUT OFF;

Illustrations

```
SQL> BEGIN
        DBMS_OUTPUT.PUT_LINE('First Program in PL/SQL');
        DBMS_OUTPUT.PUT_LINE('Illustration By');
        DBMS_OUTPUT.PUT_LINE('''Satish K Yellanki''');
        END;
        /
```

Steps To Compile And Produce Output

- At SQL Prompt Type
  SQL> SET SERVEROUTPUT ON;
- Place Forward Slash And Press Enter Key.
  SQL>/

```
SQL> BEGIN
        DBMS_OUTPUT.ENABLE;
        DBMS_OUTPUT.PUT_LINE('First Program in PL/SQL');
        DBMS_OUTPUT.PUT_LINE('Illustration By');
        DBMS_OUTPUT.PUT_LINE('''Satish K Yellanki''');
```

```
END;
/
```

- DBMS_OUTPUT.ENABLE Once Declared Will Make The SERVEROUTPUT Process to Get Activated.

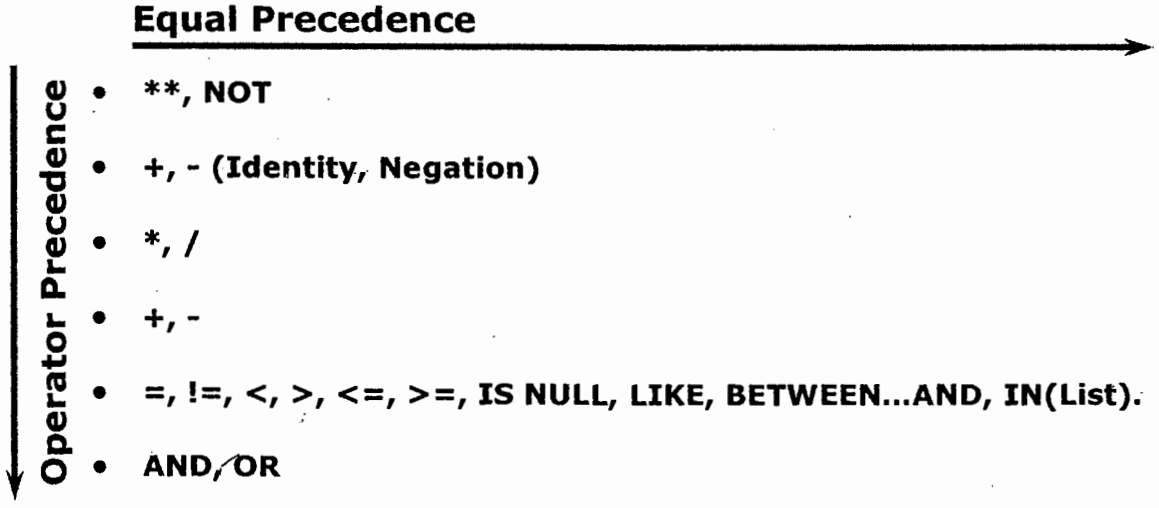
```
SQL> DECLARE
        V_Sample1 NUMBER(2);
        V_Sample2 CONSTANT NUMBER(2) := 99;
        V_Sample3 NUMBER(2) NOT NULL := 0;
        V_Sample4 NUMBER(2) := 50;
        V_Sample5 NUMBER(2) DEFAULT 25;
        BEGIN
        DBMS_OUTPUT.PUT_LINE('The Value in Sample1 : '|| V_Sample1);
        DBMS_OUTPUT.PUT_LINE('The Value in Sample2 : '|| V_Sample2);
        DBMS_OUTPUT.PUT_LINE('The Value in Sample3 : '|| V_Sample3);
        DBMS_OUTPUT.PUT_LINE('The Value in Sample4 : '|| V_Sample4);
        DBMS_OUTPUT.PUT_LINE('The Value in Sample5 : '|| V_Sample5);
        END;
        /

SQL> BEGIN
        DBMS_OUTPUT.PUT_LINE('Text With Out Quotes');
        DBMS_OUTPUT.PUT_LINE('''Text With in Quotes''');
        DBMS_OUTPUT.PUT_LINE('''Text" "With" "in" "Quotes''');
        DBMS_OUTPUT.PUT_LINE(Q'!'Text With in Quotes'!');
        DBMS_OUTPUT.PUT_LINE(Q'!'Text' 'With' 'in' 'Quotes'!');
        END;
        /

SQL> DECLARE
        MyNumber1 NUMBER := 10;
        MyNumber2 NUMBER := 20;
        BEGIN
        DBMS_OUTPUT.PUT_LINE('Your First Number is : '||MyNumber1);
        DBMS_OUTPUT.PUT_LINE('Your Second Number is : '||MyNumber2);
        DBMS_OUTPUT.PUT_LINE('The Sum of '||MyNumber1||' and '||MyNumber2||' is :
        '||TO_CHAR(MyNumber1 + MyNumber2, '999D99'));
        END;
        /

SQL> DECLARE
        String1 VARCHAR2(30) := '&String1';
        String2 VARCHAR2(30) := '&String2';
        BEGIN
        DBMS_OUTPUT.PUT_LINE('Your First String  is : '||String1);
        DBMS_OUTPUT.PUT_LINE('Your Second  String is : '||String2);
        DBMS_OUTPUT.PUT_LINE('Your Final String  is : '||String1||String2);
        END;
        /

SQL> DECLARE
        V_FirstName   VARCHAR2(30) := '&FName';
        V_MiddleName        VARCHAR2(30) := '&MName';
        V_LastName   VARCHAR2(30) := '&LName';
        V_DOB               DATE := '&DateOfBirth';
        BEGIN
```

```
        DBMS_OUTPUT.PUT_LINE('Your First Name is : '||V_FirstName);
        DBMS_OUTPUT.PUT_LINE('Your Middle Name is : '||V_MiddleName);
        DBMS_OUTPUT.PUT_LINE('Your Last Name is : '||V_LastName);
        DBMS_OUTPUT.PUT_LINE('******Concatenating******');
        DBMS_OUTPUT.PUT_LINE('Your Full Name is : '||V_FirstName||' '||V_MiddleName||'
        '||V_LastName);
        DBMS_OUTPUT.PUT_LINE('Your Date of Birth is : '|| V_DOB);
        DBMS_OUTPUT.PUT_LINE('Your Were Born on : '|| TO_CHAR(V_DOB, 'Day'));
        DBMS_OUTPUT.PUT_LINE('Your Present Age is : '|| TRUNC((SYSDATE -
        V_DOB)/365));
        END;
        /
SQL> DECLARE
        V_FirstNum NUMBER := &FNumber;
        V_SecondNum NUMBER := &SNumber;
        V_Result NUMBER := 0;
        BEGIN
        DBMS_OUTPUT.PUT_LINE('You Gave Me '||V_FirstNum||' And '||V_SecondNum);
        DBMS_OUTPUT.PUT_LINE('I Executed Their Sum And The Result is : '||
        TO_CHAR(V_FirstNum + V_SecondNum, '9999.99'));
        DBMS_OUTPUT.PUT_LINE('As You Did Not Reinitialize The Result Variable It is
        Having : '||V_Result||' Only');
        END;
        /
```

Operators in PL/SQL

- Logical : AND, OR, NOT
- Arithmetic : +, -, *, /
- Concatenation : ||
- Parenthesis To Control Order of Operations
- Exponentiation : **
- Comparison : =, !=, <, >, <=, >=
- SQL*Plus : IS NULL, LIKE, BETWEEN...AND..., IN

Operator Precedence



Operators in PL/SQL, Things To Note

- Comparison Involving NULL's Always Yields NULL Value.
- Applying Logical Operator NOT To A NULL Yields NULL.

- In Conditional Control Statements, If The Condition Yields NULL, Its Associated Sequence of Statements Are Not Executed.

Scope And Visibility Of PL/SQL Identifiers

- References To The Identifiers Are Resolved According To The Scope And Visibility Concept in PL/SQL.

Scope

- Scope of An Identifier is That Region of A Program Unit, From Which We Can Reference The Identifier Either With Named Reference OR Without Named Reference.
- Once The Scope of The Variable is Lost it Means That The Life of The Variable No More.
- All Variables Loose Their Scope As Soon As The PL/SQL Block Ends OR Terminates.
- Oracle Automatically Releases The Space Upon That Variable Once its Scope is Closed.

Visibility

- It is The Region From Which We Can Reference The Identifier Using An Unqualified Name.
- If A Variable Looses Its Visibility But is Carrying Scope, Then We Can Extend The Visibility of The Variable Using The Qualified Name.
- Identifiers Declared In A PL/SQL Block Are Considered Local To That Block And Global To All Its Sub-Blocks
- If A GLOBAL IDENTIFIER is Re-Declared In A Sub Block, Both Identifiers Remain In Scope.
- Within The Sub Block, Only The Local Identifier Are Visible.
- A Qualified Name Should Be Used To Reference The Global Identifier.
- The Same Identifier Can Be Declared In Two Different Blocks, The Change In One Does Not Affect The Other.

Concepts of Scoping

- With In The Same Scope, All Identifiers Must Be Unique.
- Even When The Data Types Are Differing, Variables And Parameters Cannot Share The Same Name.

```
SQL> DECLARE
        V_Boolean    BOOLEAN;
        V_Boolean    VARCHAR2(5) := 'SAMPLE';
        BEGIN
        DBMS_OUTPUT.PUT_LINE('The Value is : '||V_Boolean);
        V_Boolean := FALSE;
        END;
```

Name Resolution Standards

- The Names of Database Columns Take Precedence Over The Names of Local Variables And Formal Parameters.
- To Avoid The Ambiguity of Name Resolution, Add A Prefix To The Names of The Local Variables And Formal Parameters, OR A Block Label Can Be Used To Qualify References.
- Hence it is Better Not To Declare A Variable Name in PL/SQL Block Which is Equal To The Name of The SQL Table OR Column Name.

Scope And Visibility Diagram

```
SQL> DECLARE
        X BINARY_INTEGER;
        BEGIN
        /*Some Operational Statements*/
            DECLARE
            Y NUMBER(4);
            BEGIN .
            /*Some Operational Statements*/
            END;
        /*Some Operational Statements*/
        END;
```

Nested Blocks And Variable Scope

- The Variable 'Y' Can Reference The Variable Named 'X', But The Variable 'X' Cannot Reference Variable 'Y'.
- If The Variable Named 'Y' In The Nested Block is Given The Same Name As The Variable Name 'X' In The Outer Block Its Value is Valid Only For The Duration of The Nested Block.
- To Avoid The Practical Confusion It is Better Not To Repeat The Variable Names in Declaration.
- They Cannot Be Repeated in The Same Block, In Different Blocks They Are Referenced Using The Name Resolution Technique.

```
SQL> DECLARE
        X REAL := 205;
        BEGIN
        DBMS_OUTPUT.PUT_LINE('The Value of Outer X : '||X);
              DECLARE
              X REAL := 405;
              BEGIN
              DBMS_OUTPUT.PUT_LINE('The Value of Inner X : '||X);
              END;
        DBMS_OUTPUT.PUT_LINE('The Value  of Outer X : '||X);
        END;
        /
```

Using Named Blocks To Increase Visibility

```
SQL> <<OuterBlock>>
        DECLARE
        X REAL := 205;
        BEGIN
        DBMS_OUTPUT.PUT_LINE('The Value of Outer X : '|| X);
        <<InnerBlock>>
        DECLARE
        X REAL := 405;
        BEGIN
        DBMS_OUTPUT.PUT_LINE('The Value of Inner X : '||X);
        DBMS_OUTPUT.PUT_LINE('The Value of Inner X : '||OuterBlock.X);
        END;
        DBMS_OUTPUT.PUT_LINE('The Value of Outer X : '|| X);
        END;
        /
```

```
SQL> DECLARE
        A VARCHAR2(30) := 'Hello Guys';
        B REAL := 300;
        BEGIN
        DBMS_OUTPUT.PUT_LINE('The Value of Outer String A is Available Here : ' ||A);
        DBMS_OUTPUT.PUT_LINE('The Value of Outer Real B is Availabe Here : ' ||B);
              DECLARE
              A INTEGER := 2000;
              B REAL := 400;
              BEGIN
              DBMS_OUTPUT.PUT_LINE('The Value of Inner Integer A is Availabe Here : ' ||A);
              DBMS_OUTPUT.PUT_LINE('The Value of Inner Real B is Availabe : ' ||B);
              END;
              DECLARE
              D REAL := 500;
              A VARCHAR2(30) := 'Hello Gals';
              BEGIN
```

```
            DBMS_OUTPUT.PUT_LINE('The Value of Last D is Availabe : ' ||D);
            DBMS_OUTPUT.PUT_LINE('The Value of Last String A is Availabe Here : ' ||A);
            END;
        DBMS_OUTPUT.PUT_LINE('The Value of Outer String A is Availabe : ' ||A);
        DBMS_OUTPUT.PUT_LINE('The Value of Outer Real B is Availabe : ' ||B);
        END;
        /
SQL> <<FirstBlock>>
        DECLARE
        A VARCHAR2(30) := 'Hello Guys';
        B REAL := 300;
        BEGIN
        DBMS_OUTPUT.PUT_LINE('The Value of Outer String A is Availabe Here : ' ||A);
        DBMS_OUTPUT.PUT_LINE('The Value of Outer Real B is Availabe Here : ' ||B);
            DECLARE
            A INTEGER := 2000;
            B REAL := 400;
            BEGIN
            DBMS_OUTPUT.PUT_LINE('The Value of Inner Integer A is Availabe Here : ' ||A);
            DBMS_OUTPUT.PUT_LINE('The Value of Inner Real B is Availabe : ' ||B);
            DBMS_OUTPUT.PUT_LINE('The Value of Outer String A is Availabe Here : '
            ||FirstBlock.A);
            END;
            DECLARE
            D REAL := 500;
            A VARCHAR2(30) := 'Hello Gals';
            BEGIN
            DBMS_OUTPUT.PUT_LINE('The Value of Last D is Available : ' ||D);
            DBMS_OUTPUT.PUT_LINE('The Value of Outer String and Last String together is :
            ' ||FirstBlock.A||' and '||A);
            END;
        DBMS_OUTPUT.PUT_LINE('The Value of Outer String A is Available : ' ||A);
        DBMS_OUTPUT.PUT_LINE('The Value of Outer Real B is Available : ' ||B);
        END <<FirstBlock>>;
```

Variable Types in PL/SQL Programming

- While Passing The Values To Arguments At Runtime, The PL/SQL Variables Can Be Treated As Two Types
  - Bind Variables
  - PL/SQL Variables

Bind Variables

- Bind Variables Are Declared in SQL*Plus Session And Can Be Referenced in Both SQL*Plus Sessions As Well As PL/SQL Program.
- Before Referencing The Bind Variable in PL/SQL It Must Be Declared in SQL*Plus Session.

Bind Variable Syntax

SQL> VARIABLE <VariableName> DataType

- In PL/SQL To Reference Bind Variables, The Bind Variable Must Be Prefixed With A Colon(:).
- In SQL*Plus The Bind Variable Has To Be Displayed Using The PRINT Command.

```
SQL> VARIABLE G_SALARY  NUMBER;
SQL> DECLARE
        V_sal NUMBER(7,2);
        BEGIN
          SELECT Sal INTO V  Sal
          FROM Emp
          WHERE Empno = 7369;
```

```
        :G_Salary := V_Sal;
    END;
    /
SQL> PRINT G_Salary;
    (OR)
SQL> SELECT :G_Salary FROM DUAL;
SQL> VARIABLE Bind_Variable NUMBER;
SQL> DECLARE
    V_Number1 NUMBER(9,2):=&B_Number1;
    V_Number2 Number(9,2) :=&B_Number2;
    BEGIN
    :Bind_Variable := (V_Number1 / V_Number2) + V_Number2;
    END;
    /
SQL> VARIABLE B_Total NUMBER
SQL> DECLARE
    V_Salary NUMBER(9, 2) := &B_Salary;
    V_Bonus NUMBER(9, 2) := &B_Bonus;
    BEGIN
    :B_Total := NVL(V_Salary, 0) * (1 + NVL(V_Bonus, 0)) / 100;
    END;
    /
```

Step 1

```
SQL> DECLARE
    V_Sal NUMBER(7, 2);
    V_Empno NUMBER(4) := &GiveEmpno;
    BEGIN
    SELECT Sal INTO V_Sal
    FROM Emp
    WHERE Empno = V_Empno;
    :B_Sal := V_Sal;
    :B_Empno := V_Empno;
    END;
    /
```

Step 2

```
SQL> DECLARE
    V_Comm NUMBER(7, 2);
    BEGIN
    SELECT Comm INTO V_Comm
    FROM Emp
    WHERE Empno = :B_Empno;
    :B_Comm := V_Comm;
    END;
    /
```

Step 3

```
SQL> BEGIN
    DBMS_OUTPUT.PUT_LINE('The Total Salary of '||(:B_Empno)||' is '||(:B_Sal +
    NVL(:B_Comm, 0)));
    END;
    /
```

PL/SQL Variables

- These Are Substitution Variables That Are Directly Declared Within The Program.
- These Variables Can Be Associated To The Variables Declared In ACCEPT Command OR At Run Time.
- The PL/SQL Variables Can Be Projected Using The DBMS_OUTPUT.PUT_LINE Method Within PL/SQL Program Only.
- The Life of PL/SQL Variable is With in The Scope of The PL/SQL Block Only.

```
SQL> DECLARE
        V_Number1 NUMBER(9, 2) := &P_Number1;
        V_Number2 Number(9, 2) := &P_Number2;
        V_Result NUMBER(9, 2);
        BEGIN
        V_Result := (V_Number1 / V_Number2) + V_Number2;
        DBMS_OUTPUT.PUT_LINE(V_Result);
        END;
        /
```

Code Maintenance Can Be Made Easier By

- Documenting Code With Comments.
- Developing A Case Convention For The Code.
- Developing Naming Conventions For Identifiers And Other Objects.
- Enhancing Readability By Indenting.
- SQL Statements, PL/SQL Key Words And Data Types Are Kept In Uppercasing.
- Identifiers And Parameters, Database Tables And Columns Are Kept In Lower Casing And Initcap Casing.
- The Names of Local Variables And Formal Parameters Take Precedence Over The Names of Database Tables.
- The Names of Columns Take Precedence Over The Names of Local Variables.

Let Us Understand The Intelligence of PL/SQL Using Control Structures

- To Write Programs That Reflect The Real Time Requirement, We Need
  - Branching
  - Selection And
  - Looping.
- Control Structures Are The Most Important PL/SQL Extensions To SQL.
- PL/SQL Helps Us Manipulate And Process Oracle Data Using
  - Conditional Statements.
  - Iterative Statements.
  - Sequential Flow-Of-Control Statements.
- The Statement Structures Provided By PL/SQL For This Purpose Are
  - IF-THEN-ELSE
  - END IF
  - ELSIF
  - CASE
  - END CASE
  - END
  - LOOP
  - FOR-LOOP
  - WHILE-LOOP
  - END LOOP
  - EXIT-WHEN And
  - GOTO.
- By Integrating All The Above Statements In A Proper Way A PL/SQL Programmer Can Handle Any Real Time Situation.

Conditional Statements

- Conditional Statements In PL/SQL Are Provided In Two Different Flavors
  - Branching Statements.
  - Selection Statements.

Branching Statements

- In PL/SQL Branching is Implemented Using The "IF...THEN...ELSE..." Statements.
- PL/SQL Provides Three Types of Branching Control.
  - Simple IF..END IF Statements.
  - IF ELSE..END IF Statements.
  - IF..ELSIF..END IF Statements.
  - The Branching Standards Can Be Implemented Either As
    - Simple IF Statements.
    - Nested IF Statements.
    - ELSE IF OR ELSIF Ladders.

General Syntax

```
IF Condition1 THEN
        Statement1;
        Statement2;
END IF;
```

General Points To Ponder

- "IF...THEN" is A Reserved Word And Marks The Beginning of The "IF" Statement.
- The "END IF" is A Reserved Phrase That Indicates The End of The "IF..THEN" Construct.
- When "IF...THEN" is Executed, A Condition is Evaluated To Either "TRUE" OR "FALSE"
- The Condition Need Not Be Specified in Brackets.
- Conditions Are Compared By Using Either The Comparison Operators OR SQL*Plus Operators.
- One "IF" Keyword Can Manage Any Number of Conditions At A Point of Time Using The LOGICAL CONNECTIVITIES Like "AND", "OR".

- Even Though The Multiple Conditions Need Not Be Constrained By Using Brackets, It is Better To Control Their Precedence Using The Proper Implementation of Brackets To Avoid Ambiguity.
- Every "IF" That is Implemented Needs Compulsorily A "TRUE" State Evaluation For its Successful Execution, But An Evaluation State is Not Compulsory When The Condition is "FALSE".
- After The Actual Job of "IF" Has Been Completed, The Control Structure Returns The Job To The Original State of The PL/SQL Block.

```
DECLARE
V_Number1 NUMBER := &Number1;
V_Number2 NUMBER := &Number2;
V_TEMP  NUMBER;
BEGIN
DBMS_OUTPUT.PUT_LINE('Original V_Number1 = '|| V_NUMBER1);
DBMS_OUTPUT.PUT_LINE('Original V_Number2 = '|| V_Number2);
      IF V_Number1 > V_Number2 THEN
          V_TEMP := V_Number1;
          V_Number1 := V_Number2;
          V_Number2 := V_TEMP;
      END IF;
DBMS_OUTPUT.PUT_LINE('Swapped V_Number1 = '|| V_NUMBER1);
DBMS_OUTPUT.PUT_LINE('Swapped V_Number2 = '|| V_Number2);
END;
```

IF...THEN...ELSE...END IF Statement

- IF...THEN...ELSE Statement Enables Us To Specify Two Different Groups of Statements For Execution.
- One Group is Evaluated When The Condition Evaluates To "TRUE", The Next Group is Evaluated When The Condition Evaluates To "FALSE".

Syntax

```
IF Condition THEN
    Statement 1;
ELSE
    Statement 2;
END IF;
```

- This Concept Should Be Used When Trying To Choose Between Two Mutually Exclusive Actions.

```
SQL> DECLARE
        V_Num NUMBER := &EnterNumber;
        BEGIN
        IF MOD(V_Num,2) = 0 THEN
        DBMS_OUTPUT.PUT_LINE(V_Num||' is an Even Number.');
        ELSE
        DBMS_OUTPUT.PUT_LINE(V_Num||' is an Odd Number.');
        END IF;
        END;
        /
SQL> DECLARE
        V_Number1 NUMBER := &Number1;
        V_Number2 NUMBER := &Number2;
        BEGIN
        IF
        V_Number1 > V_Number2 THEN
        DBMS_OUTPUT.PUT_LINE('The Greatest Number is : '||V_Number1);
        ELSE
```

```
        IF V_Number2 > V_Number1 THEN
        DBMS_OUTPUT.PUT_LINE('The Greatest Numeber is : '||V_Number2);
        ELSE
        DBMS_OUTPUT.PUT_LINE('The Numbers are equal '||V_Number1 ||' and
        '||V_Number2);
        END IF;
    END IF;
    END;
```

Behavior of NULL's .

- In Simple "IF" When A Condition is Evaluated To NULL, Then The Statements in "TRUE" State Will Not Be Executed, Instead The Control Will Be Passed To The First Executable Statement After The "END IF".
- In IF...THEN...ELSE Construct The FALSE Block is Executed Whenever The Condition Evaluates To NULL.
- Hence When Ever A Conditional Process is Executed it is Better To Cross Verify The NULL Status of Any Variable OR Value Before Execution.

```
SQL> DECLARE
        V_Num1 NUMBER := &Number1;
        V_Num2 NUMBER := &Number2;
        BEGIN
        IF V_Num1 = V_Num2 THEN
        DBMS_OUTPUT.PUT_LINE('Given Numbers are Equal');
        END IF;
        DBMS_OUTPUT.PUT_LINE('Did you Watch The NULL Effect.');
        END;
        /
```

Note : Supply NULL At Runtime To Check The Affect of NULL.

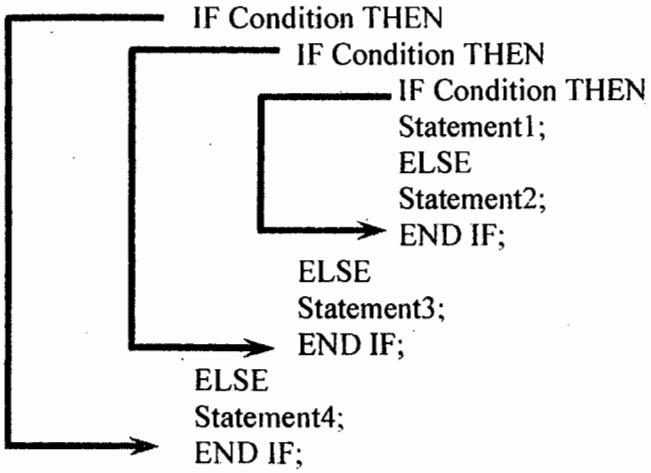
```
SQL> DECLARE
        V_Num NUMBER := &EnterNumber;
        BEGIN
        IF MOD(V_Num, 2) = 0 THEN
        DBMS_OUTPUT.PUT_LINE(V_Num||' is an Even Number.');
        ELSE
        DBMS_OUTPUT.PUT_LINE(V_Num||' is an Odd Number.');
        END IF;
        DBMS_OUTPUT.PUT_LINE('Did You Watch The Difference...');
        END;
        /
```

Note : Supply NULL At Runtime To Check The Affect of NULL.

Nested "IF" Statements

- The "IF" Statements Can Be Nested Into One Another As Per Requirements.
- Nested "IF" is A Situation in Which An "IF" Follows Another "IF" Immediately For Every "TRUE" State of An "IF" Condition.
- Each "IF" is Considered As An Individual Block of "IF" And Needs Proper Nesting.
- Each "IF" Block That is Opened Needs A Proper Close With "END IF". Else PL/SQL Raises Exceptions.
- Nested "IF" Situations Have To Be Planned When We Have A Series of Conditions Fall Sequentially.

Syntax:




```
SQL> DECLARE
        V_Year NUMBER := &Year;
        BEGIN
        IF MOD(V_Year,4) = 0 THEN
              IF MOD(V_Year,100) <> 0 THEN
              DBMS_OUTPUT.PUT_LINE(V_Year || ' is a Leap Year');
              ELSE
                    IF MOD(V_Year,400) = 0 THEN
                    DBMS_OUTPUT.PUT_LINE(V_Year || ' is a Leap Year');
                    ELSE
                    DBMS_OUTPUT.PUT_LINE(V_Year || ' is a not a Leap Year');
                    END IF;
              END IF;
        ELSE
        DBMS_OUTPUT.PUT_LINE(V_Year || ' is a not a Leap Year');
        END IF;
        END;
       /
```

Branching With Logical Connectivity's

- In This Situation One "IF" is Associated With A Collection of Conditions Using Either Logical "AND" OR Logical "OR" Operator.

Syntax1

```
IF Condition1 AND Condition2 THEN
        Statement1;
        Statement2;
ELSE
        Statement3;
        Statement4;
END IF;
```

Syntax2

```
IF Condition1 OR Condition2 THEN
        Statement1;
        Statement2;
ELSE
        Statement3;
        Statement4;
END IF;
```

Syntax3

```
IF Condition1 AND Condition2 OR Condition3 THEN
        Statement1;
        Statement2;
ELSE
        Statement3;
        Statement4;
END IF;
```

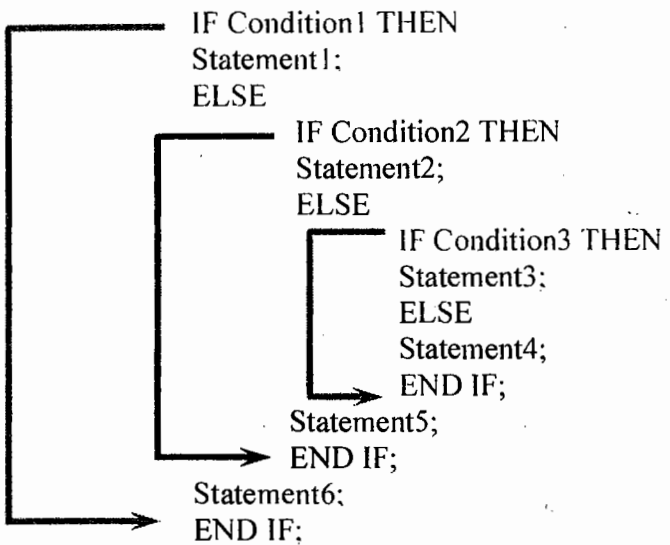
```
SQL> DECLARE
        V_Name VARCHAR2(20) := INITCAP('&GiveName');
        V_Graduate CHAR(1) := UPPER('&Graduate');
        V_Passport CHAR(1) := UPPER('&Passport');
        V_Toefl CHAR(1) := UPPER('&Toefl');
        BEGIN
        IF V_Graduate = 'Y' AND V_Passport = 'Y' AND V_Toefl = 'Y' THEN
            DBMS_OUTPUT.PUT_LINE('Congratulations '||V_Name||' You Are Eligible To
            Apply For US Universities.');
        ELSE
            DBMS_OUTPUT.PUT_LINE('Sorry! '||V_Name||' Requirements Do Not Match. Try
            Again.');
        END IF;
        END;
        /
```

```
SQL> DECLARE
        V_Name VARCHAR2(20) := INITCAP('&GiveName');
        V_PAN CHAR(1) := UPPER('&PAN');
        V_DRLicence CHAR(1) := UPPER('&DRLicence');
        V_VoterCard CHAR(1) := UPPER('&VoterCard');
        BEGIN
        IF V_PAN = 'Y' AND
        (V_DRLicence = 'Y' OR V_VoterCard = 'Y') THEN
        DBMS_OUTPUT.PUT_LINE('Congratulations, '||V_Name||' You Are Eligible For Taking
        The Bank Loan.');
        ELSE
        DBMS_OUTPUT.PUT_LINE('Sorry! '||V_Name||' Requirements Do Not Match. Try
        Again.');
        END IF;
        END;
```

"ELSE...IF" Ladder

- This is A Situation Where A Condition is Followed For Every "FALSE" State of An "IF" Condition.
- Each False State is Immediately Associated With A Condition of Its Own.
- This Concept of "ELSE...IF" Should Be Used When We Have A Series of Alternative Operations To Be Executed With An Association At Different Levels.
- In This Case Each FALSE State Has An ELSE With A Condition Applied With One Block of Branching State.
- To Avoid Confusion Proper Nesting And Indentation Has To Be Followed.

Syntax




```
SQL> DECLARE
        V_operator VARCHAR2(2) := '&Operator';
        V_Number1 NUMBER := &Operand1;
        V_Number2 NUMBER := &Operand2;
        BEGIN
        IF V_Operator = '+' THEN
        DBMS_OUTPUT.PUT_LINE('The Sum of '||V_Number1||' and '||V_Number2||' is :
        '||TO_NUMBER(V_Number1 + V_Number2));
        ELSE IF V_Operator = '-' THEN
                DBMS_OUTPUT.PUT_LINE('The Difference of '||V_Number1||' and
                '||V_Number2||' is : '||TO_NUMBER(V_Number1 - V_Number2));
                ELSE IF V_Operator = '*' THEN
                        DBMS_OUTPUT.PUT_LINE('The Product of '||V_Number1||' and
                        '||V_Number2||' is : '||TO_NUMBER(V_Number1 * V_Number2));
                        ELSE IF V_Operator = '/' THEN
                                DBMS_OUTPUT.PUT_LINE('The Quotient of '||V_Number1||'
                                and '||V_Number2||' is : '||TO_NUMBER(V_Number1 /
                                V_Number2));
                                ELSE IF V_Operator = '**' THEN
                                        DBMS_OUTPUT.PUT_LINE('The  Power of
                                        '||V_Number1||' and '||V_Number2||' is :
                                        '||TO_NUMBER(V_Number1 ** V_Number2));
                                        ELSE
                                        DBMS_OUTPUT.PUT_LINE('Invalid Operator...
                                        Please Check Your Self...!');
                                        END IF;
                                END IF;
                        END IF;
                END IF;
        END IF;
        END;
        /

SQL> DECLARE
        V_Number1 NUMBER := &Number1;
        V_Number2 NUMBER := &Number2;
        BEGIN
        IF V_Number1 > V_Number2 THEN
```

```
DBMS_OUTPUT.PUT_LINE('The Greatest Number is : '||V_Number1);
ELSE
        IF V_Number2 > V_Number1 THEN
        DBMS_OUTPUT.PUT_LINE('The Greatest Number is : '||V_Number2);
        ELSE
        DBMS_OUTPUT.PUT_LINE('The Numbers are equal '||V_Number1 ||' and
        '||V_Number2);
        END IF;
END IF;
END;
/
```

ELSIF Statements

Syntax:

```
IF Condition1 THEN
Statements1;
ELSIF Condition2 THEN
        Statement2;
        ELSIF Condition3 THEN
                Statement3;
                ELSE
                Statement n;
                END IF;
```

- In This Construct For Every False State of An IF Condition Another IF Follows.
- This Construct is Also Specified As ELSIF Ladders.

ELSIF AND "ELSE...IF" Comparison

Syntax:

```
IF Condition1 THEN
        Statements1;
ELSIF Condition2 THEN
        Statement2;
ELSIF Condition3 THEN
        Statement3;
        ELSE
        Statement n;
        END IF;
```

Syntax:

```
IF Condition1 THEN
        Statement1;
ELSE  IF Condition2 THEN
        Statement2;
        ELSE IF Condition3 THEN
                Statement3;
                ELSE
                Statement4;
                END IF;
        END IF;
END IF;
```

```
SQL> DECLARE
        V_operator VARCHAR2(2) := '&Operator';
        V_Number1 NUMBER := &Operand1;
        V_Number2 NUMBER := &Operand2;
        BEGIN
        IF V_Operator = '+' THEN
        DBMS_OUTPUT.PUT_LINE('The Sum of '||V_Number1||' and '||V_Number2||' is :
        '||TO_NUMBER(V_Number1 + V_Number2));
        ELSIF V_Operator = '-' THEN
                DBMS_OUTPUT.PUT_LINE('The Difference of '||V_Number1||' and
                '||V_Number2||' is : '||TO_NUMBER(V_Number1 - V_Number2));
        ELSIF V_Operator = '*' THEN
                DBMS_OUTPUT.PUT_LINE('The Product of '||V_Number1||' and '||V_Number2||'
                is : '||TO_NUMBER(V_Number1 * V_Number2));
        ELSIF V_Operator = '/' THEN
```

```
            DBMS_OUTPUT.PUT_LINE('The Quotient of '||V_Number1||' and '||V_Number2||'
            is : '||TO_NUMBER(V_Number1 / V_Number2));
        ELSIF V_Operator = '**' THEN
            DBMS_OUTPUT.PUT_LINE('The Power of '||V_Number1||' and '||V_Number2||' is
            : '||TO_NUMBER(V_Number1 ** V_Number2));
        ELSE
            DBMS_OUTPUT.PUT_LINE('Invalid Operator... Please Check Your Self...!');
        END IF;
        END;
        /

SQL> DECLARE
        V_Sal NUMBER(8,2) := 25005;
        V_Name VARCHAR2(30) := INITCAP('&GiveName');
        V_Result VARCHAR2(400);
        V_Gender CHAR(1) := UPPER('&GiveGender');
        V_Marital CHAR(1) := UPPER('&GiveMaritalStatus');
        V_Country VARCHAR2(30) := UPPER('&GiveCountry');
        BEGIN
        DBMS_OUTPUT.PUT_LINE('The Country you Gave is : '||V_Country);
        IF V_Marital IN('M','S') THEN
            IF V_Gender = 'M' THEN
                V_Result := 'Mr. '||INITCAP(V_Name)||', Your Salary as per your country is';
            ELSIF V_Gender = 'F' THEN
          IF V_Marital = 'S' THEN
              V_Result := 'Miss. '||INITCAP(V_Name)||', Your Salary as per your country is';
          ELSE
              V_Result := 'Mrs. '||INITCAP(V_Name)||', Your Salary as per your country is';
          END IF;
        ELSE
          DBMS_OUTPUT.PUT_LINE('Sorry !'||INITCAP(V_Name)||'! the gender you gave
                                                  cannot be processed...Please Try Again.');
        END IF;
       IF V_Country IN('INDIA','BHUTAN') THEN
          DBMS_OUTPUT.PUT_LINE(V_Result||' : INR -  '||TO_CHAR(V_Sal,'0,99,999.99'));
       ELSIF V_Country = 'MAURITIUS' THEN
          DBMS_OUTPUT.PUT_LINE(V_Result||' : MUR -  '||TO_CHAR(V_Sal,'0,99,999.99'));
       ELSIF V_Country = 'NEPALESE' THEN
          DBMS_OUTPUT.PUT_LINE(V_Result||' : NPR - '||TO_CHAR(V_Sal,'0,99,999.99'));
       ELSIF V_Country = 'PAKISTAN' THEN
          DBMS_OUTPUT.PUT_LINE(V_Result||' : PKR -  '||TO_CHAR(V_Sal,'0,99,999.99'));
       ELSIF V_Country = 'SEYCHELLES' THEN
          DBMS_OUTPUT.PUT_LINE(V_Result||' : SCR -  '||TO_CHAR(V_Sal,'0,99,999.99'));
       ELSIF V_Country = 'SRI LANKAN' THEN
          DBMS_OUTPUT.PUT_LINE(V_Result||' : LKR -  '||TO_CHAR(V_Sal,'0,99,999.99'));
       ELSIF V_Country = 'INDONESIA' THEN
          DBMS_OUTPUT.PUT_LINE(V_Result||' : LKR -  '||TO_CHAR(V_Sal,'0,99,999.99'));
       ELSE
          DBMS_OUTPUT.PUT_LINE('Sorry '|| INITCAP(V_Name) ||'! Your Country Does not
                                                  Exist...Plese Try Again.');
       END IF;
     ELSE
       DBMS_OUTPUT.PUT_LINE('Sorry !'||INITCAP(V_Name)||'! Your Marital Status Cannot
                                                  be confirmed properly.');
      END IF;
```

```
END;
/
```

Selections in PL/SQL

- The SELECTION Strategy is Implemented In PL/SQL Using CASE STATEMENTS.
- The Case Statements In PL/SQL Has Two Forms
  - Simple CASE
  - Searched CASE
- In Simple CASE We Have To Specify A SELECTOR, Which Determines Which Group of Action To Get Executed.
- In Searched Case, The Selector is Not Present, It Has Series of Search Conditions, That Are Evaluated in Order To Determine Which Group of Actions To Take Place.

Simple CASE Syntax

```
CASE SELECTOR
WHEN Expr1 THEN
Statement1;
WHEN Expr2 THEN
Statement2;
ELSE
Statement n;
END CASE;
```

- The Reserved Word "CASE" Marks The Beginning of The CASE Statement.
- The Selector is A Value That Determines, Which "WHEN" Clause Should Be Executed.
- Each When Clause Contains Ac Expression And One OR More Executable Statements Associated With It.
- The "ELSE" Clause is Optional.
- Each "CASE" Statement is Marked With "END CASE;".
- The Selector is Evaluated Only Once For The Whole Selection Process.
- Simple "CASE' Internally Evaluates The Selector Using The Equality Operator, Directly Upon The Process And Compares With The "Expression".
- Simple "CASE" is Not Suitable for Multi Conditional Analysis And Other Than The Equality Operator.

```
SQL> DECLARE
        V_Grade CHAR := UPPER('&EnterGrade');
        BEGIN
        CASE V_Grade
          WHEN 'A' THEN
          DBMS_OUTPUT.PUT_LINE('You are Awarded with Excellent Grade');
          WHEN 'B' THEN
          DBMS_OUTPUT.PUT_LINE('You are Awarded with Very Good Grade');
          WHEN 'C' THEN
          DBMS_OUTPUT.PUT_LINE('You are Awarded with Good Grade');
          WHEN 'D' THEN
          DBMS_OUTPUT.PUT_LINE('You are Awarded with Fair Grade');
          WHEN 'E' THEN
          DBMS_OUTPUT.PUT_LINE('You are Awarded with Poor Grade');
          ELSE
          DBMS_OUTPUT.PUT_LINE('Sorry!...No Such Grade is Existing');
        END CASE;
        END;
        /

SQL> DECLARE
        V_operator VARCHAR2(2) := '&Operator';
        V_Number1 NUMBER := &Operand1;
```

```
V_Number2 NUMBER := &Operand2;
BEGIN
CASE V_Operator
 WHEN '+' THEN
        DBMS_OUTPUT.PUT_LINE('The Sum of '||V_Number1||' and '||V_Number2||' is
        :'||TO_NUMBER(V_Number1 + V_Number2));
       WHEN '-' THEN
        DBMS_OUTPUT.PUT_LINE('The Difference of '||V_Number1||' and
        '||V_Number2||' is :'||TO_NUMBER(V_Number1 - V_Number2));
       WHEN '*' THEN
        DBMS_OUTPUT.PUT_LINE('The Product of '||V_Number1||' and '||V_Number2||'
        is :'||TO_NUMBER(V_Number1 * V_Number2));
       WHEN '/' THEN
        DBMS_OUTPUT.PUT_LINE('The Quotient of '||V_Number1||' and '||V_Number2||'
        is :'||TO_NUMBER(V_Number1 / V_Number2));
       WHEN '**' THEN
        DBMS_OUTPUT.PUT_LINE('The '||V_Number1||' to the Power of '||V_Number2||'
        is :'||TO_NUMBER(V_Number1 ** V_Number2));
       ELSE
        DBMS_OUTPUT.PUT_LINE('Invalid Operator... Please Check Your Self...!');
END CASE;
END;
/
```

Searched CASE

- A SEARCHED CASE Statement Has Search Conditions That Yield Boolean Values, "TURE", "FALSE" OR "NULL".
- When A Particular Search Condition Evaluates To TRUE, The Group of Statements Associated With This Condition Are Executed.
- The Searched CASE Can Evaluate Multiple Conditions Using The Range Evaluation Method.
- We can Evaluate Conditions Using The Normal Conditional Operators OR SQL*Plus Operators.

Searched CASE Syntax

```
CASE
WHEN SearchCondition1 THEN
        Statement1;
WHEN SearchCondition2 THEN
        Statement2;
ELSE
        StatementN;
END CASE;
```

- When A SEARCH CONDITION Evaluates To TRUE, Control is Passed To The Statement Associated With It.
- If No Search Condition Yields To TRUE, Then Statements Associated With "ELSE" Clause Are Executed.

```
SQL> DECLARE
        V_Grade CHAR := UPPER('&EnterGrade');
       BEGIN
       CASE
       WHEN V_Grade = 'A' THEN
                DBMS_OUTPUT.PUT_LINE('You are Awarded with Excellent Grade');
       WHEN V_Grade = 'B' THEN
                DBMS_OUTPUT.PUT_LINE('You are Awarded with Very Good Grade');
       WHEN V_Grade = 'C' THEN
```

```
                DBMS_OUTPUT.PUT_LINE('You are Awarded with Good Grade');
        WHEN V_Grade = 'D' THEN
                DBMS_OUTPUT.PUT_LINE('You are Awarded with Fair Grade');
        WHEN V_Grade = 'E' THEN
                DBMS_OUTPUT.PUT_LINE('You are Awarded with Poor Grade');
        ELSE
                DBMS_OUTPUT.PUT_LINE('Sorry!...No Such Grade is Existing');
        END CASE;
        END;
        /
SQL> DECLARE
        V_operator VARCHAR2(2) := '&Operator';
        V_Number1 NUMBER := &Operand1;
        V_Number2 NUMBER := &Operand2;
        BEGIN
        CASE
        WHEN V_Operator = '+' THEN
                DBMS_OUTPUT.PUT_LINE('The Sum of '||V_Number1||' and '||V_Number2||'
                is : '||TO_NUMBER(V_Number1 + V_Number2));
        WHEN V_Operator = '-' THEN
                DBMS_OUTPUT.PUT_LINE('The Difference of '||V_Number1||' and
                '||V_Number2||' is : '||TO_NUMBER(V_Number1 - V_Number2));
        WHEN V_Operator = '*' THEN
                DBMS_OUTPUT.PUT_LINE('The Product of '||V_Number1||' and
                '||V_Number2||' is : '||TO_NUMBER(V_Number1 * V_Number2));
        WHEN V_Operator = '/' THEN
                DBMS_OUTPUT.PUT_LINE('The Quotient of '||V_Number1||' and
                '||V_Number2||' is : '||TO_NUMBER(V_Number1 / V_Number2));
        WHEN V_Operator = '**' THEN
                DBMS_OUTPUT.PUT_LINE('The Power of '||V_Number1||' and '||V_Number2||'
                is : '||TO_NUMBER(V_Number1 ** V_Number2));
        ELSE
                DBMS_OUTPUT.PUT_LINE('Invalid Operator... Please Check Your Self...!');
        END CASE;
        END;
        /
```

CASE Expressions

- The Result of An Expression Yields A Single Value Which Can Be Assigned To A Variable At Run Time.
- A "CASE" Expression is An Expression Where The "CASE" Evaluates An Expression To A Single Value And Then The Result of The Expression is Assigned To A Variable.
- A "CASE" Expression Has A Structure Similar To A "CASE" Statement.
- "CASE" Expressions Reduce The Redundancy of The Code And Help in Result Reusability.

CASE Expression Syntax

```
ExprVar := CASE
                WHEN SearchCondition1 THEN
                        Statement1
                WHEN SearchCondition2 THEN
                        Statement2
                ELSE
                        StatementN
                END;
```

```
SQL> DECLARE
        V_Number NUMBER := &Number;
        V_Result VARCHAR2(30);
        BEGIN
        V_Result := CASE
                        WHEN MOD(V_Number,2) = 0 THEN
                            V_Number || ' is an Even Number'
                        ELSE
                            V_Number || ' is an Odd Number'
                        END;
       DBMS_OUTPUT.PUT_LINE(V_Result);
       DBMS_OUTPUT.PUT_LINE('Did You Get The Difference...!');
       END;
        /
SQL> DECLARE
        V_operator VARCHAR2(2) := '&Operator';
        V_Number1 NUMBER := &Operand1;
        V_Number2 NUMBER := &Operand2;
        V_Result VARCHAR2(300);
        BEGIN
        V_Result := CASE
                        WHEN V_Operator = '+' THEN
                            'The Sum of '||V_Number1||' and '||V_Number2||' is :
                            '||TO_NUMBER(V_Number1 + V_Number2)
                        WHEN V_Operator = '-' THEN
                            'The Difference of '||V_Number1||' and '||V_Number2||' is :
                            '||TO_NUMBER(V_Number1 - V_Number2)
                        WHEN V_Operator = '*' THEN
                            'The Product of '||V_Number1||' and '||V_Number2||' is :
                            '||TO_NUMBER(V_Number1 * V_Number2)
                        WHEN V_Operator = '/' THEN
                            'The Quotient of '||V_Number1||' and '||V_Number2||' is :
                            '||TO_NUMBER(V_Number1 / V_Number2)
                        WHEN V_Operator = '**' THEN
                            'The Power of '||V_Number1||' and '||V_Number2||' is :
                            '||TO_NUMBER(V_Number1 ** V_Number2)
                        END;
       DBMS_OUTPUT.PUT_LINE(V_Result);
       END;
       /
```

Unconditional Branching Using "GOTO"

- "GOTO" Provides The Ability To Jump Through A Program From One Place To Another.
- "GOTO" Can Be Used With Complex EXCEPTION Handlers in Nested Blocks, To Make The Execution Section More Readable.
- It is Better To Have A Limited Usage of "GOTO" In Programming Block.
- When A Jump is Proposed Using "GOTO" It is Associated With An Appropriate LABEL.

Syntax

GOTO LabelName;

Rules To Cross Check:

- "GOTO" Cannot Reference A LABEL in A Nested Block.
- "GOTO" Cannot Be Executed Outside An "IF" Clause To LABEL Inside The "IF" Clause.
- "GOTO" Cannot Be Executed From Inside An "IF" Clause To A LABEL Inside Another "IF" Cause.

- "GOTO" Cannot Navigate From The EXCEPTION Section To Any Other Section of The PL/SQL Block.

```
SQL> DECLARE
        V_MyNumber NUMBER := &GiveNumber;
        BEGIN
        IF MOD(V_MyNumber, 2) = 0 THEN
              GOTO L_EVEN;
        ELSE
              GOTO L_ODD;
        END IF;
        <<L_EVEN>>
        DBMS_OUTPUT.PUT_LINE('Even    Number.');
        RETURN;
        <<L_ODD>>
        DBMS_OUTPUT.PUT_LINE('Odd    Number.');
        RETURN;
        END;
        /
```

- "GOTO" Always Needs A Proper LABEL To Jump.
- A LABEL Does Not Need Any "GOTO" For Execution.
- Hence To Keeps Proper Meaning Within The Sequence "RETURN" Should Be Used.

Iterations In PL/SQL

- Loops Facilitate To Repeat A Statement OR Sequence of Statements Multiple Times.
- The Different Types of PL/SQL Loops Are
  - Basic OR Simple LOOP.
  - WHILE Loop.
  - FOR Loop.

"BASIC" OR "SIMPLE LOOP"

- It is The Simplest Form of The LOOP Construct In PL/SQL.
- It Encloses A Sequence of Statements Between The Keywords "LOOP" And "END LOOP".
- It Allows Execution of Its Statements At Lease Once.
- To Keep The Loop In Finite State The "EXIT" Statement is Used.

Syntax

```
LOOP
        Statements;
        EXIT [When Condition];
END LOOP;
```

"EXIT" Statement

- "EXIT" Statement is Used To Terminate A LOOP.
- Once The Loop is Terminated, The Control Passes To The Next Statement After The "END LOOP".
- "EXIT" Can Be Issued Either As An Action Within An "IF" Statement OR As A Stand Alone Statement With in The "LOOP".
- The "EXIT" Statement Should Always Be Placed Inside The Loop Only.
- "EXIT" Can Be Associated With A "WHEN" Clause To Allow Conditional Termination of The Loop.
- A Basic Loop Can Contain Multiple "EXIT" Statements.
- The "EXIT" Condition Can Be At The Top of The Loop OR At The End of The Loop As Per Logical Convenience.
- Depending Upon The Circumstances We Can Make Use of This LOOP As Pre-Tested Loop OR Post-Tested Loop Construct.

- But Under Major Construct it is Convenient To Use This Loop As Post Tested Loop Construct Only.
- The Loop Terminates Its Process When The Conditional State is "TRUE".

```
SQL> DECLARE
        V_Num NUMBER := 1;
        BEGIN
                LOOP
                DBMS_OUTPUT.PUT_LINE ('The Line ' || V_Num || ' Output is ' || V_Num);
                V_Num := V_Num + 1;
                        IF V_NUM > 5 THEN
                        EXIT;
                        END IF;
                END LOOP;
                DBMS_OUTPUT.PUT_LINE ('The Total Lines of Outputs are ' || (V_Num - 1));
        END;
        /
```

Nested Loops

- It is A Situation Where One Loop is Embedded Into The Other.
- The Outer Loop And The Inner Loop Get Associated With One Another And Execute Simultaneously.
- The Overall Loop Terminates is Dictated By The Outer Loop's "EXIT WHEN" Condition OR "EXIT" Condition.
- In Nested Loop's The Outer Loops Condition Evaluated As TRUE, Always Makes The Inner Loop To Resume Its Process And The Inner Loop's Termination Actually Makes The Outer Loop To Update its Process.

```
SQL> DECLARE
        V_Num  NUMBER := 1;
        BEGIN
                LOOP
                EXIT WHEN V_Num  > 10;
                        LOOP
                        EXIT WHEN V_Num > 5;
                        DBMS_OUTPUT.PUT_LINE('Inner Loop : ' ||   V_Num);
                        V_Num  := V_Num  + 1;
                        END LOOP;
                DBMS_OUTPUT.PUT_LINE('Outer Loop : ' ||   V_Num);
                V_Num  := V_Num  + 1;
                END LOOP;
        END;
        /
```

Nested Loops and Labels:

- Loops Can Be Nested To Multiple Levels.
- All The Loops Can Be Nested Into One Another.
- Loops Can Be Labeled As Per The Requirements.
- A Label is Placed Before The Statement, Either On The Same Line OR On A Separate Line.
- Label Loops By Placing The Label Before The Word Loop Within The Label Delimiters.
- When The Loop is Labeled, The Label Name Can Be Optionally Included After The END LOOP Statement For Clarity.

```
SQL> DECLARE
        V_Num  NUMBER := 1;
        BEGIN
                <<OuterLoop>>
```

```
        LOOP
                <<InnerLoop>>
                LOOP
                EXIT WHEN V_Num > 5;
                DBMS_OUTPUT.PUT_LINE('Inner Loop : ' || V_Num);
                V_Num := V_Num + 1;
                END LOOP InnerLoop;
        DBMS_OUTPUT.PUT_LINE('Outer Loop : ' || V_Num);
        V_Num := V_Num + 1;
        EXIT WHEN V_Num > 10;
        END LOOP OuterLoop;
END;
/
```

"WHILE" Loop

- It Can Be Used To Repeat A Sequence of Statements Until The Controlling Condition is No Longer "TRUE".
- The Condition is Evaluated At The Start of Each Iteration.
- The Loop Terminates When The Condition is "FALSE".

Syntax

```
WHILE Condition
LOOP
    Statement1;
    Statement2;
END LOOP;
```

- If The Condition Yields NULL, The LOOP is Bypassed And Control Passes To The Next Statement.

```
SQL> DECLARE
        V_Number NUMBER(2) := 1;
        V_Output VARCHAR2(100);
        BEGIN
                WHILE V_Number <= 5
                LOOP
                        V_Output := V_Output ||' '||V_Number;
                        V_Number := V_Number + 1;
                END LOOP;
        DBMS_OUTPUT.PUT_LINE(V_Output);
        END;
       /

SQL> DECLARE
        V_Num NUMBER;
        V_Position NUMBER;
        V_Result NUMBER := 0;
        BEGIN
        V_Num := &NumberToReverse;
                WHILE V_Num > 0
                LOOP
                        V_Position := MOD(V_Num, 10);
                        V_Result := (V_Result * 10) + V_Position;
                        V_Num := TRUNC(V_Num / 10);
                END LOOP;
        DBMS_OUTPUT.PUT_LINE('The Reverse of the Entered Number is : ');
        DBMS_OUTPUT.PUT_LINE(V_Result);
        END;
```

FOR Loop

- It Has The Same General Structure As The Basic Loop.
- "FOR LOOP" Contains A Control Statement At The Front of The LOOP Keyword, To Determine The Number of Iterations That PL/SQL Performs.

Syntax:

```
FOR Counter IN [REVERSE] LowerBound..UpperBound
LOOP
Statement1;
Statement2;
END LOOP;
```

Counter

- It Is An Implicitly Declared INTEGER Whose Value is Automatically Increased OR Decreased By 1 On Each Iteration of The LOOP Until The Upper Bound OR Lower Bound is Reached.

Reverse

- It Is A Keyword, And Causes The Counter To Decrement With Each Iteration From The Upper Bound To The Lower Bound.
- The Counter Need Not Be Declared, As It Is Implicitly Declared As An Integer.
- The Lower Bound And The Upper Bound of The Loop Can Be Literals, Variables, OR Expressions, But They Should Be Evaluated To Integers.
- The Lower And Upper Bounds of A Loop Statement Need Not Be Numeric Literals, They Can Be Expressions That Covert To Numeric Values At Run Time.
- FOR Loops Should Be Planned in Such Circumstances When We Have To Implement An Operation Upon A Range Analysis.
- For Loops Take Their Range Itself As The Conditional Statement, Hence They Are More Suitable When Working On Database Oriented Logic.

```
SQL> DECLARE
        V_StartRange NUMBER := &StartRange;
        V_EndRange NUMBER := &EndRange;
        V_Result VARCHAR2(500) := NULL;
        BEGIN
        DBMS_OUTPUT.PUT_LINE('Illustration of For Loop...!');
                FOR MyIndex  IN V_StartRange..V_EndRange
                LOOP
                        V_Result := V_Result||' '||V_StartRange;
                        V_StartRange := V_StartRange + 1;
                END LOOP;
        DBMS_OUTPUT.PUT_LINE(V_Result);
        END;
        /

SQL> DECLARE
        V_NumFact NUMBER := &GiveNumber;
        V_Factorial NUMBER := 1;
        BEGIN
                FOR IndexI IN 1..V_NumFact
                LOOP
                        V_Factorial := V_Factorial * IndexI;
                END LOOP;
        DBMS_OUTPUT.PUT_LINE(' The Factorial of '||V_NumFact|| ' is  : '||V_Factorial);
        END;
        /
SQL> DECLARE
        V_NumFact NUMBER := &GiveNumber;
        V_Factorial NUMBER := 1;
```

```
        BEGIN
              FOR IndexI IN REVERSE 1..V_NumFact
              LOOP
                    V_Factorial := V_Factorial * IndexI;
                     DBMS_OUTPUT.PUT_LINE(' The IndexI Number is '||IndexI);
              END LOOP;
        DBMS_OUTPUT.PUT_LINE(' The Factorial of '||V_NumFact|| ' is  : '||V_Factorial);
        END;
        /

SQL> DECLARE
        V_Name VARCHAR2(30);
        V_Position VARCHAR2(100) := NULL;
        BEGIN
        V_Name := '&EnterYourName' ;
              FOR  I IN 1..LENGTH(V_Name)
              LOOP
              V_Position := V_Position || ' ' ||SUBSTR(V_Name, I, 1);
              END LOOP;
        DBMS_OUTPUT.PUT_LINE(V_Position);
        END;
        /
```

Illustrations on Nested FOR Loops

```
SQL> DECLARE
        V_OuterLoopRange NUMBER := 5;
        V_InnerLoopRange NUMBER := 3;
        BEGIN
        <<OuterLoop>>
        FOR MyIndex IN 1..V_OuterLoopRange
        LOOP
        DBMS_OUTPUT.PUT_LINE('When Outer Loop Index Value = '||MyIndex);
              <<InnerLoop>>
              FOR MyIndex2 IN 1..V_InnerLoopRange
              LOOP
              DBMS_OUTPUT.PUT_LINE('Outer Loop Index Value ='||OUTERLOOP.MyIndex||'
              , '||'Inner Loop Index Value = '||MyIndex2);
              END LOOP InnerLoop;
        END LOOP OuterLoop;
        END;
        /
SQL> DECLARE
        V_MyChar VARCHAR2(20);
        BEGIN
        <<OuterLoop>>
        FOR MyIndex1 IN 1..15
        LOOP
              <<InnerLoop>>
              FOR MyIndex2 IN 1..MyIndex1
              LOOP
              V_MyChar := V_MyChar|| '*';
              END LOOP InnerLoop;
        DBMS_OUTPUT.PUT_LINE(V_MyChar);
        V_MyChar := NULL;
        END LOOP OuterLoop;
        END;
```

Using SQL Within PL/SQL

- All Most Everything We Learned in SQL Can Be Implemented in PL/SQL Using Proper Control And Memory Management.
- All SQL Functions Can Be Used Within The Same Syntax As That of The SQL.
- The Data Manipulation Statements From SQL Can Be Used More Intelligently in PL/SQL As We Have The Liberty of Controlling The Transactions And Data Manipulations Using The Control Structures.
- We Can Even Control The Transactions Using The "COMMIT" And "ROLLBACK" Statements From PL/SQL.

Data Retrieval Standards

- Data Retrieval Options Range From Basic SELECT Statements To Pattern Matching With Regular Expressions.
- The Data Retrieval Standards Are Implemented Using The "SELECT" Statements With Almost The Same Syntax of SQL.
- The "SELECT" Statement in PL/SQL is Restricted To Select Only One Record At A Time From The Database.
- If More Records Have To Be Retrieved And Managed By The PL/SQL We Have To Implement The Concepts of "CURSORS"
- All The Data Retrieved From The Database Server Has To Be Transferred into The Local PL/SQL Variables Before Processing The Business Logic.

"SELECT" Statement in PL/SQL

Syntax

```
SELECT  SelectList [INTO VariableList] FROM TableList [WHERE WHERE Clause]
        [GROUP BY Clause] [HAVING Clause] [ORDER BY ColumnList]
```

- The "SELECT LIST" Can Be Columns, Strings, Built-In Functions OR Projection Operator "*" To Retrieve Data.
- Arithmetic Operations Are Allowed In The "SELECT LIST".
- Variables Can Be Declared As A
  - Single Data Type (Scalar Data Types).
  - Anchor Data Type.
  - Entire Record Types And
  - PL/SQL Tables.
- The "TABLE LIST" in The "FROM" Clause Can Be One OR More "TABLES", "VIEWS" OR "INLINE VIEWS".
- The "WHERE" Clause Restricts The Result Set As Per The Condition Defined.
- When Selecting A Value Into A Variable, Be Sure To Return One And Only One Value into The Variable.
- Only One Record Can Be Returned Through A SELECT Statement In PL/SQL.
- The Common Errors Encountered Are:
  - "ORA – 01403" → NO Data Found
  - "ORA – 01422" → Exact Fetch Returns More Than Requested Number of Rows.

Note

- The SELECT Statements Should Be Restricted With Rules What We Discussed in Single Row Sub-Queries in SQL*Plus, To Avoid The ORA -01422 Exception.

```
SQL> DECLARE
        V_Empno     NUMBER := &EnterEmpno;
        V_EName     VARCHAR2(30);
        V_Job       VARCHAR2(30);
        V_Sal       NUMBER(7, 2);
        BEGIN
              SELECT Ename, Job, Sal  INTO  V_Ename, V_Job, V_Sal  FROM EMP
                                                    WHERE Empno = V_Empno;
```

```
        DBMS_OUTPUT.PUT_LINE('The Name : '||V_Ename);
        DBMS_OUTPUT.PUT_LINE('The job : '||V_Job);
        DBMS_OUTPUT.PUT_LINE('The Sal: '||V_Sal);
        END;
        /
SQL> DECLARE
        V_Empno     NUMBER(4):=&EnterEmpno;
        V_EName     VARCHAR2(30);
        V_Sal       NUMBER(7,2);
        V_Comm              NUMBER(7,2);
        BEGIN
              SELECT  EName, Sal, Comm  INTO V_EName, V_Sal, V_Comm
                                        FROM EMP WHERE Empno = V_Empno;
        DBMS_OUTPUT.PUT_LINE('The Name : '||INITCAP(V_Ename));
        DBMS_OUTPUT.PUT_LINE('The Salary : '||TO_CHAR(V_Sal, '99,999.99'));
        DBMS_OUTPUT.PUT_LINE('The Commission : '||NVL(TO_CHAR(V_Comm),
                                                        'No Commission'));
        DBMS_OUTPUT.PUT_LINE('The Total Sal is : '||TO_CHAR(V_Sal +
                                                    NVL(V_Comm, 0)));
        END;

SQL> DECLARE
        V_Empno             NUMBER(4)   :=&EnterEmpNo;
        V_Ename             VARCHAR2(30);
        V_Deptno            NUMBER(2);
        V_Job               VARCHAR2(30);
        V_MGR               NUMBER(4);
        V_HireDate          DATE;
        V_Sal               NUMBER(7,2);
        V_Comm              NUMBER(7,2);
        BEGIN
              SELECT  Ename, Deptno, Job, MGR, HireDate, Sal, Comm INTO
                    V_Ename, V_Deptno, V_Job, V_MGR, V_HireDate, V_Sal, V_Comm
                              FROM Emp  WHERE Empno = V_Empno;
        DBMS_OUTPUT.PUT_LINE('The Details of Employee Number '||V_Empno||' Requested
                                                            by You Are ...');
        DBMS_OUTPUT.PUT_LINE('The Employee Name :  '||V_Ename);
        DBMS_OUTPUT.PUT_LINE('The Department Number : '||V_Deptno);
        DBMS_OUTPUT.PUT_LINE('The Designation : '||V_Job);
        DBMS_OUTPUT.PUT_LINE('The Manager Number :  '||V_MGR);
        DBMS_OUTPUT.PUT_LINE('The Joining Date : '||V_HireDate);
        DBMS_OUTPUT.PUT_LINE('The Basic Salary : '||V_Sal);
        DBMS_OUTPUT.PUT_LINE('The Commission Earned : '||V_Comm);
        END;
        /

SQL> DECLARE
        V_Empno             NUMBER(4) := &EnterEmpNo;
        V_Ename             VARCHAR2(30);
        V_Deptno            NUMBER(2);
        V_Job               VARCHAR2(30);
        V_MGR               NUMBER(4);
        V_HireDate          DATE;
        V_Sal               NUMBER(7,2);
```

```
V_Comm              NUMBER(7,2);
V_MGRName           VARCHAR2(30);
BEGIN
SELECT Ename, Deptno, Job, MGR, HireDate, Sal, Comm INTO
          V_Ename , V_Deptno, V_Job, V_MGR, V_HireDate, V_Sal, V_Comm
                    FROM Emp WHERE Empno = V_Empno;
IF V_MGR IS NOT NULL THEN
      SELECT Ename INTO V_MGRName   FROM Emp WHERE Empno = V_MGR;
ELSE
      SELECT Ename INTO V_MGRName   FROM Emp WHERE MGR IS NULL;
END IF;
DBMS_OUTPUT.PUT_LINE('The Details of Employee Number '||V_Empno||' Requested
                                                              by You Are ...');
DBMS_OUTPUT.PUT_LINE('The Employee Name :  '||INITCAP(V_Ename));
DBMS_OUTPUT.PUT_LINE('The Department Number : '||V_Deptno);
DBMS_OUTPUT.PUT_LINE('The Designation : '||INITCAP(V_Job));
DBMS_OUTPUT.PUT_LINE('The Manager Number :
'||NVL(TO_CHAR(V_MGR),'"Cannot be Managed"')||', and his Name is : '||V_MGRName);
DBMS_OUTPUT.PUT_LINE('The Joining Date : '||TO_CHAR(V_HireDate,
                                                    'FMDD, Month Year'));
DBMS_OUTPUT.PUT_LINE('The Basic Salary : '||CONCAT('INR
',TO_CHAR(V_Sal,'99,999.99')));
DBMS_OUTPUT.PUT_LINE('The Commission Earned : '||NVL(TO_CHAR(V_Comm),'No
Commission...Sorry!'));
END;
/
```

Achieving Data Independency And Structural Independency in PL/SQL Using Anchor Type And Composite Type Variables

%TYPE Variable Declaration:

- The "%TYPE" Variable is Used To Anchor PL/SQL Variable To The Database Type Columns Directly.
- This Methodology is More Suitable Where The Variable That is Declared in The Program is Mapping Directly To A Column in The Database Table.

Syntax

Variablename TableName.ColumnName%TYPE;

- This Method Keeps The PL/SQL Program To Be Unaffected Even When The Data Types Within The Database Are Changing.
- The Methodology Provides Data Independency For The PL/SQL Program.
- Using This Methodology We Can Achieve Architectural Independency Between The Oracle Server Environment And The Client Environment.
- This Declaration Eliminates The "bit" Architectural Differences That Arise Between The Server And The Client.
- By Any Means if The Data Type of A Particular Column is Changing in The Back End Process of The Server, Either With Data Type OR Width, Will Never Effect The PL/SQL Modules With Their Operations.

Illustration

```
SQL> DECLARE
        V_EmpNo     Emp.Empno%TYPE;
        V_Ename     Emp.Ename%TYPE;
        V_Sal       Emp.Sal%TYPE;
        BEGIN
        /*Executable Statements*/
        END;
```

- It is Always Better To Declare The Variables That Are Acting As Communication Agents Between SQL And PL/SQL Environments With This Notation.
- All Remaining Variables Can Be Declared As Scalar Types.

```
SQL> DECLARE
        V_Empno     Emp.Empno%TYPE := &EnterEmpno;
        V_EName     Emp.EName%TYPE;
        V_Job       Emp.Job%TYPE;
        V_Sal       Emp.Sal%TYPE;
        BEGIN
            SELECT  Ename, Job, Sal INTO V_Ename, V_Job, V_Sal FROM EMP
                                                    WHERE Empno = V_Empno;
        DBMS_OUTPUT.PUT_LINE('The Name : '||V_Ename);
        DBMS_OUTPUT.PUT_LINE('The Job : '||V_Job);
        DBMS_OUTPUT.PUT_LINE('The Sal : '||V_Sal);
        END;
        /

SQL> DECLARE
        V_Empno         Emp.Empno%TYPE := &EnterEmpNo;
        V_Ename         Emp.Ename%TYPE;
        V_Deptno        Emp.Deptno%TYPE;
        V_Job           Emp.Job%TYPE;
        V_MGR           Emp.MGR%TYPE;
        V_HireDate      Emp.HireDate%TYPE;
        V_Sal           Emp.Sal%TYPE;
        V_Comm          Emp.Comm%TYPE;
        V_MGRName       Emp.Ename%TYPE;
```

```
BEGIN
      SELECT Ename, Deptno, Job, MGR, HireDate, Sal, Comm INTO V_Ename ,
                  V_Deptno, V_Job, V_MGR, V_HireDate, V_Sal, V_Comm
                        FROM Emp WHERE Empno = V_Empno;
      IF V_MGR IS NOT NULL THEN
         SELECT Ename INTO V_MGRName FROM Emp WHERE  Empno = V_MGR;
      ELSE
        SELECT Ename INTO V_MGRName FROM Emp WHERE  MGR IS NULL;
      END IF;
DBMS_OUTPUT.PUT_LINE('The Details of Employee Number  '||V_Empno||' Requested
                                                          by You Are ...');
DBMS_OUTPUT.PUT_LINE('The Employee Name :  '||INITCAP(V_Ename));
DBMS_OUTPUT.PUT_LINE('The Department Number : '||V_Deptno);
DBMS_OUTPUT.PUT_LINE('The Designation : '||INITCAP(V_Job));
DBMS_OUTPUT.PUT_LINE('The Manager Number :  '||NVL(TO_CHAR(V_MGR),
                  '"Cannot be Managed"')||' , and his Name is : '||V_MGRName);
DBMS_OUTPUT.PUT_LINE('The Joining Date : '||TO_CHAR(V_HireDate,'FMDD,
                                                          Month Year'));
DBMS_OUTPUT.PUT_LINE('The Basic Salary : '||CONCAT('INR
                                                ',TO_CHAR(V_Sal,'99,999.99')));
DBMS_OUTPUT.PUT_LINE('The Commission Earned : '||NVL(TO_CHAR(V_Comm),
                                                'No Commission...Sorry!'));
END;
/
```

Applying Composite Data Types in PL/SQL

- The Composite Data Types in PL/SQL Are Also Called As "COLLECTIONS".
- The Types of Composite Data Types in PL/SQL Are
  - %ROWTYPE.
  - PL/SQL RECORD Type.
  - PL/SQL TABLE Type.
  - NESTED TABLE.
  - VARRAY.

%ROWTYPE Declaration

- To Declare A Record Based On A Collection of Columns in A Database Table OR View, We Can Use The %ROWTYPE Attribute.
- The Fields in The Record Take Their Names And Data Types From The Columns of The Table OR View.
- The Record Can Also Store An Entire Row of Data Fetched From "CURSOR" Variable.
- Prefix "%ROWTYPE" With The Database Table Name.

Syntax

VariableName TableName%ROWTYPE;

Illustration

V_EmpRecord Emp%ROWTYPE;

Advantages

- The Number of Columns And Data Types of The Underlying Database Columns Need Not Be Known.
- The Number of Columns And Data Types of The Underlying Database Columns May Change At Runtime.
- It is Mostly Useful When Retrieving The Data For The Entire Row With The "SELECT" Statement Using Projection "*" Operator.
- The "%ROWTYPE" Declaration Provides The Structural Independency When Designing The Applications.

```
SQL> DECLARE
        V_EmpRecord Emp%ROWTYPE;
        BEGIN
              SELECT  * INTO V_EmpRecord FROM EMP WHERE Empno = &EnterEmpno;
        DBMS_OUTPUT.PUT_LINE('The Name : '|| V_EmpRecord.Ename);
        DBMS_OUTPUT.PUT_LINE('The Job : '|| V_EmpRecord.Job);
        DBMS_OUTPUT.PUT_LINE('The Salary : '||V_EmpRecord.Sal);
        DBMS_OUTPUT.PUT_LINE('The Commission : '|| V_EmpRecord.Comm);
        DBMS_OUTPUT.PUT_LINE('The Gross Salary : ' ||(V_EmpRecord.Sal +
                                                    NVL(V_EmpRecord.Comm, 0)));
        END;
        /

SQL> DECLARE
        EmpRecord Emp%ROWTYPE;
        V_EmpRecord EmpRecord%TYPE;
        BEGIN
              SELECT  * INTO V_EmpRecord FROM EMP WHERE Empno = &EnterEmpno;
        DBMS_OUTPUT.PUT_LINE('The Name : '|| V_EmpRecord.Ename);
        DBMS_OUTPUT.PUT_LINE('The Job : '|| V_EmpRecord.Job);
        DBMS_OUTPUT.PUT_LINE('The Salary : '||V_EmpRecord.Sal);
        DBMS_OUTPUT.PUT_LINE('The Commission : '|| V_EmpRecord.Comm);
        DBMS_OUTPUT.PUT_LINE('The Gross Salary : ' ||(V_EmpRecord.Sal +
                                                    NVL(V_EmpRecord.Comm, 0)));
        END;
        /

SQL> DECLARE
        V_Empno Emp.Empno%TYPE := &EnterEmpNo;
        V_EmpRecord Emp%ROWTYPE;
        V_DeptRecord Dept%ROWTYPE;
        V_SalGradeRecord SalGrade%ROWTYPE;
        V_MGRName Emp.Ename%TYPE;
        BEGIN
          SELECT * INTO V_EmpRecord FROM Emp WHERE Empno = V_Empno;
          SELECT * INTO V_DeptRecord FROM Dept WHERE Deptno = V_EmpRecord.Deptno;
          SELECT * INTO V_SalGradeRecord FROM SalGrade WHERE V_EmpRecord.Sal
                                              BETWEEN LoSal AND HiSal;
           IF V_EmpRecord.MGR IS NOT NULL THEN
               SELECT Ename INTO V_MGRName FROM Emp WHERE
                                                    Empno = V_EmpRecord.MGR;
          ELSE
                SELECT Ename INTO V_MGRName FROM Emp WHERE MGR IS NULL;
           END IF;
        DBMS_OUTPUT.PUT_LINE('The Details of Employee Number '||V_EmpRecord.Empno||'
                                                    Requested by You Are ...');
        DBMS_OUTPUT.PUT_LINE('The Employee Name :  '||INITCAP(V_EmpRecord.Ename));
        DBMS_OUTPUT.PUT_LINE('The Department Number : '||V_EmpRecord.Deptno||',
        Department Name is '||INITCAP(V_DeptRecord.Dname)||', and is Situated at
                                                    '||INITCAP(V_DeptRecord.Loc));
        DBMS_OUTPUT.PUT_LINE('The Designation : '||INITCAP(V_EmpRecord.Job));
        DBMS_OUTPUT.PUT_LINE('The Manager Number :
              '||NVL(TO_CHAR(V_EmpRecord.MGR),'"Cannot be Managed"')||', and his
                                              Name is : '||V_MGRName);
        DBMS_OUTPUT.PUT_LINE('The Joining Date :
        '||TO_CHAR(V_EmpRecord.HireDate,'FMDD, Month Year')||' You Are Serving Us Since :
```

```
                    '||TRUNC((SYSDATE - V_EmpRecord.HireDate)/365)||' Years.');
DBMS_OUTPUT.PUT_LINE('The Basic Salary : '||CONCAT('INR
',TO_CHAR(V_EmpRecord.Sal,'99,999.99'))||' and his Grade as Per Salary is
                    '||V_SalGradeRecord.Grade);
DBMS_OUTPUT.PUT_LINE('The Commission Earned : '||CONCAT('INR ',
      NVL(TO_CHAR(V_EmpRecord.Comm, '99G999D99'),'No Commission...Sorry!')));
DBMS_OUTPUT.PUT_LINE('The Gross Salary Earned : '|| CONCAT('INR ',
      TO_CHAR((V_EmpRecord.Sal + NVL(V_EmpRecord.Comm,0)), '99G999D99')));
END;
/
```

PL/SQL Records

- A "PL/SQL RECORD" is A Group of Related Data Items Stored in Individual Fields, Each With Its Own Attribute Name And Data Type.
- When A "RECORD TYPE" of Fields Are Declared Then They Can Be Manipulated As A Unit Through Out The Application.

The Points To Note Are

- Each "RECORD" Defined Can Have As Many Fields As Necessary.
- "RECORDS" Can Be Assigned Initial Values And Can Be Defined As "NOT NULL".
- Fields Without Initial Values Are Initialized To "NULL".
- The "DEFAULT" Key Word Can Also Be Used When Defining Fields.
- "RECORD" Types Can Be Declared As User Define Records In The Declarative Part of Any PL/SQL Block.
- A "RECORD" Can Be The Component of Another "RECORD".
- We Can "DECLARE" And Reference Nested Records in The Program As Per Our Convenience And Analysis.
- After The Record Data Type is Created It Should Be Instantiated, Before The Operational Process Can Be Implemented.

Syntax

```
TYPE TypeName IS
RECORD
( FieldName1, FieldName2, FieldNameN );
```

- The FieldName Can Have The Following Syntax

```
FieldName { FieldDataType(Width) OR
                    RecordVariable%TYPE OR
                    Table.Column%TYPE OR
                    Table%ROWTYPE }
                    [NOT NULL]
                    [:=[DEFAULT] Expr]
```

Defining And Declaring A PL/SQL Record

- To Create A User Defined "RECORD" Data Type, We Define A "RECORD TYPE" And Then Declare Records of That Type.
  - Type Name → is The Name Of The Record Type.
  - Field Name → It is The Name of The Field Within The Record.
  - Field Type → Is The Data Type of The Field.
  - Expr → It is The Field Type OR An Initial Value.
- The "NOT NULL" Constraint Prevents The Assigning of NULL's To Those Fields.
- Field Declarations Are Like Variable Declarations Each Field Has A Unique Name And A Specific Data Type.
- The Most Important Point To Note is We Must Create The Data Type First And Then Declare An Identifier Using The Declared Data Type.

Illustration

```
DECLARE
TYPE EmpRecordType IS RECORD
( Empno     NUMBER(4) NOT NULL := 0,
  Ename     Emp.Ename%TYPE,
  Job  Emp.Job%TYPE );
```

- After The "RECORD" Type Has Been Created, We Should Instantiate The Individual "RECORD" Type Variables For Operations.

Illustration

```
EmpRecordInstance          EmpRecordDataType;
```

- Fields in a "RECORD" Are Accessed By Their Name.
- To Reference OR Initialize An Individual Field, We Must Use Dot Notation.

Syntax

```
RecordName.FieldName
```

Illustration

```
EmpRecord.Ename := 'SAMPATH';
SQL> DECLARE
        V_Empno NUMBER(4) := &EnterEmpNo;
        TYPE EmpRecordType IS RECORD
        ( V_Ename VARCHAR2(30), V_Deptno NUMBER(2), V_Job  VARCHAR2(30));
        EmpRecord    EmpRecordType;
        BEGIN
        SELECT Ename, Deptno, Job INTO EmpRecord.V_Ename, EmpRecord.V_Deptno,
                    EmpRecord.V_Job FROM Emp WHERE Empno = V_Empno;
        DBMS_OUTPUT.PUT_LINE('The Details of Employee Number '||V_Empno||' Requested
                                                              by You Are ...');
        DBMS_OUTPUT.PUT_LINE('The Employee Name :  '||EmpRecord.V_Ename);
        DBMS_OUTPUT.PUT_LINE('The Department Number : '||EmpRecord.V_Deptno);
        DBMS_OUTPUT.PUT_LINE('The Designation : '||EmpRecord.V_Job);
        END;
        /
SQL> DECLARE
        V_Empno NUMBER(4) := &EnterEmpNo;
        TYPE EmpRecordType IS RECORD
        ( Ename VARCHAR2(30), Deptno NUMBER(2), Job VARCHAR2(30),
          MGR NUMBER(4), HireDate DATE, Sal NUMBER(7,2), Comm NUMBER(7,2));
        EmpRecord EmpRecordType;
        BEGIN
        SELECT Ename, Deptno, Job, MGR, HireDate, Sal, Comm INTO
              EmpRecord.Ename , EmpRecord.Deptno, EmpRecord.Job, EmpRecord.MGR,
                    EmpRecord.HireDate, EmpRecord.Sal, EmpRecord.Comm
                          FROM Emp WHERE Empno = V_Empno;
        DBMS_OUTPUT.PUT_LINE('The Details of Employee Number '||V_Empno||' Requested
                                                              by You Are ...');
        DBMS_OUTPUT.PUT_LINE('The Employee Name :  '||EmpRecord.Ename);
        DBMS_OUTPUT.PUT_LINE('The Department Number : '||EmpRecord.Deptno);
        DBMS_OUTPUT.PUT_LINE('The Designation : '||EmpRecord.Job);
        DBMS_OUTPUT.PUT_LINE('The Manager Number :  '||EmpRecord.MGR);
        DBMS_OUTPUT.PUT_LINE('The Joining Date : '||EmpRecord.HireDate);
        DBMS_OUTPUT.PUT_LINE('The Basic Salary : '||EmpRecord.Sal);
        DBMS_OUTPUT.PUT_LINE('The Commission Earned : '||EmpRecord.Comm);
        END;
```

Illustration1

- Atomic Type Collection

```
TYPE AtomicTypesRec IS RECORD(
    Empno Emp.Empno%TYPE, MgrName Emp.Ename%TYPE);
```

- Nested Type Collection

```
TYPE AllTablesRecord IS RECORD(
        EmpRecord  Emp%ROWTYPE, DeptRecord  Dept%ROWTYPE,
                SalGrRecord   SalGrade%ROWTYPE,  AtomicType    AtomicTypesRec);
```

- Type Instantiation

```
V_AllTablesRecord AllTablesRecord;
```

Illustration2

- Employee Type Creation

```
TYPE EmpRecordType IS RECORD(
 Ename  Emp.Ename%TYPE, Deptno Emp.Deptno%TYPE, Job  Emp.Job%TYPE,
        MGR Emp.MGR%TYPE, HireDate Emp.HireDate%TYPE, Sal Emp.Sal%TYPE,
                                                    Comm Emp.Comm%TYPE);
```

- Department Type Creation

```
TYPE DeptRecordType IS RECORD(
        Deptno Dept.Deptno%TYPE, Dname Dept.Deptno%TYPE, Loc Dept.Deptno%TYPE);
```

- Sal Grade Type Creation

```
TYPE SalGradeRecordType IS RECORD(
        Grade   SalGrade.Grade%TYPE, LoSal  SalGrade.LoSal%TYPE,
                                                HiSal SalGrade.HiSal%TYPE);
```

- Integrating All Types As One Collection

```
TYPE AllTablesRecord IS RECORD(
        EmpRecord EmpRecordType, DeptRecord DeptRecordType, SalGradeRecord
                                        SalGradeRecordType);
```

- Instantiating The Final Collection For Usage

```
V_AllTablesRecord AllTablesRecord;
```

Illustration3

- Employee Type Creation

```
TYPE EmpRecordType IS RECORD(
        Ename Emp.Ename%TYPE, Deptno Emp.Deptno%TYPE,  Job  Emp.Job%TYPE,
                MGR Emp.MGR%TYPE, HireDate  Emp.HireDate%TYPE,
                        Sal Emp.Sal%TYPE,Comm Emp.Comm%TYPE);
```

- Department Type Creation

```
TYPE DeptRecordType IS RECORD(
        Deptno  Dept.Deptno%TYPE, Dname Dept.Deptno%TYPE, Loc Dept.Deptno%TYPE,
                                                EmpRecord EmpRecordType,);
```

- Sal Grade Type Creation

```
TYPE SalGradeRecordType IS RECORD(
        Grade  SalGrade.Grade%TYPE, LoSal  SalGrade.LoSal%TYPE,
                        HiSal    SalGrade.HiSal%TYPE, DeptRecord DeptRecordType);
```

- Instantiating The Final Collection For Usage

```
V_AllTablesRecord SalGradeRecordType
```

Illustration4

- Employee Type Creation

```
TYPE EmpRecordType IS RECORD( EmpRecord Emp%ROWTYPE);
```

- Department Type Creation

```
TYPE DeptRecordType IS RECORD
(
 DeptRecord     Dept%ROWTYPE,
 EmpRecord      EmpRecordType
);
```

- Sal Grade Type Creation

```
TYPE SalGradeRecordType IS RECORD(
        SalDrRecord  SalGrade%ROWTYPE, DeptRecord DeptRecordType);
```

Illustrative Example

```
SQL> DECLARE TYPE AtomicTypes IS RECORD /*Atomic Type Creation*/
        ( Empno Emp.Empno%TYPE,
          MgrName Emp.Ename%TYPE );
        TYPE AllTablesRecord IS RECORD /*Nested Type Creation*/
        ( EmpRecord Emp%ROWTYPE,
         DeptRecord Dept%ROWTYPE,
         SalGradeRecord SalGrade%ROWTYPE,
         AtomicType  AtomicTypes /*Type Instance in Another Type*/ );
        V_AllTablesRecord AllTablesRecord; /*Type Instantiation*/
        BEGIN
        V_AllTablesRecord.AtomicType.Empno := &EnterEmpno;
        SELECT  Ename, Deptno, Job, MGR, HireDate, Sal, Comm  INTO
              V_AllTablesRecord.Emprecord.Ename , V_AllTablesRecord.Emprecord.Deptno,
              V_AllTablesRecord.Emprecord.Job, V_AllTablesRecord.Emprecord.MGR,
              V_AllTablesRecord.Emprecord.HireDate, V_AllTablesRecord.Emprecord.Sal,
              V_AllTablesRecord.Emprecord.Comm FROM Emp
                    WHERE Empno = V_AllTablesRecord.AtomicType.Empno;
        SELECT Dname, Loc  INTO  V_AllTablesRecord.DeptRecord.Dname,
                    V_AllTablesRecord.DeptRecord.Loc FROM Dept
                          WHERE Deptno = V_AllTablesRecord.Emprecord.Deptno;
        SELECT Grade INTO V_AllTablesRecord.SalGradeRecord.Grade FROM SalGrade
              WHERE V_AllTablesRecord.Emprecord.Sal  BETWEEN LoSal AND HiSal;
        IF V_AllTablesRecord.Emprecord.MGR IS NOT NULL THEN
              SELECT  Ename INTO V_AllTablesRecord.AtomicType.MgrName FROM Emp
                                    WHERE Empno = V_AllTablesRecord.Emprecord.MGR;
        ELSE
              SELECT  Ename  INTO  V_AllTablesRecord.AtomicType.MgrNameFROM Emp
                                                      WHERE MGR IS NULL;
        END IF;
        DBMS_OUTPUT.PUT_LINE('The Details of Employee Number
                    '||V_AllTablesRecord.AtomicType.Empno||' Requested by You Are ...');
        DBMS_OUTPUT.PUT_LINE('The Employee Name :
                    '||V_AllTablesRecord.Emprecord.Ename);
        DBMS_OUTPUT.PUT_LINE('The Department Number :
                    '||V_AllTablesRecord.Emprecord.Deptno||', Department Name is
                    '||INITCAP(V_AllTablesRecord.DeptRecord.Dname)||', and is Situated at
                          '||INITCAP(V_AllTablesRecord.DeptRecord.Loc));
        DBMS_OUTPUT.PUT_LINE('The Designation : '||V_AllTablesRecord.Emprecord.Job);
        DBMS_OUTPUT.PUT_LINE('The Manager Number :
              '||NVL(TO_CHAR(V_AllTablesRecord.Emprecord.MGR),'"Cannot be Managed"')||',
                    and his Name is : '||V_AllTablesRecord.AtomicType.MgrName);
        DBMS_OUTPUT.PUT_LINE('The Joining Date :
                    '||V_AllTablesRecord.Emprecord.HireDate);
        DBMS_OUTPUT.PUT_LINE('The Basic Salary : '||CONCAT('INR
```

```
',TO_CHAR(V_AllTablesRecord.Emprecord.Sal,'99,999.99'))||' and his Grade as Per
        Salary is '||V_AllTablesRecord.SalGradeRecord.Grade);
DBMS_OUTPUT.PUT_LINE('The Commission Earned :
  '||NVL(TO_CHAR(V_AllTablesRecord.Emprecord.Comm),'No Commission...Sorry!'));
  END;
  /
```

PL/SQL Tables

- Objects of Type "TABLE" Are Called PL/SQL Tables.
- They Are Modeled As Database Tables, But Are Not Same.
- PL/SQL TABLES Use A "PRIMARY KEY" To Give Array Like Access To Rows.
- PL/SQL Tables Are Very Dynamic in Operation, Giving The Simulation To Pointers in 'C' Language.
- They Help in Integrating The Cursors For Dynamic Management of Records At Run Time.
- They Make Runtime Management of Result Sets Very Convenient.

Things To Note

- It is Similar To An Array in Third Generation Languages.
- PL/SQL Table Should Contain Two Components.
  - A "PRIMARY KEY" of Data Type BINARY_INTEGER, That Indexes The PL/SQL Table.
  - A Column of A Scalar OR Record Data Type Which Stores The PL/SQL Table Elements.
- PL/SQL Tables Can Increase in Size Dynamically As They Are Unconstrained.

Creating A PL/SQL Table

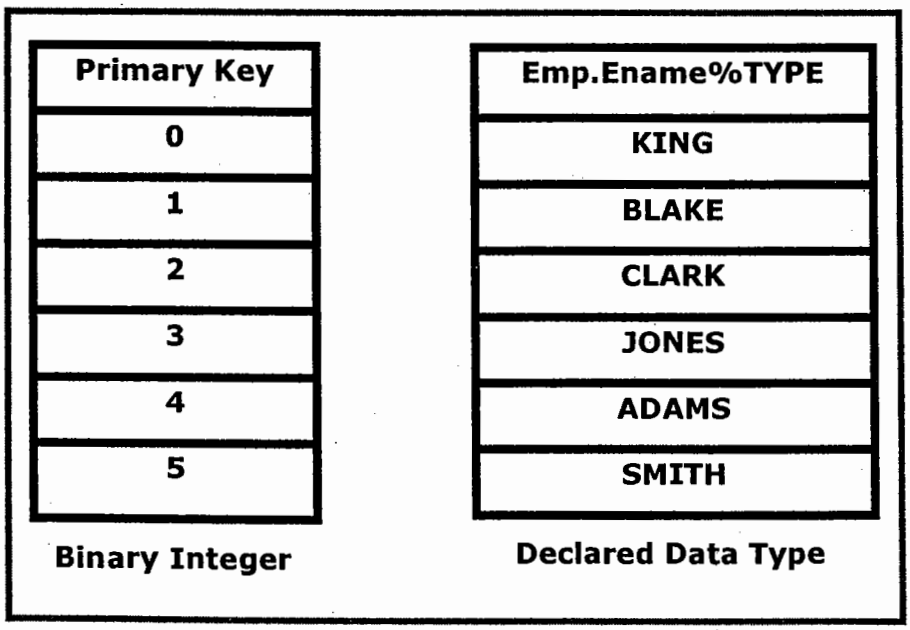
- There Are Two Steps Involved In Creating A PL/SQL Table.
  - Declare A PL/SQL Table Data Type.
  - Declare A Variable of That Data Type.

Syntax

```
TYPE TypeName IS  TABLE OF {Column Type OR Variable%TYPE OR Table.Column%TYPE
                OR Table%ROWTYPE} [NOT NULL] INDEX BY BINARY_INTEGER;
PLSQLTableInstance TypeName;
```

Illustration

```
TYPE EnameTableType  IS TABLE OF  Emp.Ename%TYPE INDEX BY BINARY_INTEGER;
EnameTable     EnameTableType;
```



The Number of Rows in A PL/SQL Table Can Increase Dynamically, Hence A PL/SQL Table Can Grow As New Rows Are Added.

- PL/SQL Tables Can Have One Column For data Storage And A Primary Key For Index Management, Neither of Which Can Be Named.
- The Data Column Can Belong To Any Scalar OR Record Data Type, But The PRIMARY KEY Must Belong To Type BINARY_INTEGER Only.
- We Cannot Initialize A PL/SQL Table In Its Declaration.

Referencing A PL/SQL Table

Syntax

PLSQL_TableName(Primary_Key_Value);

- PRIMARY_KEY_VALUE Belongs To Type BINARY_INTEGER.
- The Primary Key Value Can Be Negative Indexing Need Not Start With 1.
- The Methods That Make PL/SQL Tables Easier To Use Are
  - EXISTS(N)
    - Returns "TRUE" If The Nth Element In PL/SQL Table Exists.
  - COUNT
    - Returns The Number of Elements That A PL/SQL Table Currently Contains.
  - FIRST And LAST
    - Returns The FIRST And LAST Index Numbers in A PL/SQL Table.
    - Returns NULL If The PL/SQL Table is Empty.
  - PRIOR(N)
    - Returns The Index Number That Precedes Index N In A PL/SQL Table.
  - NEXT(N)
    - Returns The Index Number That Succeeds Index N In A PL/SQL Table.
  - EXTEND(N.I)
    - It Increases The Size Of A PL/SQL Table.
    - It Appends One NULL Element To A PL/SQL Table.
  - EXTEND(N)
    - Appends N NULL Elements Into A PL/SQL Table.
  - EXTEND(N, i)
    - Appends N Copies of The i^{th} Element To A PL/SQL Table.
  - TRIM
    - It Removes One Element From The End Of A PL/SQL Table.
  - TRIM(N)
    - Removes N Elements From The End of A PL/SQL Table.
  - DELETE
    - It Removes All Elements From A PL/SQL Table.
    - DELETE(N), Delete (M, N)
- To Reference PL/SQL Table of Records We Use The Notation On The Index Number of The Table.
  - TableName(Index).Field

```
SQL> DECLARE
        TYPE Dept_Table_Type  IS  TABLE OF  Dept.Dname%TYPE INDEX  BY
                                                    BINARY_INTEGER;
        My_Dept_Table  Dept_Table_Type;
        V_Count         NUMBER(2);
        BEGIN
        SELECT  COUNT(*) INTO V_Count FROM Dept;
        FOR  MyIndex  IN  1..V_Count  LOOP
              SELECT  Dname INTO  My_Dept_Table(MyIndex) FROM  Dept
                                          WHERE Deptno = MyIndex * 10;
        END  LOOP;
        FOR  MYIndex IN   1..V_Count  LOOP
              DBMS_OUTPUT.PUT_LINE(My_Dept_Table(MyIndex));
```

```
        END LOOP;
        END;
        /

SQL> DECLARE
        TYPE Dept_Table_Type IS TABLE OF Dept%ROWTYPE INDEX BY
                                                        BINARY_INTEGER;
        My_Dept_Table   Dept_Table_Type;
        V_Count     NUMBER(2);
        BEGIN
        SELECT COUNT(*) INTO V_Count FROM  Dept; FOR MyIndex IN 1..V_Count
        LOOP
            SELECT * INTO My_Dept_Table(MyIndex) FROM Dept
                                        WHERE  Deptno = MyIndex * 10;
        END  LOOP;
        FOR MyIndex IN 1..V_Count
        LOOP
              DBMS_OUTPUT.PUT_LINE('Dept. '|| My_Dept_Table(MyIndex).Deptno ||' '
                                    ||My_Dept_Table(MyIndex).Dname||' is Located in ' ||
              My_Dept_Table(MyIndex).Loc);
        END LOOP;
        END;
        /

SQL> DECLARE
        TYPE Emp_Table_Type IS TABLE OF Emp%ROWTYPE
        INDEX BY BINARY_INTEGER;
        My_Emp_Table Emp_Table_Type;
        V_RowID NUMBER(2) := 65;
        V_EmpCount NUMBER(2);
        BEGIN
        SELECT COUNT(*) INTO V_EmpCount FROM Emp;
        FOR MyIndex IN 1..V_EmpCount LOOP
              SELECT * INTO My_Emp_Table(MyIndex) FROM Emp
                          WHERE SUBSTR(ROWID,18,1) = CHR(V_RowID);
              V_RowID := V_RowID + 1;
        END LOOP;
        DBMS_OUTPUT.PUT_LINE(RPAD('Employees Information', 80));
        DBMS_OUTPUT.PUT_LINE(RPAD('-', 80,'-'));
        DBMS_OUTPUT.PUT_LINE(RPAD('EmpNo', 8)|| RPAD('Ename', 12)||RPAD('Job',
              12)||RPAD('Deptno', 8)||RPAD('Mgr', 10)||RPAD('Hiredate', 12)||RPAD('Sal',
                                                        12)||RPAD('Comm', 10));
        DBMS_OUTPUT.PUT_LINE(RPAD('-', 80,'-'));
        FOR MyIndex IN 1..V_EmpCount LOOP
              DBMS_OUTPUT.PUT_LINE(RPAD(My_Emp_Table(MyIndex).Empno, 8)
              ||RPAD(My_Emp_Table(MyIndex).Ename,
              12)||RPAD(My_Emp_Table(MyIndex).Job,
              12)||RPAD(My_Emp_Table(MyIndex).Deptno,
              8)||NVL(TO_CHAR(RPAD(My_Emp_Table(MyIndex).MGR, 10)),'No
              Manager')||RPAD(My_Emp_Table(MyIndex).Hiredate,
              12)||RPAD(My_Emp_Table(MyIndex).Sal,
              12)||NVL(TO_CHAR(RPAD(My_Emp_Table(MyIndex).Comm, 12)),'-NA-'));
        END LOOP;
        DBMS_OUTPUT.PUT_LINE(My_Emp_Table.COUNT||' Rows Selected.');
        END;
```

```
SQL> DECLARE
        TYPE Dept_Table_Type IS TABLE OF Dept.Dname%TYPE
        INDEX BY BINARY_INTEGER;
        My_Dept_Table Dept_Table_Type;
        V_Count       NUMBER(2);
        V_RecNo       NUMBER(2) := &EnterRecordNumber;
        BEGIN
        SELECT COUNT(*) INTO V_Count FROM Dept;
        FOR MyIndex IN 1..V_Count LOOP
              SELECT Dname INTO My_Dept_Table(MyIndex) FROM Dept
                                                  WHERE Deptno = MyIndex * 10;
        END LOOP;
        FOR MYIndex IN 1..V_Count LOOP
              DBMS_OUTPUT.PUT_LINE('The Department Number, '||MyIndex*10||' is named
              as : '||My_Dept_Table(MyIndex));
        END LOOP;
        DBMS_OUTPUT.PUT_LINE('The PLSQL Table Contains '||My_Dept_Table.COUNT||'
        Records.');
        DBMS_OUTPUT.PUT_LINE('The First Index Number of PLSQL Table is
        '||My_Dept_Table.FIRST||' and the First Record is :
        '||My_Dept_Table(My_Dept_Table.FIRST));
        DBMS_OUTPUT.PUT_LINE('The Last Index Number of PLSQL Table is
        '||My_Dept_Table.LAST||' and the Last Record is :
        '||My_Dept_Table(My_Dept_Table.LAST));
        DBMS_OUTPUT.PUT_LINE('The Previous Record of '||My_Dept_Table.LAST||'th Record
        of PLSQL Table is '||My_Dept_Table.PRIOR(My_Dept_Table.LAST)||' and the Data is : '||
        My_Dept_Table( My_Dept_Table.PRIOR( My_Dept_Table.LAST)));
        DBMS_OUTPUT.PUT_LINE('The Next Record of '||My_Dept_Table.FIRST||'st Record of
        PLSQL Table is '||My_Dept_Table.NEXT(My_Dept_Table.FIRST)||' and the Data is : '||
        My_Dept_Table( My_Dept_Table.NEXT( My_Dept_Table.FIRST)));
        IF My_Dept_Table.EXISTS(V_RecNo) THEN
              DBMS_OUTPUT.PUT_LINE('The Department Exists');
              DBMS_OUTPUT.PUT_LINE('The Department Name is,
              '||My_Dept_Table(V_RecNo));
        ELSE
              DBMS_OUTPUT.PUT_LINE('Sorry! ... Your Requested Data does not Exist...Check
              the Record Range.');
        END IF;
        END;
        /
```

Data Manipulation Through PL/SQL

- The Data in The Database is Manipulated Using The DML Commands From PL/SQL.
- The DML Commands INSERT, UPDATE And DELETE Can Be Used in PL/SQL Without Any Restrictions.
- The ROW LOCKS And TABLE LOCKS in The Database Are Released Using The COMMIT And ROLLBACK Statements.

Inserting Data

- The SQL INSERT Statement is Used As it Exists, in The PL/SQL Block For Data Insertion Standards.
- We Can Use All The Required SQL Built-in Functions, Like SYSDATE As Per Requirements.
- The PRIMARY KEY Values Can Be Generated Using The Database SEQUENCES.

- We Can Derive Values Within The PL/SQL Block As Per Necessity.
- Add Column Default Values If Any, As Per The Necessity.
- Any Identifiers in INSERT Clause Must Be An Database Column Name Only, To Avoid Ambiguity, And The Difference in "bit" Architecture.

```
SQL> BEGIN
        INSERT INTO Emp(Empno, Ename, Deptno, Job, MGR, HireDate, Sal, Comm)
        VALUES(EmpSequence.NEXTVAL, 'SATISH', 10, 'MANAGER', 7839,'22-NOV-1998',
                    4500,NULL);
        COMMIT;
        END;
        /
SQL> DECLARE
        V_Empno NUMBER(4) := &EnterEmpNo;
        V_Ename VARCHAR2(15) := '&EnterEname';
        V_Deptno NUMBER(2) := &EnterDepto;
        V_Job VARCHAR2(15) := '&EnterJob';
        V_MGR NUMBER := &EnterMGRNo;
        V_HireDate Emp.HireDate%TYPE := '&EnterHireDate';
        V_Sal Emp.Sal%TYPE := &EnterSal;
        V_Comm Emp.Comm%TYPE := &EnterComm;
        BEGIN
        INSERT INTO Emp(Empno, Ename, Deptno, Job, MGR, HireDate, Sal, Comm) VALUES
        (V_Empno,V_Ename, V_Deptno, V_Job, V_MGR, V_HireDate, V_Sal, V_Comm);
        COMMIT;
        END;
        /
```

Updating Data

- The Data Updation is Also Similar To The UPDATE Statements Used In SQL.
- A Small Amount of Ambiguity Can Arise in The SET Clause of The UPDATE Statement, As The Identifier on The Right Side of The Assignment Operator Can Be A PL/SQL Variable, OR Another Database Column Also.
- In The Set Clause, The Identifier on The Left is Always A Database Column.
- It is Always Better To Implement A WHERE Clause To Check That The Required Rows Only, Are Updated.
- If Any Rows Are Not Updated, Then NO ERROR is Returned.
- A Point To Note At This State is
  - PL/SQL Variable Assignment Use :=.
  - SQL Column Assignment Uses =.

```
SQL> DECLARE
        V_SalRaise Emp.Sal%TYPE := 3500;
        BEGIN
        UPDATE Emp SET Sal = Sal + V_SalRaise WHERE Job = 'ANALYST';
        END;

SQL> DECLARE
        V_JobChng Emp.Job%TYPE := UPPER('&GiveJob');
        V_Empno     Emp.Empno%TYPE := &GiveEmpno;
        BEGIN
        UPDATE Emp SET  Job = V_JobChngWHERE Empno = V_Empno;
        END;
```

Deleting Data

- Data Deletion Through PL/SQL is Also Practically Similar To The DELETE Statement Used in SQL.
- When Deleting The Data Utmost Care Has To Be Taken By Implementing WHERE Clause, Else Unwanted Data May Also Be Lost Along The Wanted Data.

Points To Note

- The Child And Parent Relations Should Be Cross Checked While Deletion Takes Place, To Avoid Orphan Records Generation.
- The ON DELETE Option If Set Through Constraints At The Time of Table Creation Will Get Activated As Per The Statement Requirement.
- Commit if Issued Releases All LOCKS That Are Existing.

```
SQL> DECLARE
        V_Deptno Emp.Deptno%TYPE := 30;
        BEGIN
        DELETE FROM Emp WHERE Deptno = V_Deptno;
        END;

SQL> DECLARE
        V_Deptno Dept.Deptno%TYPE := 40;
        BEGIN
        DELETE FROM Dept WHERE Deptno = V_Deptno;
        END;
```

Data Manipulation Using Cursors in PL/SQL

- To Process SQL Statements Oracle Needs To Create An Area of Memory Known As The CONTEXT AREA, OR PROCESS GLOBAL AREA (PGA).
- The CONTEXT AREA Contains The Information Needed To Process The Statement.
- The CONTEXT AREA Information Includes...
  - The Number of Rows Processed By The Statement.
  - A Pointer To The Parsed Representation of The Statement.
- Every Query Contains An ACTIVE SET, Which Refers To The Rows That Will Be Returned By The Query.

CURSOR Definition

- CURSOR is a Handle, OR Pointer To The CONTEXT AREA.

CURSOR Usage

- Using A CURSOR, The PL/SQL Program Can Control The CONTEXT AREA, As The SQL Statement is Being Processed.

CURSOR Features

- CURSOR Allows To FETCH And PROCESS Rows Returned By A SELECT Statement, One Row At A Time.
- A CURSOR is Named, Such That It Can Be Referenced By The PL/SQL Programmer Dynamically At Run Time.

Cursor Types

- CURSORS Are Broadly Recognized As Two Types.
  - Implicit Cursors
  - Explicit Cursors

Implicit Cursors

- It is A CURSOR That is Automatically Declared By Oracle Every Time An SQL Statement is Executed.
- The Programmer Cannot Control OR Process The Information in An Implicit Cursor.

Explicit Cursor

- It is A CURSOR That is Defined By The Programmer Within The Program For Any Query That Returns More Than One Row of Data.
- This Cursor is Declared Within The PL/SQL Block, And Allows Sequential Process of Each Row of The Returned Data From Database.

Implicit Cursors

- The Process Flow of An Implicit Cursor is....
  - Any Given PL/SQL Block Issues An Implicit Cursor Automatically Whenever An SQL Statement is Executed, As Long As Explicit CURSOR Does Not Exist.
  - A CURSOR is Automatically Associated With Every DML Statement.
  - All UPDATE And DELETE Statements Have Cursors That Identify The Set of Rows That Will Be Affected By The Operations.
  - An INSERT Statement Needs A Place To Receive The Data That is To Be Inserted in The Database, Which is Handled By The IMPLICIT Cursor.
  - The Most Recently Opened Cursor Is Called 'SQL%' Cursor, Which is A Representation For An IMPLICIT Cursor.
  - During Processing of An IMPLICIT Cursor, Oracle Automatically Performs The Operations Like "OPEN", "FETCH", AND "CLOSE" of The Context Area.

Cursor Attributes

- %ISOPEN
- %NOTFOUND
- %FOUND
- %ROWCOUNT
- The Attributes Return Information About The Execution Status of The Data Manipulation Statement Through CURSOR.

%ISOPEN

- Returns BOOLEAN Value.

- Evaluates To TRUE If The Cursor is Open Else Evaluates To FALSE.

%NOTFOUND

- Returns BOOLEAN Value.
- Evaluates To TRUE If The Most Recent Fetch Does Not Return A Row Else Evaluates To FALSE.

%FOUND

- Returns BOOLEAN Value.
- Evaluates To TRUE If The Most Recent Fetch Returns A Row Else Evaluates To FALSE.

%ROWCOUNT

- Returns A Number.
- Evaluates The Total Number of Row Returned So Far By The Cursor.

```
SQL> DECLARE
        V_Deptno Emp.Deptno%TYPE := &DepartmentNumber;
        BEGIN
        UPDATE Emp SET Sal = 1500 WHERE Deptno = V_Deptno;
        DBMS_OUTPUT.PUT_LINE(SQL%ROWCOUNT||' Rows were Updated.');
        IF SQL%NOTFOUND
        THEN
              DBMS_OUTPUT.PUT_LINE('Data Not Found... Updation Abruptly Terminated.');
        END IF;
        END;
        /
```

- As The Records Are Retrieved Into Memory At The Time The Cursor Is Opened, The Data Throughout The Transaction Has A Consistent View.
- After A Cursor is Opened, The New OR Changed Data is Not Reflected In The Cursor Result Set.

```
SQL> DECLARE
        V_Deptno Emp.Deptno%TYPE := &DeptNumber;
        BEGIN
        DELETE FROM Emp WHERE Deptno = V_Deptno ;
        DBMS_OUTPUT.PUT_LINE(SQL%ROWCOUNT||' Row(s) Affected.');
        IF SQL%NOTFOUND
        THEN
              DBMS_OUTPUT.PUT_LINE('Data Not Found... Deletion Abruptly Terminated.');
        END IF;
        END;
        /
```

Explicit Cursor

- An EXPLICIT CURSOR is Generated Using A Name With Association of A SELECT Statement in The DECLARE Section of The PL/SQL Block.

Advantages

- Explicit Cursors Provide More Programmatic Control For Programmers.
- Explicit Cursors Are More Efficient in Implementation, Hence Easy To Trap Errors.
- Explicit Cursors Are Designed To Work With SELECT Statements That Return More Than One Record At A Time.
- Explicit Cursor Require Additional Steps To Operate Than Implicit Cursors.
- There Are Four Steps That Should Be Performed By An Explicit Cursors.
  - Cursor Declaration.
  - CURSOR Opening.
  - CURSOR Fetching.
  - CURSOR Closing.

The Pseudo Structure

```
SQL> DECLARE
        Cursor Declaration
     BEGIN
     Opening of Cursor.
        Fetching of Cursor.
     CLOSE Cursor.
     END;
```

Illustration1

```
SQL> DECLARE
     V_Deptno Dept.Deptno%TYPE := &Deptno;
     CURSOR EDeptCursor  IS SELECT Empno, Ename, Deptno, Sal, Job
                            FROM Emp WHERE Deptno = V_Deptno;
     BEGIN
     OPEN EDeptCursor;
        Place Required FETCH Statements;
     CLOSE EDeptCursor;
     END;
```

Illustration2

```
SQL> DECLARE
     CURSOR EmpDetailsCursor  IS SELECT Empno, Ename, D.Deptno, Dname
                FROM Emp E, Dept D WHERE E.Deptno = D.Deptno;
     V_EmpDept EmpDetailsCursor%ROWTYPE;
     BEGIN
     OPEN EmpDetailsCursor;
        Place Required FETCH Statements;
     CLOSE EmpDetailsCursor;
     END;
```

Cursor Declaration

- The CURSOR is Declared in The Declarative Block And is Provided With A Name And A SELECT Statement.

Syntax

```
CURSOR CursorName{parameter List} [RETURN ReturnType] IS Query
        [FOR UPDATE[OF(ColumnList)] [NOWAIT]];
```

- The CursorName Can Be Any Valid Identifier.
- Parameter List is Optional And Can Be Any Valid Parameter Used For Query Execution.
- The Return Clause is Optional And Specifies The Type of Data To Be Returned.
- The Query Can Be Any SELECT Statement.
- The FOR UPDATE Clause LOCKS The Records When The Cursor is Opened, And Maintains The Status of The Cursor As READ ONLY.
- If NOWAIT is Specified, The Program Will Exit Immediately On Open If An Exclusive Lock Cannot Be Obtained.

Illustration1

```
SQL> DECLARE CURSOR EmpCursor  IS SELECT Ename, Sal, Job FROM Emp
                                        WHERE Deptno = 10;
```

Illustration2

```
SQL> DECLARE
     CURSOR EmpCursor(V_EmpDept NUMBER) IS SELECT Ename, Sal, Job FROM Emp
                                WHERE Deptno = V_EmpDept;
```

Opening The Cursor

- To Process Any Cursor For The Business Logic, It is Mandatory That It Should Be Opened.
- The Opening of Cursor Enables The Creation of Process Global Area(PGA) OR Context Area.
- Cursor Can Be Opened Only in The EXECUTION OR EXCEPTION Section of The PL/SQL Block.
- Any Cursor Can Be Opened Only Once in The PL/SQL Module, Second Time Opening of The Cursor Raises An Exception.
- Only Opened Cursors Can Be Processed.

Syntax

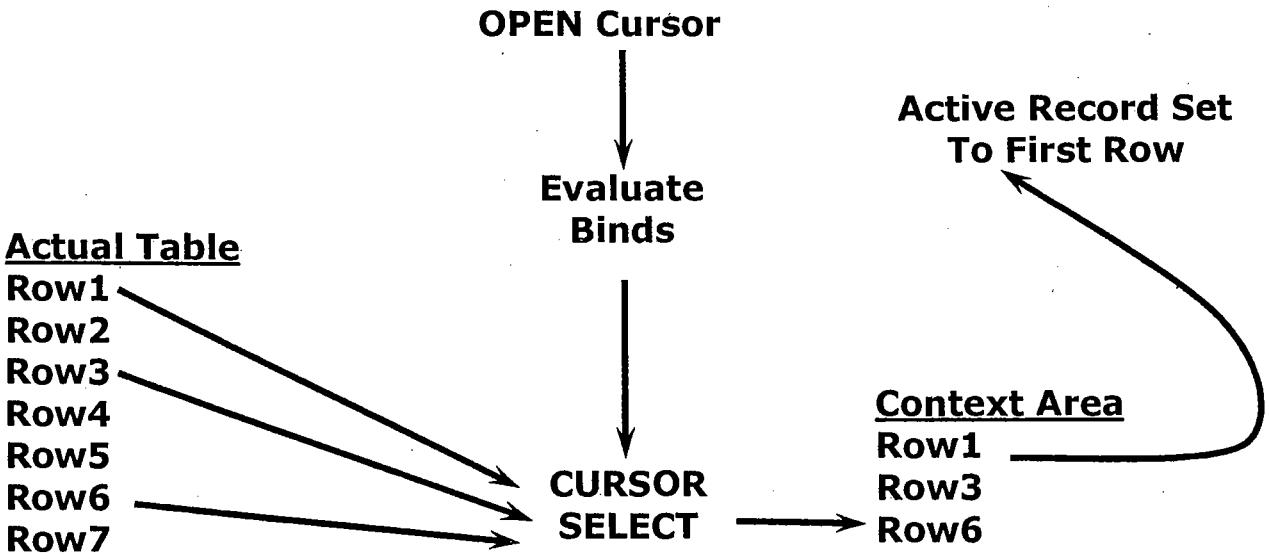
OPEN CursorName[(Parameter Values)];

Illustration1

1. OPEN EmpCursor;
2. OPEN EmpCursor(20);

Things To Note When Cursor is Opened

- The Query is Parsed, When Executed.
- Any Bind Values Are Evaluated First.
- The Rows Are Fetched And Recorded In The Context Area.
- The Result Set is Made Ready For Processing.
- There Can Be Only One Active Record in A Cursor At A Time.
- On The Opening of The Cursor, The Active Record Set is The First One Returned By The Cursors Query.



Fetching Records From The Cursor

- FETCH Retrievers Records From The CONTEXT AREA Into A Variable Such That The Variable Can Be Used For Business Logic.
- The FETCH Command Operates On The Current Record Only And Processes Through The Result Set One Record At A Time.

Syntax

FETCH CursorName INTO VariableName(s) OR PL/SQL_RECORD;

- The VariableName(s) Can Be One OR More Comma Delimited Variables That Match The Number And Type of Columns Included in The Result Set.

Illustration

FETCH EmpCursor INTO V_EmpRecord;

Closing The Cursor

- The Explicit Cursor Should Be Closed, Else It May Lead To Memory Leak.
- Until The CURSOR is Closed, The Memory is Not Released.

Syntax

CLOSE CursorName;

- CLOSE If Unnecessarily Used Raises An Exception.

Illustration

```
SQL> DECLARE
        V_Ename  EMP.ENAME%TYPE;
        V_Sal  EMP.SAL%TYPE;
        CURSOR MyCursor IS
        SELECT Ename, Sal FROM EMP;
        BEGIN
        OPEN MyCursor;
        FETCH MyCursor INTO V_Ename, V_Sal;
        DBMS_OUTPUT.PUT_LINE(V_Ename||' Salary is $ '||V_Sal);
        FETCH MyCursor INTO V_Ename, V_Sal;
        DBMS_OUTPUT.PUT_LINE(V_Ename||' Salary is $ '||V_Sal);
        CLOSE MyCursor;
        END;
        /
```

Note

- It is Always Better To Declare All The Bind Variables Processed By Cursor, Before The Cursor Declaration.
- After All Bind Variables Are Declared Declare The Required Cursor.
- To Process The Data Selected By Cursor, It is Necessary To Declare A "%ROWTYPE" Upon The Cursor.

Explicit Cursor Attributes

- The Cursor Attributes That Are Implemented Upon The Implicit Cursor Can Be Used As They Exit Along With Some Additional Attributes...

%BULK_EXCEPTIONS

- The Attribute is Used For ARRAY OR BULK COLLECT Operations Done in OLAP Applications.
- It Provides Information Regarding Exception Encountered During The Operation.

%BULK_ROWCOUNT

- Used On Bulk Collect Operations, The Attribute Provides Information Regarding The Number of Rows Changed During The Operation.

Cursor Loops

- CURSORS Are Generally Very Active With LOOPS, To Provide A Way To Navigate Through The ACTIVE RECORD SET.

Cursors With Simple Loops

- It Can Be Used With The Same Syntax As Discussed In Simple Loops OR Basic Loops.
- Within The Loop Each Record in The Active Set is Retrieved And Used.
- Each Loop Iteration Advances The Cursor Pointer By One Record in The Active Set.
- The EXIT WHEN Statement is Mandatory To Be Part of The Loop.

Points To Note

- The Loop Can Be Implemented As Pre Tested And Post Tested Loop.
- The Loop is Terminated With Proper Cursor Attribute Operating on %NOTFOUND Status.
- Before The Attribute Can Be Implemented The Context Area Status Han To Be Maintained With Minimum One Record of Fetching.

- If Proper Attributes Are Not Implemented OR The Attribute Position in The Loop is Not Proper Then The Loop Will Give Improper Results.
- The Loop Terminates When The Condition is TRUE.

```
SQL> DECLARE
        CURSOR EmpCursor
        IS
        SELECT * FROM Emp;
        V_EmpData EmpCursor%ROWTYPE;
        BEGIN
        OPEN EmpCursor;
        LOOP
                FETCH EmpCursor INTO V_EmpData;
                EXIT WHEN EmpCursor%NOTFOUND;
                DBMS_OUTPUT.PUT_LINE(V_EmpData.Ename);
        END LOOP;
        CLOSE EmpCursor;
        END;
        /

SQL> DECLARE
        CURSOR EmpCursor IS
        SELECT * FROM Emp;
        V_EmpData EmpCursor%ROWTYPE;
        BEGIN
        OPEN EmpCursor;
        LOOP
                FETCH EmpCursor INTO V_EmpData;
                DBMS_OUTPUT.PUT_LINE('Record Number : '||EmpCursor%ROWCOUNT||'
                                                        '||V_EmpData.Ename);
                EXIT WHEN EmpCursor%NOTFOUND;
        END LOOP;
        CLOSE EmpCursor;
        END;
        /

SQL> DECLARE
        CURSOR EmpCursor IS
        SELECT * FROM Emp;
        V_EmpData EmpCursor%ROWTYPE;
        BEGIN
        OPEN EmpCursor;
        DBMS_OUTPUT.PUT_LINE('The Details of Employees in Your Organization are as
        Follows');
        DBMS_OUTPUT.PUT_LINE('-------------------------------------------------------------------');
        DBMS_OUTPUT.PUT_LINE('      Name   '||' Designation  '||'  Basic Salary  '||'
                                                              Annual Salary  ');
        DBMS_OUTPUT.PUT_LINE('  ------ '||'------------ '||' ------------ '||' --------------- ');
        LOOP
        FETCH EmpCursor INTO V_EmpData;
        EXIT WHEN EmpCursor%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(RPAD(V_EmpData.Ename, 12)||RPAD(V_EmpData.Job,
                        12)||LPAD(V_EmpData.Sal, 10)||LPAD(V_EmpData.Sal * 12, 20));
        END LOOP;
        CLOSE EmpCursor;
```

```
END;
/

SQL> DECLARE
        V_RowCount NUMBER(4);
        CURSOR EmpCursor IS
        SELECT * FROM Emp;
        V_EmpData EmpCursor%ROWTYPE;
        BEGIN
        OPEN EmpCursor;
        DBMS_OUTPUT.PUT_LINE(RPAD('Employees Information', 80));
        DBMS_OUTPUT.PUT_LINE(RPAD('-', 80,'-'));
        DBMS_OUTPUT.PUT_LINE(RPAD('Ename', 12)||RPAD('Job', 12)||RPAD('Sal',
                                                        12)||RPAD('Comm', 10));
        DBMS_OUTPUT.PUT_LINE(RPAD('-', 80,'-'));
        LOOP
        FETCH EmpCursor INTO V_EmpData;
        EXIT WHEN EmpCursor%NOTFOUND;
        V_RoWCount := EmpCursor%ROWCOUNT;
        IF V_EmpData.Job = 'SALESMAN'
        THEN
                DBMS_OUTPUT.PUT_LINE(RPAD(V_EmpData.Ename,
                12)||RPAD(V_EmpData.Job, 12)||RPAD(V_EmpData.Sal, 12)||V_EmpData.Comm||'
                                Eligible For Commission.');
        ELSE
                DBMS_OUTPUT.PUT_LINE(RPAD(V_EmpData.Ename,
                        12)||RPAD(V_EmpData.Job, 12)||RPAD(V_EmpData.Sal, 12)||'Sorry!
                                        Not Eligible For Commission.');
        END IF;
        END LOOP;
        DBMS_OUTPUT.PUT_LINE(V_RowCount||' Rows Processed So Far...');
        CLOSE EmpCursor;
        END;
        /
```

Cursors With While Loops

- It Can Be Used With The Same Syntax As Discussed In While Loops.
- Within The Loop Each Record in The Active Set is Retrieved And Used.
- Each Loop Iteration Advances The Cursor Pointer By One Record in The Active Set.
- The Loop Depends on Minimum One TRUE State To Commence And Minimum One Condition To Terminate.
- The Loop Needs One FETCH Before The Loop To Make it To Commence, And One FETCH Inside The Loop To Keep it in Processing State.

```
SQL> DECLARE
        CURSOR EmpCursor IS
        SELECT * FROM Emp;
        V_EmpData EmpCursor%ROWTYPE;
        BEGIN
        OPEN EmpCursor;
        FETCH EmpCursor INTO V_EmpData;
        WHILE EmpCursor%FOUND
        LOOP
        DBMS_OUTPUT.PUT_LINE(V_EmpData.Ename);
        FETCH EmpCursor INTO V_EmpData;
        END LOOP;
```

```
        CLOSE EmpCursor;
        END;
        /

SQL> DECLARE
        CURSOR EmpCursor IS
        SELECT * FROM Emp;
        V_EmpData EmpCursor%ROWTYPE;
        BEGIN
        OPEN EmpCursor;
        FETCH EmpCursor INTO V_EmpData;
        WHILE EmpCursor%FOUND
        LOOP
        DBMS_OUTPUT.PUT_LINE('Record Number : '||EmpCursor%ROWCOUNT||'
                                                    '||V_EmpData.Ename);
        FETCH EmpCursor INTO V_EmpData;
        END LOOP;
        CLOSE EmpCursor;
        END;
        /
```

Cursors With For Loops

- It Can Be Used With The Same Syntax As Discussed In For Loops.
- Within The Loop Each Record in The Active Set is Retrieved And Used.
- Each Loop Iteration Advances The Cursor Pointer By One Record in The Active Set.
- The Loop Works On The Range Oriented Operational Logic.
- The Loop is Very Useful When Traveling The Entire Data in The Database Table.
- It is More Dynamic in Operation Than The Simple Loop And Basic Loop.
- Most Applications Are Implemented in Real Time Using The FOR LOOP Concept.
- While Using A FOR LOOP, The Cursor Does Not Require Explicit OPEN, FETCH, And CLOSE.
- The Processing of The Cursor is Implicitly Handled By The PL/SQL.
- The INDEX Variable Within The FOR LOOP Need Not Be Explicitly Declared.
- The Cursor For Loops Make The Program More Compact And Simple in Representation.
- If Proper Care is Not Taken in Implementing The Cursor Attributes, The Loop Can Be Confusional.
- The Loop Range is Associated With Proper Cursor Attributes, To Keep The Loop in Finite State.

```
SQL> DECLARE
        CURSOR EmpCursor IS
        SELECT * FROM Emp;
        V_EmpData EmpCursor%ROWTYPE;
        BEGIN
        FOR V_EmpData IN EmpCursor
        LOOP
                DBMS_OUTPUT.PUT_LINE(V_EmpData.Ename);
        END LOOP;
        END;
        /

SQL> DECLARE
        CURSOR EmpCursor IS
        SELECT * FROM Emp;
        BEGIN
        FOR EmpCursorIndex IN EmpCursor
        LOOP
```

```
        DBMS_OUTPUT.PUT_LINE('Record Number : '||EmpCursor%ROWCOUNT||'
                                                    '||EmpCursorIndex.Ename);
    END LOOP;
    END;
SQL> DECLARE
    CURSOR EmpCursor IS
    SELECT * FROM Emp ORDER BY Sal DESC;
    BEGIN
    FOR V_EmpData IN EmpCursor
    LOOP
    DBMS_OUTPUT.PUT_LINE('Rank No : '||EmpCursor%ROWCOUNT||'
                        '||V_EmpData.Ename||' With Salary of '||V_EmpData.Sal);
    EXIT WHEN EmpCursor%ROWCOUNT = 5;
    END LOOP;
    END;
    /
```

Cursor For Loops Using Sub Queries

- If A Sub Query is Instituted Into A PL/SQL Block Then A CURSOR DECLARATION is Not Necessary.
- In This Case The Sub Query is Embedded Into The FOR LOOP.
- In This Case The Performance of The Cursor Improves.
- The Cursor is Totally Implemented Implicitly By The FOR Loop.
- This Methodology Improves The Quality of The Cursor Management, And Makes The Process Compact.
- But This Methodology Can Be Some What Complicated If Not Followed Properly.

```
SQL> DECLARE
    V_Dname Dept.Dname%TYPE;
    V_Loc Dept.Loc%TYPE;
    BEGIN
    FOR EmpRecord IN (SELECT Ename, Deptno FROM Emp  ORDER BY Deptno)
    LOOP
    SELECT  Dname,  Loc  INTO  V_Dname,  V_Loc FROM Dept WHERE
                                            Deptno = EmpRecord.Deptno;
        DBMS_OUTPUT.PUT_LINE(EmpRecord.Ename||' '||EmpRecord.Deptno||'
                                            '||V_Dname||' '||V_Loc);
    END LOOP;
    END;

SQL> BEGIN
    DBMS_OUTPUT.PUT_LINE('The Departments Available in Our Organization Are...');
    FOR DeptRecord IN (SELECT * FROM Dept )
    LOOP
        DBMS_OUTPUT.PUT_LINE('Department '||DeptRecord.Deptno||' Named
                            '||DeptRecord.Dname||' Located At '||DeptRecord.Loc);
    END LOOP;
    END;
    /

SQL> DECLARE
    V_EmpCount  NUMBER(2);
    V_SumInvest  NUMBER(7);
    V_AvgInvest  NUMBER(7, 2);
    BEGIN
    DBMS_OUTPUT.PUT_LINE('The Departments Available in Our Organization Are...');
```

```
FOR DeptRecord IN (SELECT * FROM Dept )
LOOP
SELECT COUNT(*), SUM(Sal), AVG(Sal) INTO V_EmpCount, V_SumInvest,
      V_AvgInvest FROM Emp WHERE Deptno = DeptRecord.Deptno;
DBMS_OUTPUT.PUT_LINE('Department '||DeptRecord.Deptno||' Named
'||DeptRecord.Dname||' Located At '||DeptRecord.Loc||' With '||V_EmpCount||' Employees '||'
Invested With '||NVL(V_SumInvest, 0)||' Averaging '||NVL(V_AvgInvest, 0));
END LOOP;
END;
/
```

Applying Cursor Attribute Into Practice

- FETCH Rows Only When The CURSOR is OPEN.
- Use The %ISOPEN Cursor Attribute Before Performing A FETCH To Test Whether The Cursor is OPEN OR not.
- Use The %ROWCOUNT Cursor Attribute To Retrieve An Exact Number of Rows.
- Use The %NOTFOUND Cursor Attribute To Determine When To EXIT The Loop.
- The %FOUND Cursor Attribute Can Be Used in Situations Where We Can Take A Decision For Business Logic Implementation, Depending on The Identified Record.

```
SQL> DECLARE
      V_Empno Emp.Empno%TYPe;
      V_Ename Emp.Ename%TYPE;
      V_Sal Emp.Sal%TYPE;
      V_Desig Emp.Job%TYPE;
      CURSOR EmpCursor IS SELECT Empno, Ename, Sal, Job FROM Emp;
      BEGIN
      IF NOT EmpCursor%ISOPEN THEN
            OPEN EmpCursor;
      END IF;
      LOOP
      FETCH EmpCursor
      INTO V_Empno, V_Ename, V_Sal, V_Desig;
      EXIT WHEN
      EmpCursor%ROWCOUNT > 10 OR EmpCursor%NOTFOUND;
      DBMS_OUTPUT.PUT_LINE('Row Number : '||EmpCursor%ROWCOUNT||'
      '||RPAD(V_Empno, 12)||RPAD(V_Ename, 12)||RPAD(V_Desig, 12)||LPAD(V_Sal, 10));
      END LOOP;
      CLOSE EmpCursor;
      END;
      /
```

Applying Transactions Based On Data Fetched Through Cursor

- Cursors In Real Time Practice Are Used To Handle Business Logic More Effectively.
- In Practicality, The Data Fetched By A Cursor Can Be Used For Transactional Purpose Many Times.
- Cursors Can Help in Very Complicated Business Validations Involving The Cross Table Manipulations.
- The Data Fetched From One Table Can Be Used For Validation Purpose, Before Transacting On Another Table.

```
SQL> CREATE TABLE BigHeads ( Ename VARCHAR(20), Salary NUMBER(6));

SQL> DECLARE
      V_Number NUMBER(2) := &GiveNumber;
      V_Ename Emp.Ename%TYPE;
```

```
        V_Sal Emp.Sal%TYPE;
        CURSOR EmpCursor IS SELECT Ename, Sal FROM Emp WHERE
                                        Sal IS NOT NULL ORDER BY Sal DESC;
        BEGIN
        DELETE BigHeads;
        COMMIT;
        OPEN EmpCursor;
        FETCH EmpCursor INTO V_Ename, V_Sal;
        WHILE EmpCursor%ROWCOUNT <= V_Number AND EmpCursor%FOUND
        LOOP
              INSERT INTO BigHeads(Ename, Salary) VALUES(V_Ename, V_Sal);
        FETCH EmpCursor INTO V_Ename, V_Sal;
        END LOOP;
        CLOSE EmpCursor;
        COMMIT;
        END;
        /
```

```
SQL> CREATE TABLE BigHeads1 ( Ename VARCHAR(20), Salary NUMBER(6) );
```

```
SQL> DECLARE
        V_Number NUMBER(2) := &GiveNumber;
        V_Ename Emp.Ename%TYPE;
        V_CurrentSal Emp.Sal%TYPE;
        V_LastSal Emp.Sal%TYPE;
        V_Job Emp.Job%TYPE;
        CURSOR EmpCursor IS SELECT Ename, Sal, Job FROM Emp
                        WHERE Sal IS NOT NULL ORDER BY Sal DESC;
        BEGIN
        DELETE BigHeads1;
        COMMIT;
        OPEN EmpCursor;
        FETCH EmpCursor INTO V_Ename, V_CurrentSal, V_Job;
        WHILE EmpCursor%ROWCOUNT <= V_Number AND EmpCursor%FOUND
        LOOP
              INSERT INTO BigHeads1(Ename, Salary) VALUES(V_Ename, V_CurrentSal);
              V_LastSal := V_CurrentSal;
              FETCH EmpCursor INTO V_Ename, V_CurrentSal, V_Job;
              IF V_LastSal = V_CurrentSal THEN
                    V_Number := V_Number + 1;
              END IF;
        END LOOP;
        CLOSE EmpCursor;
        COMMIT;
        END;
        /
```

Cursors With Parameters

- A Cursor Can Be Declared With Parameters.
- The Advantages Are...
  - Enables Generation of Specific Result Sets.As Required.
  - Marks The Cursor Reusable.
  - Cursor Parameters Can Be Assigned With Default Values.
- The Mode of Cursor Parameters Can Be Only "IN" Mode.

- If A Cursor is Declared As Taking Parameters Then It Must Be Called With Values For Those Parameters.

Syntax

CURSOR CursorName(ParameterName Datatype,....) IS SELECT Statement;

Illustration

```
SQL> DECLARE
        CURSOR EmpCursor( PDeptno NUMBER, PDesig VARCHAR2 ) IS
        SELECT Empno, Job FROM Emp WHERE Deptno = PDeptno AND Job = PDesig;
```

Methods of Opening A Parametric Cursor:

1. Open Empcursor(20, 'CLERK');
2.
```
DECLARE
V_EmpJob Emp.Job%TYPE := &GiveJob;
BEGIN
OPEN(10, V_EmpJob);
```
3.
```
DECLARE
V_EmpJob Emp.Job%TYPE := &GiveJob;
V_EmpDept Emp.DeptNo%TYPE := &GD;
BEGIN
OPEN(V_EmpDept, V_EmpJob);
```
4.
```
DECLARE
CURSOR EmpCursor( PDeptno NUMBER, PDesig VARCHAR2 ) IS
      SELECT Empno, Job FROM Emp WHERE Deptno = PDeptno AND Job = PDesig;
BEGIN
FOR EmpRecord IN
EmpCursor(10, 'ANALYST')
LOOP
```

Points To Note

- Parameters Allow Values To Be Passed To A Cursor When It is Opened And To Be Used in The Query When it Executes.
- A Parametric Cursor Can Be Opened And Closed Explicitly Several Times in A PL/SQL Block.
- A Parametric Cursor Returns A Different Active Set on Each Occasion.
- Each FORMAL PARAMETER in The Cursor Declaration Must Have Corresponding ACTUAL PARAMETER in The OPEN Statement.
- When A Cursor is Opened We Pass Values To Each Parameter Positionally.
- PARAMETER Notation Does Not Offer Greater Functionality But Simply Allows Us To Specify Input Values Easily And Clearly.
- The Concept is More Useful When The Same Cursor is Referenced Repeatedly.

Working With Cursor Communication

- The Values That Are Fetched By One Cursor Can Be Passed As Parameters To The Other Cursor And Make The Programming More Dynamic.
- In This Case The Data Fetched By The Outer Cursor Are Used As Parameters By The Inner Cursor For Business Validation OR Transaction.

```
SQL> DECLARE
        CURSOR DeptCursor  IS SELECT Deptno, Dname FROM Dept ORDER BY Deptno;
        CURSOR EmpCursor(V_Deptno Emp.Deptno%TYPE)  IS SELECT Ename, Job, HireDate,
                              Sal FROM Emp WHERE Deptno = V_Deptno;
        V_CurrDeptno Dept.Deptno%TYPE;
        V_CurrDname Dept.Dname%TYPE;
        V_Ename Emp.Ename%TYPE;
        V_Job Emp.Job%TYPE;
        V_Mgr Emp.MGR%TYPE;
        V_HireDate Emp.HireDate%TYPE;
```

```
        V_Sal Emp.Sal%TYPE;
        BEGIN
        OPEN DeptCursor;
        LOOP
                FETCH DeptCursor INTO V_CurrDeptno, V_CurrDname;
                EXIT WHEN DeptCursor%NOTFOUND;
                DBMS_OUTPUT.PUT_LINE('Department Number : ' || V_CurrDeptno||'
                                                        Department Name : ' || V_CurrDname);
                IF EmpCursor%ISOPEN THEN
                CLOSE EmpCursor;
                END IF;
                OPEN EmpCursor(V_CurrDeptno);
                        LOOP
                                FETCH EmpCursor INTO V_Ename, V_Job, V_HireDate, V_Sal;
                                IF EmpCursor%FOUND THEN
                                        DBMS_OUTPUT.PUT_LINE(V_Ename||' '||V_Job||' '||'
                                        '||V_HireDate||' '||V_Sal);
                                ELSE
                                        DBMS_OUTPUT.PUT_LINE('****End of Employees, in
                                        Department : '||V_CurrDeptno||'.');
                                END IF;
                         EXIT WHEN EmpCursor%NOTFOUND;
                        END LOOP;
                IF EmpCursor%ISOPEN THEN
                CLOSE EmpCursor;
                END IF;
        END LOOP;
        IF EmpCursor%ISOPEN THEN
        CLOSE EmpCursor;
        END IF;
        CLOSE DeptCursor;
        END;

SQL> DECLARE
        SUMAmount NUMBER := 0;
        V_DeptNo Emp.Deptno%TYPE;
        CURSOR MyCursor (PDeptno NUMBER) IS SELECT Sal FROM Emp
                                                WHERE Deptno = PDeptno;
        V_MyCursor MyCursor%ROWTYPE;
        BEGIN
                BEGIN
                SUMAmount := 0;
                V_DeptNo := 10;
                OPEN MyCursor(V_Deptno);
                LOOP
                        FETCH MyCursor INTO V_MyCursor;
                        EXIT WHEN MyCursor%NOTFOUND;
                        SUMAmount := SUMAmount + V_MyCursor.Sal;
                END LOOP;
                DBMS_OUTPUT.PUT_LINE ('The Investment on Department '||V_Deptno||' is
                '||TO_CHAR(SUMAmount));
                CLOSE MyCursor;
                END;
                BEGIN
                SUMAmount := 0;
```

```
        V_DeptNo := 20;
        OPEN MyCursor(V_Deptno);
        LOOP
                FETCH MyCursor INTO V_MyCursor;
                EXIT WHEN MyCursor%NOTFOUND;
                SUMAmount := SUMAmount + V_MyCursor.Sal;
        END LOOP;
        DBMS_OUTPUT.PUT_LINE ('The Investment on Department '||V_Deptno||' is
        '||TO_CHAR(SUMAmount));
        CLOSE MyCursor;
        END;
        BEGIN
        SUMAmount := 0;
        V_DeptNo := 30;
        OPEN MyCursor(V_Deptno);
        LOOP
                FETCH MyCursor INTO V_MyCursor;
                EXIT WHEN MyCursor%NOTFOUND;
                SUMAmount := SUMAmount + V_MyCursor.Sal;
        END LOOP;
        DBMS_OUTPUT.PUT_LINE ('The Investment on Department '||V_Deptno||' is
        '||TO_CHAR(SUMAmount));
        CLOSE MyCursor;
        END;
END;
/
```

Cursors With For Update Clause

- Many Times While Working in Real Time We Have To LOCK Rows During Data UPDATION And Data DELETIONS.
- The FOR UPDATE Clause in The Cursor Query is Used To LOCK The Affected Rows While The Cursor is Opened.
- When FOR UPDATE Clause is Used We Need Not Use COMMIT As Oracle Server Releases LOCKS At The End of Each Transaction Automatically.
- The FOR UPDATE Cursors Are Compulsory While Operating With Real Time Data Updations And Deletions in Client Server Architecture.

Syntax

CURSOR CursorName IS SELECT ... FROM ...FOR UPDATE [OF ColumnReference][NOWAIT];

- The FOR UPDATE Clause is The Last Clause in The SELECT Statement.
- If The ORACLE SERVER Cannot Acquire The LOCKS on The Rows Then It Needs To Wait Indefinitely IF NOWAIT Clause is Not Used.

Illustration

```
SQL> DECLARE
        CURSOR EmpCursor IS SELECT EmpNo, EName, Sal FROM Emp WHERE DeptNo = 30
                                        FOR UPDATE OF Sal NOWAIT;
```

The WHERE CURRENT OF Clause:

- The WHERE CURRENT OF Clause is Used When Referencing The CURRENT ROW From An EXPLICIT FOR UPDATE CURSOR.
- The Clause Allows To Apply UPDATES And DELETES To The Row Currently Being Addressed By The Cursor, Without The Need To Explicitly Reference ROWID From The SQL Architecture.

- We Must Include The FOR UPDATE Clause in The Cursor Query, Such That The Rows Are LOCKED Upon Opening The Cursor.

Syntax

{UPDATE OR DELETE Statement} WHERE CURRENT OF CursorName;

- We Can Update Rows Based On Criteria From A FOR UPDATE CURSOR.
- We Can Write DELETE OR UPDATE Statement To Contain The WHERE CURRENT OF CursorName To Refer To The Latest Row Processed By The FETCH Statement of The FOR UPDATE Cursor.
- WHERE CURRENT OF When Used, The Cursor Must Contain OR Should Be of FOR UPDATE CLAUSE Cursor.

```
SQL> DECLARE
        CURSOR SalCursor  IS SELECT Sal FROM Emp WHERE Deptno = 30 FOR UPDATE OF
                                                              Sal NOWAIT;
        BEGIN
        FOR EmpRec IN SalCursor
        LOOP
              UPDATE Emp SET Sal = Sal + (EmpRec.Sal * 1.10) WHERE CURRENT
                                                                  OF SalCursor;
        END LOOP;
        COMMIT;
        END;
        /

SQL> DECLARE
        V_FromSal Emp.Sal%TYPE;
        V_ToSal Emp.Sal%TYPE;
        V_CurrentRow ROWID;
        CURSOR SalCursor IS SELECT ROWID, Ename, Sal FROM Emp WHERE Deptno = 30
        FOR UPDATE OF Sal NOWAIT;
        V_EmpRec SalCursor%ROWTYPE;
        BEGIN
        OPEN SalCursor;
        LOOP
        FETCH SalCursor INTO V_EmpRec;
        EXIT WHEN SalCursor%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE('The Salary for Update is '||V_EmpRec.Sal||' of
                                                          '||V_EmpRec.Ename||'.');

        V_FromSal := V_EmpRec.Sal;
        V_CurrentRow := V_EmpRec.ROWID;
        UPDATE Emp SET Sal = Sal + (V_EmpRec.Sal * 1.10)WHERE CURRENT OF SalCursor;
        SELECT Sal INTO V_ToSal  FROM Emp WHERE ROWID = V_CurrentRow;
        DBMS_OUTPUT.PUT_LINE('The Updated Salary Value is from '||V_FromSal||' To
                                  '||V_ToSal ||' Confirm The Update Before it is Committed.');
        END LOOP;
        END;
        /

SQL> CREATE TABLE MyEmp AS SELECT * FROM Emp;

SQL> DECLARE
        CURSOR SalCursor  IS SELECT ROWID, Sal FROM MyEmp WHERE Deptno = 30
                                                          FOR UPDATE OF Sal NOWAIT;
        BEGIN
        FOR EmpRec IN SalCursor
```

```
        LOOP
        DBMS_OUTPUT.PUT_LINE('The Present ROWID For Updation is : '||EmpRec.ROWID);
        DBMS_OUTPUT.PUT_LINE('Now The Row Will Be Locked...For Data Deletion.');
        DELETE MyEmp
        WHERE CURRENT OF SalCursor;
        DBMS_OUTPUT.PUT_LINE('Data Deletion is Successful Hence Releasing The ROW
                                                                    Lock.');
        END LOOP;
        END;
        /

SQL> CREATE TABLE EmpSalGraph AS SELECT Empno, Ename, Sal, Comm, Ename SalGraph
                    FROM Emp  WHERE 1 = 2
        /

SQL> ALTER TABLE EmpSalGraph MODIFY SalGraph VARCHAR2(70)
        /

SQL> INSERT INTO  EmpSalGraph(Empno, Ename, Sal, Comm) SELECT Empno, Ename, Sal,
                                                            Comm FROM Emp
        /
SQL> DECLARE
        V_Asterisk EmpSalGraph.SalGraph%TYPE := NULL;
        CURSOR EmpCursor IS SELECT NVL(ROUND(Sal/100),0) Sal FROM EmpSalGraph
                                                        FOR UPDATE;
        BEGIN
        FOR EmpSalGraphRec IN EmpCursor
        LOOP
                FOR MyIndex IN 1..EmpSalGraphRec.Sal
                LOOP
                        V_Asterisk := V_Asterisk||'*';
                END LOOP;
        UPDATE EmpSalGraph SET  SalGraph = V_Asterisk WHERE CURRENT OF EmpCursor;
        V_Asterisk := NULL;
        END LOOP;
        COMMIT;
        END;
        /
```

Cursors With Sub Queries

- A CURSOR Can Be Constructed Upon The Result Provided Through A SUBQUERY.
- The Sub Query Can Be An Ordinary One OR A Correlated Sub Query.
- The Sub Query When Evaluated Can Provide A Value On A Set of Values To The Statement.

```
SQL> DECLARE
        V_Deptno Dept.Deptno%TYPE;
        V_Dname Dept.DnaME%TYPE;
        V_Staff NUMBER(4);
        CURSOR StaffCountCursor IS SELECT T1.Deptno, T1.Dname, T2.Staff
                FROM Dept T1, (SELECT Deptno, COUNT(*) Staff FROM Emp
                    GROUP BY Deptno) T2 WHERE T1.Deptno = T2.Deptno AND T2.Staff >= 5;
        BEGIN
        OPEN StaffCountCursor;
        LOOP
                FETCH StaffCountCursor
                INTO V_Deptno, V_Dname, V_Staff;
```

```
        EXIT WHEN StaffCountCursor%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(V_Deptno||' '||V_Dname||' '||V_Staff);
    END LOOP;
    CLOSE StaffCountCursor;
    END;
```

Cursor Variables

- A Cursor Variable is A Data Structure That Points To A Cursor Object, Which in Turn Points To The Cursor's Result Set.
- We Can Use Cursor Variables To More Easily Retrieve Rows in A Result Set From Client And Server Programs.
- We Can Also Use Cursor Variables To Hide Minor Variations in Queries.
- To Use Cursor Variables, We Must First Create A Ref_Cursor Type, Then Declare A Cursor Variable Based on That Type.

Syntax 1 : Strong Cursor

```
TYPE Ref_Cursor_Name
IS REF CURSOR [RETURN Record_Type];
```

- A REF CURSOR With A RETURN Clause Define A "Strong" REF CURSOR.
- Cursor Variables on A Strong REF CURSOR Can Associate With Queries With Result Sets That Match The Number And Data Type of The Record Structure.

Syntax 2 : Weak Cursor

```
TYPE Ref_Cursor_Name
IS REF CURSOR;
```

- If We Do Not Include A RETURN Clause in The Cursor, Then it is A Weak REF CURSOR.
- Cursor Variables Declared From Weak REF Cursors Can Be Associated With Any Query At Runtime.

Example 1

```
TYPE EmpCursorType
IS REF CURSOR
RETURN Emp%ROWTYPE;
```

- Create The Variable Based on The REF CURSOR.
  Employee_Cur EmpCursorType;

Example 2

```
TYPE Any_CursorType
IS REF CURSOR;
```

- This Cursor is Called As a Generic Cursor Variable of Any_CursorType.

Syntax To OPEN A Cursor Variable

```
OPEN Cursor_Name
FOR
Select_Statement;
```

- FETCH And CLOSE of a Cursor Variable Uses The Same Syntax As For Explicit Cursors.

Restrictions on Cursor Variables

- Cursor Variables Cannot Be Declared in A Package Since They Do Not Have A Persistent State.
- We Cannot Use The FOR UPDATE Clause With Cursor Variables.
- We Cannot Assign NULLS To A Cursor Variable Nor Use Comparison Operators To Test For Equality, Inequality, OR Nullity.
- Neither Database Columns Nor Collections Can Store Cursor Variables.
- We Cannot Use RPCS To Pass Cursor Variables From One Server To Another.
- Cursor Variables Cannot Be Used With The Dynamic SQL Built-in Package DBMS_SQL.

```
SQL> DECLARE
        TYPE EmpRefCursor IS REF CURSOR;
        V_EmpRefCursor EmpRefCursor;
        V_Ename Emp.Ename%TYPE;
        BEGIN
        OPEN V_EmpRefCursor FOR SELECT Ename FROM Emp;
        LOOP
        FETCH V_EmpRefCursor INTO V_Ename;
        EXIT WHEN V_EmpRefCursor%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE('Employee Number
                        '||TO_CHAR(V_EmpRefCursor%ROWCOUNT, '09')||' : '||V_Ename);
        END LOOP;
        CLOSE V_EmpRefCursor;
        END;
        /

SQL> DECLARE
        V_RowCount NUMBER(4);
        TYPE GenericCursor
        IS REF CURSOR;
        V_GenericCursor GenericCursor;
        TYPE TablesRecordType IS RECORD( EmpRecord Emp%ROWTYPE);
        V_EmpRecordType TablesRecordType;
        BEGIN
        DBMS_OUTPUT.ENABLE(1000000);
        DBMS_OUTPUT.PUT_LINE(RPAD(LPAD('Employees Information', 49, '*'), 80, '*'));
        DBMS_OUTPUT.PUT_LINE(RPAD('-', 80,'-'));
        DBMS_OUTPUT.PUT_LINE(RPAD('EmpNo', 8)|| RPAD('Ename', 12)||RPAD('Job',
        12)||RPAD('Deptno', 8)||RPAD('Mgr', 10)||RPAD('Hiredate', 12)||RPAD('Sal',
        12)||RPAD('Comm', 10));
        DBMS_OUTPUT.PUT_LINE(RPAD('-', 80,'-'));
        FOR LoopIndex IN ( SELECT Deptno, Dname  FROM Dept  )
        LOOP
                OPEN V_GenericCursor  FOR   SELECT *  FROM Emp WHERE Deptno =
                                                        LoopIndex.Deptno;
                DBMS_OUTPUT.PUT_LINE('----------------------------------------------------------------
                ------------');
                DBMS_OUTPUT.PUT_LINE('Now Printing The Employees of Department :
                '||LoopIndex.Dname);
                DBMS_OUTPUT.PUT_LINE('----------------------------------------------------------------
                ------------');       V_RoWCount := 0;
                LOOP
                        FETCH V_GenericCursor
                        INTO V_EmpRecordType.EmpRecord;
                        EXIT WHEN V_GenericCursor%NOTFOUND;
                        V_RoWCount := V_GenericCursor%ROWCOUNT;
                        DBMS_OUTPUT.PUT_LINE(RPAD(V_EmpRecordType.EmpRecord.Emp
                        no, 8) ||RPAD(V_EmpRecordType.EmpRecord.Ename,
                        12)||RPAD(V_EmpRecordType.EmpRecord.Job,
                        12)||RPAD(V_EmpRecordType.EmpRecord.Deptno,
                        8)||NVL(TO_CHAR(RPAD(V_EmpRecordType.EmpRecord.MGR, 10)),'No
                        Manager')||RPAD(V_EmpRecordType.EmpRecord.Hiredate,
                        12)||RPAD(V_EmpRecordType.EmpRecord.Sal,
                        12)||NVL(TO_CHAR(RPAD(V_EmpRecordType.EmpRecord.Comm, 12)),'-
                        NA-'));
```

```
        END LOOP;
        CLOSE V_GenericCursor;
    DBMS_OUTPUT.PUT_LINE(V_RowCount||' Rows Processed So Far...');
    END LOOP;
    END;
```

BULK COLLECT For Multiple Row Query

- The BULK COLLECT Statement Retrieves Multiple Rows of Data Through Either An Implicit OR An Explicit Query With A Single Round Trip To And From The Database.
- BULK COLLECT Reduces The Number of I/O Cycles Between The PL/SQL And SQL Engines And Reduces The Overhead of Retrieving Data.

```
SQL> DECLARE
        TYPE EmpTableType  IS TABLE OF  Emp%ROWTYPE;
        V_EmpTableType EmpTableType;
        BEGIN
        SELECT *  BULK COLLECT INTO V_EmpTableType FROM Emp;
        FOR IndexVariable IN V_EmpTableType.FIRST .. V_EmpTableType.LAST
        LOOP
        DBMS_OUTPUT.PUT_LINE(V_EmpTableType(IndexVariable).Ename || ' is Fixed With A
                                Salary of ' || V_EmpTableType(IndexVariable).Sal);
        END LOOP;
        END;
```

Exception Handling In PL/SQL

- Exception is An IDENTIFIER in PL/SQL, Raised During The Execution of A PL/SQL Block.
- Once An EXCEPTION Arises It Terminates The Main Body of Actions Performed By The PL/SQL Block.
- A PL/SQL Block Immediately Terminates When PL/SQL Raises An Exception, By Specifying An EXCEPTION HANDLER We Can Avoid This Abrupt Termination And Perform Final Action For The Business Logic.

Methods To Raise An Exception

- An ORACLE ERROR OCCURS And The Associated EXCEPTION is Raised Automatically OR Implicitly By Oracle.
- RAISE The EXCEPTION Explicitly By Issuing The RAISE Statement Within The Execution Block of PL/SQL.
- The RAISED EXCEPTION Can Be Either USER DEFINED OR PREDEFINED.

Exception Handling

- Exception Handling Can Happen in Two Ways
  - Exception Trapping
  - Exception Propagation

Exception Trapping

- If An Exception is Raised in The EXECUTABLE Section of The PL/SQL Block Then Process That EXCEPTION To The Corresponding EXCEPTION HANDLER in The EXCEPTION Section of The Same Block.
- If The EXCEPTION is Successfully Handled Then The EXCEPTION Does Not Propagate To The Enclosing Block of Programming Environment.
- If EXCEPTION is Trapped And Handled Successfully Then The PL/SQL Block Terminates Successfully.

Exception Propagation

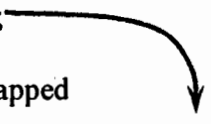
- If An EXCEPTION is Raised in The EXECUTABLE SECTION of The PL/SQL Block And There is No Corresponding EXCEPTION HANDLER Then The PL/SQL Block Terminates With Failure And The EXCEPTION is PROPAGATED To The Calling Environment.

Exception Trapping

```
SQL> DECLARE
        BEGIN
        Exception Raised;
        EXCEPTION
        Exception Trapped;
        END;
```

Exception Propagation

```
SQL> DECLARE
        BEGIN
        Exception Raised;
        EXCEPTION
        Exception Not Trapped
        END;
```



Exception Propagating To The Enclosing Blocks

Types of Exceptions

- As Per PL/SQL Exceptions Are Considered To Be of Three Types
  - Pre Defined Oracle Server Errors.
  - Non Pre Defined Oracle Server Errors.
  - User Defined Errors
- The PL/SQL Programmer Can Program For EXCEPTIONS To Avoid Disruption At Runtime.

Specifications of Exceptions

1. Exception Type
   - Predefined Oracle Server Errors.

   Description
   - These Are One of Appropriately 20 Errors That Occur Most Often in PL/SQL Code.

   Handling Tip:
   - Do Not Declare The Exception And Allow The Oracle Server To RAISE Then Implicitly.
   - But The Programmer Should Handle it Using The Proper EXCEPTION Handle.
2. Exception Type
   - Non-Predefined Oracle Server Errors.

   Description
   - It is Any Other Standard Oracle Server Error.

   Handling Tip
   - Declare The EXCEPTION Identifier With in The Declarative Section And Allow The Oracle Server To RAISE Them Implicitly.
   - But The Programmer Should Handle it Using The User Defined EXCEPTION Handle Declared in The Declarative Section.
3. Exception Type
   - User Defined Errors.

   Description
   - It is Any Condition That The Developer Determines, Which is Abnormal According To The Business Rule.

   Handling Tip
   - Declare EXCEPTION Identifier Within The Declarative Section And RAISE The EXCEPTION Explicitly Using The RAISE Statement.

Trapping Exceptions

- Including A Corresponding Exceptional Handling Routine Within The EXCEPTION Handling Section of The PL/SQL Block Can Trap Any Errors Raised While Running A PL/SQL Block.
- Each Exception Handler Consists of A WHEN Clause Which Specifies An EXCEPTION That Has To Be Handled.
- Each EXCEPTION is Followed By A Sequence of Statements To Be Executed When That Exception is Raised.

Syntax

```
EXCEPTION
WHEN Exception1[OR Exception2...] THEN
Statement1;
Statement2;
WHEN OTHERS THEN
Statement1;
Statement2;
END;
```

- OTHERS is An Optional EXCEPTION Handling Clause That Traps Unspecified Exceptions.

WHEN OTHERS Exception Handler

- The EXCEPTION Handling Section Taps Only Those Exceptions That Are Specified By The Programmer Using Proper Exception Handler.
- Any Other Exceptions Are Propagated Which Are Not Trapped, Unless The OTHERS EXCEPTION Handle is Used.
- The OTHERS Should Be The Last Exception Handle in The Definition.

Trapping Exceptions Guidelines

- Begin The EXCEPTION Handling Section of The Block With The Keywords EXCEPTION.
- Define Several EXCEPTION HANDLERS, Each With its Own Set of Action For The Block.

- When An EXCEPTION Occurs PL/SQL Processes Only One Handle Before Leaving The Block.
- Place The OTHERS Clause After All Other Exception Handling Clauses.
- We Can Have At Most One OTHERS Clause Only.
- Exceptions Cannot Appear in Argument Statement OR SQL Statements.

Handling Predefined Exceptions

- A Predefined Oracle Server Error is Trapped By Referencing Its Standard Name Within The Corresponding Exception Handling Runtime.
- All Predefined Exceptions Are Declared By PL/SQL is The Standard Packages.
- All Predefined EXCEPTIONS Stated By Oracle Have Their Own Exception Number And Exception Handle.
- As Soon As The Exception Occurs Oracle Implicitly Raises The Exception And Jumps Into The EXCEPTION Handle Block, If Proper EXCEPTION Handle is Found Manages The Specified Steps.
- In Predefined Oracle Server Exceptions Only One Exception is Raised And Handled At Any Time.

```
SQL> DECLARE
        V_Empno Emp.Empno%TYPE := &Empno;
        V_Ename Emp.Ename%TYPE;
        V_Sal Emp.Sal%TYPE;
        BEGIN
        SELECT Ename, Sal INTO V_Ename, V_Sal FROM Emp WHERE Empno = V_Empno;
        DBMS_OUTPUT.PUT_LINE('Name : '||V_Ename||' Salary : '||V_Sal);
        EXCEPTION
        WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE('Sorry, Data is Not Found.');
        END;
        /

SQL> DECLARE
        V_Emp Emp%ROWTYPE;
        V_Sal Emp.Sal%TYPE := &Sal;
        BEGIN
        SELECT * INTO V_Emp FROM Emp WHERE Sal = V_Sal;
        DBMS_OUTPUT.PUT_LINE('Name : '||V_Emp.Ename||' Salary : '||V_Emp.Sal);
        EXCEPTION
        WHEN TOO_MANY_ROWS THEN
        DBMS_OUTPUT.PUT_LINE('More Than One Employee Having Same Salary');
        END ;
        /

SQL> DECLARE
        V_Empno VARCHAR2(4):='&Empno';
        V_Ename VARCHAR2(20):='&Ename';
        V_Deptno VARCHAR2(2):='&Deptno';
        BEGIN
        INSERT INTO Emp(Empno,Ename,Deptno) VALUES (V_Empno,V_Ename,V_Deptno);
        EXCEPTION
        WHEN INVALID_NUMBER THEN
        DBMS_OUTPUT.PUT_LINE('Given Employee Number or Department Number is Invalid');
        END ;
        /
```

```
SQL> DECLARE
        V_Num1 NUMBER;
        BEGIN
        V_Num1 := '&GiveNumber1' + '&GiveNumber2';
        DBMS_OUTPUT.PUT_LINE ('The Result of the Operation is: ' || V_Num1);
        EXCEPTION
        WHEN VALUE_ERROR THEN
        DBMS_OUTPUT.PUT_LINE ('Please Check - There is a Source of Invalid Values in your
        input (OR) Operations.');
        END ;
        /

SQL> DECLARE
        V_Empno Emp.Empno%TYPE := &Empno;
        V_Ename Emp.Ename%TYPE := '&Ename';
        V_Deptno Emp.Deptno%TYPE := &Deptno;
        BEGIN
        INSERT INTO Emp(Empno,Ename,Deptno) VALUES (V_Empno,V_Ename,V_Deptno);
        DBMS_OUTPUT.PUT_LINE('Employee Successfully Inserted');
        EXCEPTION
        WHEN DUP_VAL_ON_INDEX THEN
        DBMS_OUTPUT.PUT_LINE('Employee ID Already Exists');
        END ;
        /

SQL> DECLARE
        V_Ename Emp.Ename% TYPE;
        V_Sal Emp.Sal%TYPE;
        V_RowCount PLS_INTEGER := 0;
        CURSOR EmpRowCount IS SELECT Ename, Sal FROM Emp ORDER BY Ename;
        BEGIN
        OPEN EMPROWCOUNT;
        LOOP
        FETCH EmpRowCount INTO V_Ename, V_Sal;
        EXIT WHEN EmpRowCount%NOTFOUND;
        V_RoWCount := EmpRoWCount%ROWCOUNT;
        DBMS_OUTPUT.PUT_LINE('Employee Name is,'||V_Ename||' his Salary is '||V_Sal);
        END LOOP;
        OPEN EMPROWCOUNT;
        DBMS_OUTPUT.PUT_LINE(V_RowCount||' Rows Processed So Far...');
        CLOSE EmpRowCount;
        EXCEPTION
        WHEN CURSOR_ALREADY_OPEN THEN
        DBMS_OUTPUT.PUT_LINE('The Requested Cursor is already Open.');
        END;
        /

SQL> DECLARE
        V_Ename Emp.Ename% TYPE;
        V_Sal Emp.Sal%TYPE;
        V_RowCount PLS_INTEGER := 0;
        CURSOR EmpRowCount IS SELECT Ename, Sal FROM Emp ORDER BY Ename;
        BEGIN
        OPEN EMPROWCOUNT;
        LOOP
        FETCH EmpRowCount INTO V_Ename, V_Sal;
```

```
        EXIT WHEN EmpRowCount%NOTFOUND;
        V_RoWCount := EmpRoWCount%ROWCOUNT;
        DBMS_OUTPUT.PUT_LINE('Employee Name is,'||V_Ename||' his Salary is '||V_Sal);
        END LOOP;
        DBMS_OUTPUT.PUT_LINE(V_RowCount||' Rows Processed So Far...');
        CLOSE EmpRowCount;
        CLOSE EmpRowCount;
        EXCEPTION
        WHEN INVALID_CURSOR THEN
        DBMS_OUTPUT.PUT_LINE('The Requested Cursor is either not open or is already
        closed.');
        END;
        /

SQL> DECLARE
        V_Grade CHAR := UPPER('&EnterGrade');
        BEGIN
        CASE V_Grade
                WHEN 'A' THEN
                        DBMS_OUTPUT.PUT_LINE('You are Awarded with Excellent Grade');
                WHEN 'B' THEN
                        DBMS_OUTPUT.PUT_LINE('You are Awarded with Very Good Grade');
                WHEN 'C' THEN
                        DBMS_OUTPUT.PUT_LINE('You are Awarded with Good Grade');
                WHEN 'D' THEN
                        DBMS_OUTPUT.PUT_LINE('You are Awarded with Fair Grade');
                WHEN 'E' THEN
                        DBMS_OUTPUT.PUT_LINE('You are Awarded with Poor Grade');
        END CASE;
        EXCEPTION
        WHEN CASE_NOT_FOUND THEN
        DBMS_OUTPUT.PUT_LINE('The Supplied Case '||V_Grade||' not found. Please Check
                                                                once again.');
        END;
        /

SQL> DECLARE
        V_Num1 NUMBER := &GiveNumber1;
        V_Num2 NUMBER := &GiveNumber2;
        V_Result NUMBER;
        BEGIN
        V_Result := V_Num1/V_Num2;
        DBMS_OUTPUT.PUT_LINE ('The Result is: ' || V_Result);
        EXCEPTION
        WHEN ZERO_DIVIDE THEN
        DBMS_OUTPUT.PUT_LINE ('Fatal Error - Division by zero occurred');
        END ;
        /

SQL> DECLARE
        V_Ename Emp.Ename%TYPE;
        V_Sal Emp.Sal%TYPE;
        V_Job Emp.Job%TYPE;
        V_Deptno Emp.Deptno%TYPE;
        BEGIN
```

```
SELECT Ename, Sal, Job, Deptno INTO V_Ename, V_Sal, V_Job, V_Deptno FROM Emp
                                        WHERE Empno = &GiveEmpNo;
DBMS_OUTPUT.PUT_LINE('The Employee Name is :'||V_Ename||' Working for
                              Department '||V_Deptno||' Having Salary of '||V_Sal||'.');
      DECLARE
      V_Staff NUMBER(1);
      BEGIN
        SELECT COUNT(*) INTO V_Staff FROM Emp WHERE Deptno = V_Deptno;
            DBMS_OUTPUT.PUT_LINE('The Total Number of Employees Working in
                                        Department '||V_Deptno||' are '||V_Staff||'.');
          EXCEPTION
          WHEN VALUE_ERROR OR INVALID_NUMBER THEN
           DBMS_OUTPUT.PUT_LINE('There is some Error in the Data inputs (OR)
           Data outputs, Please Check, Debug the Source.');
      END;
EXCEPTION
WHEN NO_DATA_FOUND THEN
DBMS_OUTPUT.PUT_LINE('The Given Information is Missing in the Database, Check for
Proper Values.');
END;
/
```

Trapping Non-Predefined Oracle Server Errors

- The Non-Predefined Oracle Server Error is Trapped By Declaring it First OR By Using The OTHERS Exception Handle.
- The Declared EXCEPTION is RAISED Implicitly By The Oracle Server.
- THE PL/SQL PRAGMA EXCEPTION_INIT Can Be Used For Associating EXCEPTION Name With An Oracle Error Number.
- The PRAGMA EXCEPTION_INIT Tells The PL/SQL Engine Completely To Associate An EXCEPTION Name With An Oracle Error Number.
- The PRAGMA EXCEPTION_INIT Allows Programmer To Refer To Any Internal EXCEPTION By The Name And Associate That To Specific Handles.
- PRAGMA is Also Called As PSEUDO INSTRUCTION, And is A Key Word That Signifies, That The Statement is A Compiler Directive.
- The Statement is Not Processed When That PL/SQL Block is Executed Rather The PRAGMA Directs The PL/SQL Compiler To Interpret All Occurrences of The EXCEPTION Name Within The Block As The Associated Oracle Server Error Number.

DECLARE	Associate	Reference
EXCEPTION Name	Declarative Section	EXCEPTION Handling Section
	Code The PRAGMA EXCEPTION	Handle The RAISED EXCEPTION

Syntax

- Declaring The Name For The EXCEPTION With in The Declarative Section.
  ExceptionIdentifier EXCEPTION;
- Associate The Declared EXCEPTION With PRAGMA.
  PRAGMA EXCEPTION_INIT(ExceptionIdentifier, ErrorNumber);
- Reference The Declared EXCEPTION Within The Corresponding EXCEPTION Handle At Runtime.

Illustration

```
SQL> DECLARE
        E_EmpRemaining EXCEPTION;
        PRAGMA EXCEPTION_INIT(E_EmpRemaining, -2292);

SQL> DECLARE
        E_EmpRemaining EXCEPTION;
        PRAGMA EXCEPTION_INIT(E_EmpRemaining, -2292);
        V_Deptno Emp.Deptno%TYPE := &GiveDeptno;
        BEGIN
        DELETE FROM Dept WHERE Deptno = V_Deptno;
        IF SQL%NOTFOUND THEN
        DBMS_OUTPUT.PUT_LINE('The Given Information is Missing in the Database, Check for
                                                Proper Values.');
        END IF;
        ROLLBACK;
        EXCEPTION
        WHEN E_EmpRemaining THEN
        DBMS_OUTPUT.PUT_LINE('Unable to Delete the Department Number '||V_Deptno||' as
              the Employees are Existing. Validate Your Relations and then Try Once Again.');
        WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE('The Given Information is Missing in the Database, Check for
              Proper Values.');
        END;
        /

SQL> DECLARE
        E_EmpExists EXCEPTION;
        V_Count NUMBER(2);
        V_SalSum NUMBER(6);
        PRAGMA EXCEPTION_INIT(E_EmpExists, -2292);
        V_Empno Emp.Empno%TYPE := &GiveEmpno;
        BEGIN
        DELETE FROM Emp WHERE Empno = V_Empno;
        IF SQL%NOTFOUND THEN
        DBMS_OUTPUT.PUT_LINE('The Given Employee Number '||V_Empno||' is Missing in the
        Database, Check for Proper Values.');
        ROLLBACK;
        ELSE
        COMMIT;
        END IF;
        EXCEPTION
        WHEN E_EmpExists THEN
        DBMS_OUTPUT.PUT_LINE('Unable to Delete the Employee Details '||V_Empno||' as the
                    Employees are Existing. Validate Your Relations and then Try Once Again.');
        WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE('The Given Information is Missing in the Database, Check for
                    Proper Values.');
        END;
```

```
SQL> DECLARE
        E_NotNULLViolation EXCEPTION;
        PRAGMA EXCEPTION_INIT(E_NotNULLViolation, -1400);
        BEGIN
        INSERT INTO Emp(Empno, Ename, Job, Sal, Comm, Deptno) VALUES(&Empno,
                                    'SATISH', 'ANALYST', 25000, NULL, &Deptno);
        COMMIT;
        EXCEPTION
        WHEN E_NotNULLViolation THEN
        DBMS_OUTPUT.PUT_LINE('A Field which Cannot be NULL, is not attended, Please
        Check Properly...');
        END;
        /

SQL> ALTER TABLE Emp ADD CONSTRAINT EmpEnameCHK
              CHECK(Ename = UPPER(Ename)) ADD CONSTRAINT EmpJobCHK
                                                  CHECK(Job = UPPER(Job))
        /
SQL> DECLARE
        E_NotNULLViolation EXCEPTION;
        PRAGMA EXCEPTION_INIT(E_NotNULLViolation, -1400);
        E_CheckViolation EXCEPTION;
        PRAGMA EXCEPTION_INIT(E_CheckViolation, -2290);
        BEGIN
        INSERT INTO Emp(Empno, Ename, Job, Sal, Comm, Deptno)
        VALUES(&Empno, '&Ename', '&Job', 25000, NULL, &Deptno);
        COMMIT;
        EXCEPTION
        WHEN E_CheckViolation THEN
        DBMS_OUTPUT.PUT_LINE('A Field with Check Constraint is not Attended Properly,
        Please Check Properly.');
        WHEN E_NotNULLViolation THEN
        DBMS_OUTPUT.PUT_LINE('A Field which Cannot be NULL, is not attended, Please
        Check Properly.');
        END;
        /
```

Functions For Trapping EXCEPTIONS

- When An EXCEPTION Occurs We Can Identify The Associated ERROR CODE OR ERROR MESSAGE By Using The Exception Trapping Functions.
- Based on The Values of The Code OR Message We Can Decide What Subsequent Action Has To Be Taken Based Upon The Error.

SQLCODE Function

- It Returns The Numeric Value For The Internal Oracle Errors.
- It Can Be Assigned To A NUMBER Variable.

SQLERRM Function

- An Error Number Can Be Passed To SQLERRM.
- SQLERRM Returns The Message Associated With The Error Number.
- It Can Be Assigned To A VARCHAR2 Variable.
- The Maximum Length of A Message Returned By The SQLERRM Function is 512 Bytes.

SQLCODE Values

- 0 → No Exception Has Encountered.
- 1 → User Defined Exception.

- +100 → NO_DATA_FOUND Exception.
- These Functions Are More Specific Usage When We Use OTHERS THEN Exception Handle, Such That The Unknown Error Can Be Treated And Handled in A More Proper Way.
- As Per The Standards of Object Oriented Theory EXCEPTION is Any Event That is Either Successful OR Failure, Which Can Be Properly Handled With EXCEPTION Handling Routines.

Pseudo Syntax

```
SQL> DECLARE
        V_ErroeCode NUMBER;
        V_ErrorMessage VARCHAR2(255);
        BEGIN
        /*Executional Code Here*/
        EXCEPTION
        WHEN OTHERS THEN
        ROLLBACK;
        V_ErrorCode := SQLCODE;
        V_ErrorMessage := SQLERRM;
        INSERT INTO Errors VALUES(V_ErrorCode, V_ErrorMessage);
        END;
        /

SQL> DECLARE
        V_ErrorCode NUMBER(6);
        V_ErrorMessage VARCHAR2(200);
        BEGIN
        INSERT INTO Dept VALUES(&DeptNumber, '&DeptName', '&DeptLocation');
        EXCEPTION
        WHEN OTHERS THEN
        V_ErrorCode := SQLCODE;
        V_ErrorMessage := SUBSTR(SQLERRM, 1, 200);
        DBMS_OUTPUT.PUT_LINE('The Error Code Traced is : '||V_ErrorCode);
        DBMS_OUTPUT.PUT_LINE('The Error Message Traced is : '||V_ErrorMessage);
        END;
        /

SQL> DECLARE
        V_Empno Emp.Empno%TYPE := &EmpNumber;
        V_EName Emp.EName%TYPE;
        V_Job Emp.Job%TYPE;
        V_Sal Emp.Sal%TYPE;
        V_ErrorCode NUMBER(6);
        V_ErrorMessage VARCHAr2(200);
        BEGIN
        SELECT Ename, Job, Sal INTO V_EName, V_Job, V_Sal FROM Emp
        WHERE Empno = V_Empno;
        DBMS_OUTPUT.PUT_LINE(V_Ename||'    '||V_Job||'    '||V_Sal);
        EXCEPTION
        WHEN OTHERS THEN
        V_ErrorCode := SQLCODE;
        V_ErrorMessage := SUBSTR(SQLERRM, 1, 200);
        DBMS_OUTPUT.PUT_LINE('The Error Code Traced is : '||V_ErrorCode);
        DBMS_OUTPUT.PUT_LINE('The Error Message Traced is : '||V_ErrorMessage);
        END;
        /
```

Trapping User Defined Exceptions:

- PL/SQL Helps To Determine Customized EXCEPTIONS.
- User Defined PL/SQL EXCEPTIONS Must Be
  - Declared in The DECLARE Section of The PL/SQL Block.
  - RAISED Explicitly With RAISE Statements.
- The User Defined EXCEPTIONS Should Be Planned When The EXCEPTIONS Related To Business Process Have To Be Maintained.
- The User Defined EXCEPTIONS Are The Intelligence of Domain Expertise.
- The General Process Followed is As Follows

DECLARE	RAISE	Reference
EXCEPTION Name	Explicitly RAISE The Exception By Using The RAISE Statement	Handle The RAISED Exception
Declarative Section	Executable Section	EXCEPTION Handling Section

The Steps Followed Are

1. Declare The Name For The User Defined Exception

Syntax

ExceptionIdentifier EXCEPTION

2. Use The RAISE Statement To Raise The Exception Writing in The Executable Section.

Syntax

RAISE ExceptionIdentifier;

3. Reference The Declared Exception Within The Corresponding Exception Handling Runtime.

```
SQL> DECLARE
        V_DeptNo Dept.Deptno%TYPE := &DeptNumber;
        V_DeptName Dept.Dname%TYPE := '&DeptName';
        V_DeptLoc Dept.Loc%TYPE := '&DeptLocation';
        E_InvalidDept EXCEPTION;
        BEGIN
        UPDATE Dept vSET Dname=V_DeptName, Loc = V_DeptLocWHERE Deptno= DeptNo;
        IF SQL%NOTFOUND THEN
        RAISE E_InvalidDept;
        END IF;
        COMMIT;
        EXCEPTION
        WHEN E_InvalidDept THEN
        DBMS_OUTPUT.PUT_LINE('The Specific Department Number '||V_DeptNo||' you wanted
                        to Update is not Found. Please Confirm the Data.');
        INSERT INTO MyAudit(UserName, ModDate, Message) VALUES(USER, SYSDATE,
                                                        'Tried Illegal Update.');
        END;
        /

SQL> DECLARE
        V_Ename Emp.Ename%TYPE;
        V_Job Emp.Job%TYPE;
        E_ManyEmployees EXCEPTION;
        CURSOR EmpCursor IS SELECT MGR, COUNT(*) Tot_Emp FROM Emp
                                WHERE MGR IS NOT NULL GROUP BY MGR;
        BEGIN
        FOR MgrRecord IN EmpCursor
        LOOP
        BEGIN
```

```
        SELECT Ename, Job INTO V_Ename, V_Job FROM Emp
                                        WHERE Empno = MgrRecord.Mgr;
        IF MgrRecord.Tot_Emp > 3 THEN
        RAISE E_ManyEmployees;
        ELSE
        DBMS_OUTPUT.PUT_LINE('Employee, '||V_Ename||' Manages '||MgrRecord.Tot_Emp||'
        Employees.');
        END IF;
        EXCEPTION
        WHEN E_ManyEmployees THEN
        DBMS_OUTPUT.PUT_LINE('Employee, '||V_EName||' Manages Many Employees, Chance
        of decreasing his Performance, Recommend for Extra Allowances or Emoluments.');
        END;
        END LOOP;
        END;
        /
```

Exception Propagation

- Instead of Trapping An EXCEPTION Within The PL/SQL Block We Can Propagate The EXCEPTION To Allow The Calling Environment To Handle It.
- Each Calling Environment Has Its Own Way of Displaying And Accessing Errors.
- If A PL/SQL RAISES An EXCEPTION And The Current Block Does Not Have A Handle Then The EXCEPTION PROPAGATES into Successive Enclosing Blocks Until It Finds A Corresponding Handle.

Illustration 1

```
SQL> DECLARE
        V_TestVariable CHAR(5) := '&String';
        BEGIN
        DBMS_OUTPUT.PUT_LINE ('This is A Test Line.');
        DBMS_OUTPUT.PUT_LINE(V_TestVariable);
        EXCEPTION
        WHEN INVALID_NUMBER OR VALUE_ERROR THEN
        DBMS_OUTPUT.PUT_LINE('An Error Raised.');
        END;
```

Converting Propagation into Trapping

```
SQL> BEGIN
                DECLARE
                V_TestVariable CHAR(5) := '&String';
                BEGIN
                DBMS_OUTPUT.PUT_LINE ('This is A Test Line.');
                DBMS_OUTPUT.PUT_LINE(V_TestVariable);
                EXCEPTION
                WHEN INVALID_NUMBER OR VALUE_ERROR THEN
                DBMS_OUTPUT.PUT_LINE('An Error Raised.');
                END;
        EXCEPTION
        WHEN INVALID_NUMBER OR VALUE_ERROR THEN
        DBMS_OUTPUT.PUT_LINE('An Error Raised.');
        END;
```

Re-Raising An Exception

- It is A Situation Where We Want To Handle An Exception in The Inner Block And Pass It To The Outer Block.
- In The Outer Block The Same Exception is Raised Once Again.

- When An EXCEPTION is Re-Raised Then We Should Not Supply The EXCEPTION Name Once Again Along With The RAISE Statement.
- Re-Raising of EXCEPTION is Very Rare But, It is Very Important As Per The EXCEPTION Concepts.

Illustration

```
SQL> DECLARE
        E_Exception EXCEPTION;
        BEGIN
        RAISE E_Exception;
        EXCEPTION
        WHEN E_Exception THEN
        RAISE;
        END;
        EXCEPTION
        WHEN E_Exception THEN
        DBMS_OUTPUT.PUT_LINE('An Error Occurred');
        END;
```

RAISE_APPLICATION_ERROR Procedure

- RAISE APPLICATION_ERROR is A Procedure That Lets Us To Issue USER_DEFINED Error Messages From Stored Sub Programs.
- It is Used To Communicate A Predefined Exception Interactively By Returning A Non Standard Error Code And Error Message.
- Using This Procedure We Can Report Error To Application And Avoid Returning Unhandled Exceptions.

Syntax:

```
RAISE_APPLICATION_ERROR(ErrorNumber, Message [,{TRUE|FALSE}]);
```

- ErrorNumber is A User Specified Number For The Exception Between -20000 And -20999.
- Message is The User Specified Message For The Exception. It Can Be Up To 2048 Bytes.
- TRUE/FALSE is Optional Boolean Parameters, If TRUE The Error is Placed On The Stack of Previous Error. If FALSE By Default The Error Replaces All Previous Errors.

```
SQL> BEGIN
            DECLARE
            V_Deptno Dept.Deptno%TYPE := &Deptno;
            V_TotEmp NUMBER;
            E_InvalidDept EXCEPTION;
            BEGIN
            IF V_Deptno NOT IN (10, 20, 30, 40) THEN
            RAISE E_InvalidDept;
            ELSE
            SELECT COUNT(*) INTO V_TotEmp FROM Emp WHERE Deptno = V_Deptno;
            DBMS_OUTPUT.PUT_LINE('The Total Employees in '||V_Deptno||' are
                                                        '||V_TotEmp||'.');
            END IF;
            DBMS_OUTPUT.PUT_LINE('No Exception was Raised...');
            EXCEPTION
            WHEN E_InvalidDept THEN
            RAISE_APPLICATION_ERROR(-20220, 'Sorry There is no Such Department...as
                                                    You requested.');
            END;
        EXCEPTION
        WHEN OTHERS THEN
        RAISE_APPLICATION_ERROR(-20220, 'Sorry Fatal Error.');
        END;
```

Modular Programming Through Sub-Programmed Blocks

Working With Procedures

- Benefits of Modular Coding
  - It is More Reusable.
  - It is More Manageable.

Block Structure

- Block Structure is Common For All Modules Designed And Developed in PL/SQL.
- Every PL/SQL Block Begins With A
- Header (For Named Block Only)
  - Name of The Module
  - The Parameter List If Used
- The Declaration Section, Consisting of
  - Local Variable Declarations.
  - Explicit Cursor Declarations.
  - Sub Blocks.
- The Main Part of The Module is The Execution Section
  - Contains All Calculations And Processing Logic.
  - IF...THEN...ELSE Statements.
  - Iterative Loops.
  - CASE Statements.
  - Data Manipulation Statements.
  - Calls To Other PL/SQL Modules.
- The Last Section is EXCEPTION Handling.

Procedures in PL/SQL

- A Procedure is A Module Performing One OR More Actions Associated To Business Logic.
- A Procedure Does Not Need To Return Any Values.

Syntax

```
CREATE OR REPLACE PROCEDURE  ProcedureName(ParName1 [MODE] ParType,...)
                                        IS
        Local Variable Declaration;
BEGIN
        Executable Statements;
EXCEPTION
        Exception Handlers;
END ProcedureName;
```

- PROCEDURE Can Have 0 OR Many Parameters.
- Every PROCEDURE Consists of Two Parts
  - The Header of The Procedure.
  - The Body of The Procedure.

The Header

- It Comes Before The AS Keyword.
- It Contains The PROCEDURE NAME And The PARAMETER LIST.

The Body

- It is Any Thing OR Every Thing That is Existing After The AS Keyword.
- The Word REPLACE is Optional.

```
SQL> CREATE OR REPLACE PROCEDURE MyBonus
                AS
        CURSOR DeptCursor IS SELECT Deptno FROM Dept;
        BEGIN
        FOR R_GroupBonus IN DeptCursor LOOP
                UPDATE Emp SET Sal = Sal * 0.95 WHERE Deptno = R_GroupBonus.DeptNo;
        DBMS_OUTPUT.PUT_LINe('The Bonus Information is '||R_GroupBonus.Deptno);
```

```
        END LOOP;
        END MyBonus;
```

- To Execute A Procedure We Can Follow Any One of The Following Method.
- Use EXECUTE Procedure in SQL*Plus

```
SQL> EXECUTE MyBonus;
```

- Call The Procedure in Another PL/SQL Block

```
SQL> Begin
        MyBonus;
        END;
```

- The Information Upon PROCEDURES is Provided in
  - USER_OBJECTS View.
  - USER_SOURCE View.

To See The Procedure Availability

```
SQL> SELECT  Object_Name,  Object_Type,  Status FROM USER_OBJECTS
                                        WHERE OBJECT_NAME = 'MYBONUS;
```

To See The Procedure Source Code

```
SQL> SELECT  TO_CHAR(Line, 99)||'>', Text  FROM USER_SOURCE
                                        WHERE  NAME = 'MYBONUS';
```

Recompile An Existing Procedure

Syntax

```
SQL> ALTER PROCEDURE ProcedureName Compile;
```

Illustration

```
SQL> ALTER Procedure MyBonus Compile;
```

Illustrative Examples

```
SQL> CREATE OR REPLACE PROCEDURE GetEnameSalJob00 ( PEmpno Emp.Empno%TYPE)
                                        AS
        V_Ename      Emp.Ename%TYPE;
        V_Sal        Emp.Sal%TYPE;
        V_Job        Emp.Job%TYPE;
        BEGIN
        SELECT Ename, Sal, Job  INTO V_Ename, V_Sal, V_Job FROM Emp
                                                WHERE Empno = PEmpno;
        DBMS_OUTPUT.PUT_LINE('The Details of Employee '||PEmpno||' Are...');
        DBMS_OUTPUT.PUT_LINE('The Name of The Employee is : '||V_Ename);
        DBMS_OUTPUT.PUT_LINE('The Salary of The Employee is : '||V_Sal);
        DBMS_OUTPUT.PUT_LINE('The Job of The Employee is : '||V_Job);
        END GetEnameSalJob00;
        /

SQL> CREATE OR REPLACE PROCEDURE EmpInsert ( P_Empno Emp.Empno%TYPE,
                P_Ename Emp.Ename%TYPE,  P_Sal Emp.Sal%TYPE,
                P_Deptno Emp.Deptno%TYPE, P_Job  Emp.Job%TYPE,
                P_Comm       Emp.Comm%TYPE, P_HireDate Emp.HireDate%TYPE,
                P_MGR        Emp.MGR%TYPE  )
          AS
        BEGIN
        INSERT INTO Emp (Empno, Ename, Sal, Deptno, Job, Comm, HireDate, MGR)
              VALUES(P_Empno, UPPER(P_Ename), P_Sal, P_Deptno, UPPER(P_Job),
                                        P_Comm, P_HireDate, P_MGR);
```

```
COMMIT;
Exception
WHEN DUP_VAL_ON_INDEX THEN
RAISE_APPLICATION_ERROR(-20001, 'Employee already exists');
WHEN OTHERS THEN
RAISE_APPLICATION_ERROR(-20011, SQLERRM);
END;
/
```

Calling The Above Procedure

```
SQL> EXECUTE EmpInsert(1234, 'SAMPLE', 2000, 20, 'CLERK', NULL, SYSDATE, 7566);
SQL> CREATE OR REPLACE PROCEDURE DeptDataDisplay AS
        V_RowCount NUMBER(4);
        TYPE GenericCursor IS REF CURSOR;
        V_GenericCursor GenericCursor;
        TYPE TablesRecordType IS RECORD ( DeptRecord Dept%ROWTYPE );
        V_DeptRecordType TablesRecordType;
      BEGIN
      OPEN V_GenericCursor  FOR  SELECT * FROM Dept;
      DBMS_OUTPUT.PUT_LINE(RPAD(LPAD('Department Information', 29, '*'), 49, '*'));
      DBMS_OUTPUT.PUT_LINE(RPAD('-', 40,'-'));
      DBMS_OUTPUT.PUT_LINE(RPAD('Deptno', 8)|| RPAD('Dname', 12)||RPAD('Loc', 12));
      DBMS_OUTPUT.PUT_LINE(RPAD('-', 40,'-'));
      LOOP
      FETCH V_GenericCursor INTO V_DeptRecordType.DeptRecord;
      EXIT WHEN V_GenericCursor%NOTFOUND;
      V_RoWCount := V_GenericCursor%ROWCOUNT;
      DBMS_OUTPUT.PUT_LINE(RPAD(V_DeptRecordType.DeptRecord.Deptno, 8)
      ||RPAD(V_DeptRecordType.DeptRecord.Dname,
      12)||RPAD(V_DeptRecordType.DeptRecord.Dname, 12));
      END LOOP;
      CLOSE V_GenericCursor;
      DBMS_OUTPUT.PUT_LINE(V_RowCount||' Rows Processed So Far...');
      END DeptDataDisplay;

SQL> CREATE OR REPLACE PROCEDURE EmpBonus
       AS
         CURSOR DeptCursor IS SELECT Deptno FROM Dept;
      BEGIN
      FOR R_GroupBonus IN DeptCursor
      LOOP
      DECLARE
      E_JobNotFound EXCEPTION;
      CURSOR EmpCursor IS
      SELECT * FROM Emp WHERE Deptno = R_GroupBonus.Deptno;
      BEGIN
      FOR R_EmpCursor IN EmpCursor
      LOOP
      IF R_EmpCursor.Job = 'PRESIDENT'
      THEN
      UPDATE Emp SET Sal = Sal + (Sal * 0.40) WHERE Empno = R_EmpCursor.Empno;
      DBMS_OUTPUT.PUT_LINE('The Job For Updation is '||R_EmpCursor.Job);
      DBMS_OUTPUT.PUT_LINE('Old Sal : '||R_EmpCursor.Sal);
      DBMS_OUTPUT.PUT_LINE('Increament Added : '||((R_EmpCursor.Sal +
```

```
                                (R_EmpCursor.Sal * 0.40)) - (R_EmpCursor.Sal)));
DBMS_OUTPUT.PUT_LINE('The Salary With Increment is : '||(R_EmpCursor.Sal +
                                (R_EmpCursor.Sal * 0.40)));
ELSE
IF R_EmpCursor.Job = 'MANAGER'
THEN
UPDATE Emp SET Sal = Sal + (Sal * 0.35) WHERE Empno = R_EmpCursor.Empno;
DBMS_OUTPUT.PUT_LINE('The Job For Updation is '||R_EmpCursor.Job);
DBMS_OUTPUT.PUT_LINE('Old Sal : '||R_EmpCursor.Sal);
DBMS_OUTPUT.PUT_LINE('Increament Added : '||((R_EmpCursor.Sal +
(R_EmpCursor.Sal * 0.40)) - (R_EmpCursor.Sal)));
DBMS_OUTPUT.PUT_LINE('The Salary With Increment is : '||(R_EmpCursor.Sal +
                        (R_EmpCursor.Sal * 0.40)));
IF R_EmpCursor.Job = 'ANALYST' THEN
   UPDATE Emp SET Sal = Sal + (Sal * 0.30) WHERE Empno = R_EmpCursor.Empno;
   DBMS_OUTPUT.PUT_LINE('The Job For Updation is '||R_EmpCursor.Job);
   DBMS_OUTPUT.PUT_LINE('Old Sal : '||R_EmpCursor.Sal);
   DBMS_OUTPUT.PUT_LINE('Increament Added : '||((R_EmpCursor.Sal +
                                (R_EmpCursor.Sal * 0.40)) - (R_EmpCursor.Sal)));
   DBMS_OUTPUT.PUT_LINE('The Salary With Increment is : '||(R_EmpCursor.Sal +
                                (R_EmpCursor.Sal * 0.40)));
ELSE
IF R_EmpCursor.Job = 'SALESMAN' THEN
   UPDATE Emp SET Sal = Sal + (Sal * 0.25) WHERE Empno = R_EmpCursor.Empno;
   DBMS_OUTPUT.PUT_LINE('The Job For Updation is '||R_EmpCursor.Job);
DBMS_OUTPUT.PUT_LINE('Old Sal : '||R_EmpCursor.Sal);
DBMS_OUTPUT.PUT_LINE('Increament Added : '||((R_EmpCursor.Sal +
                (R_EmpCursor.Sal * 0.40)) - (R_EmpCursor.Sal)));
DBMS_OUTPUT.PUT_LINE('The Salary With Increment is : '||(R_EmpCursor.Sal +
                (R_EmpCursor.Sal * 0.40)));
ELSE
IF R_EmpCursor.Job = 'CLERK' THEN
UPDATE Emp SET Sal = Sal + (Sal * 0.20) WHERE Empno = R_EmpCursor.Empno;
ELSE
RAISE E_JobNotFound;
DBMS_OUTPUT.PUT_LINE('The Job For Updation is '||R_EmpCursor.Job);
DBMS_OUTPUT.PUT_LINE('Old Sal : '||R_EmpCursor.Sal);
DBMS_OUTPUT.PUT_LINE('Increament Added : '||((R_EmpCursor.Sal +
(R_EmpCursor.Sal * 0.40)) - (R_EmpCursor.Sal)));
DBMS_OUTPUT.PUT_LINE('The Salary With Increment is : '||(R_EmpCursor.Sal +
(R_EmpCursor.Sal * 0.40)));
END IF;
END IF;
END IF;
END IF;
END IF;
END LOOP;
EXCEPTION
WHEN E_JobNotFound THEN
DBMS_OUTPUT.PUT_LINE('The Respective Job is Not Found...Please Verify!');
END;
DBMS_OUTPUT.PUT_LINE('The Bonus Information is '||R_GroupBonus.Deptno);
END LOOP;
END EmpBonus;
```

Passing Parameters IN And OUT of Procedures

- PARAMETERS Are The Mean To Pass Values TO And FROM The Calling Environments To The Oracle Server.
- PARAMETERS Are The Values That Will Be Processed OR Returned Via The EXECUTION of The PROCEDURE.
- A PL/SQL Procedure Can Be of Three Types
  - IN Mode.
  - OUT Mode.
  - INOUT Mode.
- The MODES Upon A Procedure Specify Whether The Parameters Passed In Are READ IN OR A RECEPTACLE For What Comes Out of The Procedure.

FORMAL And ACTUAL Parameters

- FORMAL PARAMETERS Are The Names Specified With in The Parenthesis As Part of The HEADER of The Module.
- ACTUAL PARAMETERS Are The Values, OR Expressions Specified With in Parenthesis As A Parameter List When A Call is Made To The Module.
- The FORMAL PARAMETER And The Related ACTUAL PARAMETER Must Be of The Same Compliable Data Types.
- FORMAL PARAMETERS Do Not Require Constraints Declaration, They Require Only Data Type Declaration But Not The Width.
- The Constraint Upon The Data is Made When The Procedure is Called At Runtime From The Executable Environment.
- There Are Two Types of Methods Match ACTUAL And FORMAL Parameters When Calling The Procedure
  - Positional Notation.
  - Named Notation.

Positional Notation

- It is Simply An Association of The Values By POSITION of The Arguments At Call Time With That of Declaration in The Header of The Procedure Creation.
- The Order of The Parameters Used When Executing The Procedure Should Match The Order in The PROCEDURES HEADER Exactly.

Named Notation

- It is An Explicit Association Using The Symbol =>.

Syntax

FormalParameterName => ArgValue

- In Named Notation The Order of Parameters in Does Not Matter.
- If The Notation is Mixed Then Position Notation Should Be Used First Then The Named Notation Should Be Used.

Mode: IN

- Passes A Value Into The Program, From The Calling Environment.
- It is READ ONLY Value.
- Best Suitable For Constants, Literals, Expressions.
- Cannot Be Changed Within Program.
- It is The Default Mode of Procedures.

Mode: Out

- Passes A Valued Back From The Program To The Calling Environment.
- It is WRITE ONLY Values.
- Cannot Be Assigned Default Values.
- Value Assigned Only If The Program Is Successful.

Mode: INOUT

- Passes Values IN But Also Sends Values Back To The Calling Environments.
- It Has To Be A Variable, And Cannot Be A Constant
- Values Will Be READ From The Calling Environment And Then WRITTEN To The Calling Environment.

```
SQL> CREATE OR REPLACE PROCEDURE  OddNumber(Num1 NUMBER, Num2 NUMBER)
        IS
        MyNum NUMBER(4);
        BEGIN
        MyNum := Num1;
        WHILE MyNum < Num2 LOOP
        IF MOD(MyNum,2) != 0 THEN
        DBMS_OUTPUT.PUT_LINE('The Odd Number : '||MyNum);
        End IF;
        MyNum := MyNum +1;
        END LOOP;
        END;
        /
SQL> CREATE OR REPLACE PROCEDURE FindEmp(I_Empno IN NUMBER,
        O_Ename OUT VARCHAR2, O_Job OUT VARCHAR2)
        AS
        BEGIN
        SELECT Ename, Job INTO O_Ename, O_Job FROM Emp WHERE Empno = I_Empno;
        EXCEPTION
        WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE('Error in Finding the Details of Employee Number : '||
                                                                        I_Empno);
        END FindEmp;
        /
```

Calling Routine For The Above Procedure

```
SQL> DECLARE
        V_Ename Emp.Ename%TYPE;
        V_Job Emp.Job%TYPE;
        BEGIN
        FindEmp(7839, V_Ename, V_Job);
        DBMS_OUTPUT.PUT_LINE('Employee 7839 is : '|| V_Ename||',  '||V_Job||'.');
        END;
        /
SQL> CREATE OR REPLACE PROCEDURE  EmpInfo(I_Deptno IN NUMBER)
        AS
        CURSOR EmpInfoCursor IS SELECT Ename, Job, Sal, Comm FROM Emp
                                                        WHERE Deptno = I_Deptno;
        EmpRecord EmpInfoCursor%ROWTYPE;
        NEmployees NUMBER := 0;
        TSalary NUMBER := 0;
        AVGSalary NUMBER(7,2) := 0;
        MAXSalary NUMBER(7,2) := 0;
        BEGIN
        OPEN EmpInfoCursor;
        LOOP
        FETCH EmpInfoCursor INTO EmpRecord;
        EXIT WHEN EmpInfoCursor%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE('Employee Name :  '||EmpRecord.Ename);
```

```
DBMS_OUTPUT.PUT_LINE('Employee Job : '||EmpRecord.Job);
DBMS_OUTPUT.PUT_LINE('Employee Salary :  '||EmpRecord.Sal);
DBMS_OUTPUT.PUT_LINE('Employee Comission : '||EmpRecord.Comm);
DBMS_OUTPUT.PUT_LINE('**********************************');
TSalary := TSalary + EmpRecord.Sal;
NEmployees := NEmployees + 1;
IF EmpRecord.Sal > MAXSalary THEN
MAXSalary := EmpRecord.Sal;
END IF;
END LOOP;
AVGSalary := TSalary / NEmployees;
DBMS_OUTPUT.PUT_LINE('Number of Employees :  '||NEmployees);
DBMS_OUTPUT.PUT_LINE('Total Salary :  '||TSalary);
DBMS_OUTPUT.PUT_LINE('Maximum Salary :  '||MAXSalary);
DBMS_OUTPUT.PUT_LINE('Average Salary :  '||AVGSalary);
CLOSE EmpInfoCursor;
END EmpInfo;
/
```

Working With User Defined Functions

- Functions in PL/SQL Are Stored Code And Are Similar To PROCEDURES.
- The FUNCTION is A PL/SQL Block That Returns A Single Value.
- FUNCTION Can Accept One OR Many OR No Parameters But A FUNCTION Must Have A RETURN Clause in The EXECUTABLE Section of A FUNCTION.
- The Data Type of The Return Values Must Be Declared in The Header of The Function.
- A Function is Not A Stand Alone Executable Module As A Procedure. It Has To Be Used in Some Context of Implementation.
- A Function Has Output That Needs To Be Assigned To A Variable OR It Can Be Used in A SELECT Statement.
- A FUNCTION Should RETURN Value For All The Varying Possible Execution Streams.
- The RETURN Statement Need Not Appear As The Last Line of The Main Execution Section.
- A FUNCTION Can Contain More Than One RETURN Statement, Each Exception Should Have A RETURN Statement.
- The Parameters of A Function Can Be Either IN, OUT OR INOUT Mode.

Syntax

```
CREATE OR REPLACE FUNCTION FuncName(Parameter List) RETURN DataType
IS
Local Variables
BEGIN
<Body>
RETURN(Return Value);
EXCEPTION
<Defined Pragmas>
END;
```

```
SQL> CREATE OR REPLACE FUNCTION Factorial(Num NUMBER)
        RETURN NUMBER
            IS
        Fact NUMBER(4) := 1;
        BEGIN
        FOR MyIndex IN REVERSE 1..Num
        LOOP
        Fact := Fact * MyIndex;
```

```
        END LOOP;
        RETURN Fact;
        END;
        /
```

Calling Routine For The Above Function

```
SQL> DECLARE
        V_Factorial NUMBER(4) := 0;
        BEGIN
        V_Factorial := Factorial(5);
        DBMS_OUTPUT.PUT_LINE('The Factorial is : '||V_Factorial);
        END;
        /

SQL> CREATE OR REPLACE FUNCTION  Combination(Num1 NUMBER, Num2 NUMBER)
        RETURN NUMBER  IS
        Combi NUMBER(4,2) := 1;
        BEGIN
        Combi := (Factorial(Num1) / (Factorial(Num1 - Num2) * Factorial(Num2)));
        RETURN Combi;
        END;
        /

SQL> CREATE OR REPLACE FUNCTION  EmpExp(V_Empno NUMBER) RETURN NUMBER
                                                IS
        V_HireDate Emp.HireDate%TYPE;
        V_Exp NUMBER(4,2) := 1;
        BEGIN
        SELECT HireDate INTO V_HireDate FROM Emp WHERE Empno = V_Empno;
        V_Exp := MONTHS_BETWEEN(SYSDATE,V_HireDate) / 12;
        RETURN V_Exp;
        END;
        /
SQL> CREATE OR REPLACE PROCEDURE
        EmpInfo(I_Deptno IN NUMBER)
        AS
        CURSOR EmpInfoCursor IS SELECT Empno, Ename, Job, Sal, Comm FROM Emp
                                                WHERE Deptno = I_Deptno;
        EmpRecord EmpInfoCursor%ROWTYPE;
        NEmployees NUMBER := 0;
        TSalary NUMBER := 0;
        AVGSalary NUMBER(7,2) := 0;
        MAXSalary NUMBER(7,2) := 0;
        V_Exp NUMBER(4,2) := 1;
        BEGIN
        OPEN EmpInfoCursor;
        LOOP
        FETCH EmpInfoCursor INTO EmpRecord;
        EXIT WHEN EmpInfoCursor%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE('Employee Name :  '||EmpRecord.Ename);
        DBMS_OUTPUT.PUT_LINE( 'Employee Job : '||EmpRecord.Job);
        DBMS_OUTPUT.PUT_LINE('Employee Salary :  '||EmpRecord.Sal);
        DBMS_OUTPUT.PUT_LINE('Employee Comission : '||EmpRecord.Comm);
        V_Exp := EmpExp(EmpRecord.Empno);
        DBMS_OUTPUT.PUT_LINE('Employee"s Experiance : '||V_Exp);
```

```
DBMS_OUTPUT.PUT_LINE('************************************');
TSalary := TSalary + EmpRecord.Sal;
NEmployees := NEmployees + 1;
IF EmpRecord.Sal > MAXSalary THEN
MAXSalary := EmpRecord.Sal;
END IF;
END LOOP;
AVGSalary := TSalary / NEmployees;
DBMS_OUTPUT.PUT_LINE('Number of Employees : '||NEmployees);
DBMS_OUTPUT.PUT_LINE('Total Salary : '||TSalary);
DBMS_OUTPUT.PUT_LINE('Maximum Salary : '||MAXSalary);
DBMS_OUTPUT.PUT_LINE('Average Salary : '||AVGSalary);
CLOSE EmpInfoCursor;
END EmpInfo;
/
```

Working With Packages

- Package is A Method To Handle All The Required Functions And Procedures Together.
- A Well Designed Package is A Logical Grouping of Objects Such As
  - Functions.
  - Procedure.
  - Global Variables.
  - Cursors.
- All of The Code is Loaded on The First Call of The PACKAGE.
- The First Call To The Package is Very Expensive But All Subsequent Calls Will Result in An Improved Performance.
- Packages Should Be Taken Very Seriously in All Such Applications Where Procedures And Functions Are Used Repeatedly.
- Packages Does Not Provide Any Additional Level of Security.
- The Structure of A PACKAGE Encourages The TOP-DOWN Approach.
- Every Package Has Two Parts, Which Are Individually Coded.
  - Package Specification.
  - Package Body.

Package Specification

- The PACKAGE SPECIFICATION Contains Information About The Contents of The PACKAGE.
- PACKAGE SPECIFICATION Contains Declarations of GLOBAL/PUBLIC Variables.
- Anything Placed in The Declarative Section of A PL/SQL Block Can Be Coded in The PACKAGE SPECIFICATION.
- All Objects Placed in The PACKAGE SPECIFICATION Are Called PUBLIC OBJECTS.
- Any Function OR Procedure Not in The Package Specification, But Coded in The PACKAGE BODY is Called A Private Function OR Procedure.

```
SQL> CREATE OR REPLACE PACKAGE EmpManagement
        AS
        PROCEDURE EmpFind (I_Empno IN Emp.Empno%TYPE,
                           O_Ename OUT Emp.Ename%TYPE,
                           O_Job OUT Emp.Job%TYPE);
        FUNCTION EmpExp (V_empno NUMBER)
                    RETURN NUMBER;
        END EmpManagement;
```

The Package Body

- The PACKAGE BODY Contains The Actual Executable Code For The OBJECTS Described in The PACKAGE SPECIFICATION.

- The PACKAGE BODY Contain Code For All Procedures And Function Described in The Specification And May Additionally Contain Code For Object Not Declared in The Specification.
- When Creating Stored Package The Package Specification And Body Can Be Compiled Separately.

Rules For Package Body

- There Must Be An Exact Match Between The CURSOR And MO DULE HEADERS And The Definition is The PACKAGE SPECIFICATION.
- We Should Not Repeat The Declaration of Variables, Exceptions, TYPE OR Constants in The Specification Again in The Body.
- Any Element Declared in The Specifications Can Be Reused in The Body.

Referencing Package Elements

- When An Element is Called From Outside The Package It is Referenced As PackageName.Element
- We Need Not Qualify Elements When Declared And Referenced Inside The Body of The Package OR When Declared In A Specification And Referenced Inside The Body of The Same Package.

```
SQL> CREATE OR REPLACE PACKAGE MathsBody
        IS
        FUNCTION Factorial(Num NUMBER) RETURN NUMBER;
        FUNCTION Combination(Num1 NUMBER, Num2 NUMBER) RETURN NUMBER;
        PROCEDURE ProdSeries(StartRange NUMBER);
        PROCEDURE PrintEvenOdd(Num1 NUMBER, Num2 NUMBER);
        PROCEDURE SquareArea(Side IN NUMBER);
        PROCEDURE CubeVolume(Radius IN NUMBER);
        END MathsBody;
        /

SQL> CREATE OR REPLACE PACKAGE BODY MathsBody
        AS
        FUNCTION Factorial(Num NUMBER) RETURN NUMBER
        IS
        Fact NUMBER(4) := 1;
        BEGIN
                FOR MyIndex IN REVERSE 1..Num
                LOOP
                        Fact := Fact * MyIndex;
                END LOOP;
                RETURN Fact;
        END Factorial;
        FUNCTION Combination(Num1 NUMBER, Num2 NUMBER) RETURN NUMBER
        IS
                Combi NUMBER(4,2) := 1;
        BEGIN
                Combi := (Factorial(Num1) /( Factorial(Num1-Num2) * Factorial(Num2)));
                RETURN Combi;
        END Combination;
        PROCEDURE ProdSeries(StartRange NUMBER)
        IS
        Result NUMBER;
        BEGIN
                FOR MyIndex IN 1..5 LOOP
                Result := StartRange * MyIndex;
```

```
        DBMS_OUTPUT.PUT_LINE(StartRange||' X '||MyIndex||' = '|| Result);
    END LOOP;
    END ProdSeries;
    PROCEDURE PrintEvenOdd(Num1 NUMBER, Num2 NUMBER)
    IS
    V_Num1 NUMBER;
    EvenValue VARCHAR2(1000);
    OddValue VARCHAR2(1000);
    BEGIN
        V_Num1 := Num1;
        WHILE V_Num1 < Num1
        LOOP
        IF MOD(V_Num1,2) != 0 THEN
            OddValue := OddValue||' '||V_Num1;
        ELSE
            EvenValue := EvenValue||' '||V_Num1;
        END IF;
        V_Num1 := V_Num1 + 1;
        END LOOP;
    DBMS_OUTPUT.PUT_LINE('The Odd Numbers in The Series are : '||OddValue);
    DBMS_OUTPUT.PUT_LINE('The Even Numbers in The Series are : '||EvenValue);
    END PrintEvenOdd;
    PROCEDURE SquareArea(Side IN NUMBER)
    IS
    BEGIN
        DBMS_OUTPUT.PUT_LINE('Area of The Square = ' || (Side * Side) );
    END;
    PROCEDURE CubeVolume(Radius IN NUMBER)
    IS
    BEGIN
    DBMS_OUTPUT.PUT_LINE('Volume of The Cube = ' || (Radius * Radius * Radius));
    END;
    END MathsBody;
    /
SQL> CREATE OR REPLACE PACKAGE EmpPackage
    IS
    PROCEDURE MyBonus;
    PROCEDURE FindEmp(I_Empno IN NUMBER, O_Ename OUT VARCHAR2,
                      O_Job OUT VARCHAR2);
    PROCEDURE EmpInfo(I_Deptno IN NUMBER);
    FUNCTION EmpExp(V_Empno NUMBER) RETURN NUMBER;
    FUNCTION EmpGrade(I_Grade NUMBER) RETURN VARCHAR2;
    END EmpPackage;

SQL> CREATE OR REPLACE PACKAGE BODY EmpPackage
    IS
    PROCEDURE MyBonus
    AS
    CURSOR DeptCursor IS SELECT Deptno FROm Dept;
    BEGIN
    FOR R_GroupBonus IN DeptCursor LOOP
    UPDATE Emp SET Sal = Sal * 0.95 WHERE Deptno = R_GroupBonus.DeptNo;
    DBMS_OUTPUT.PUT_LINe('The Bonus Information is '||R_GroupBonus.Deptno);
    END LOOP;
    END MyBonus;
```

```
PROCEDURE FindEmp (I_Empno IN NUMBER,  O_Ename OUT VARCHAR2,
                        O_Job OUT VARCHAR2)
AS
BEGIN
SELECT Ename, Job INTO O_Ename, O_Job FROM Emp WHERE Empno = I_Empno;
EXCEPTION
WHEN NO_DATA_FOUND THEN
DBMS_OUTPUT.PUT_LINE('Error in Finding the Details of Employee Number : '||
                                                                    I_Empno);
END FindEmp;
PROCEDURE EmpInfo(I_Deptno IN NUMBER)
AS
CURSOR EmpInfoCursor IS SELECT Ename, Job, Sal, Comm FROM Emp
                                                    WHERE Deptno = I_Deptno;
EmpRecord EmpInfoCursor%ROWTYPE;
NEmployees NUMBER := 0;
TSalary NUMBER := 0;
AVGSalary NUMBER(7,2) := 0;
MAXSalary NUMBER(7,2) := 0;
BEGIN
OPEN EmpInfoCursor;
LOOP
FETCH EmpInfoCursor INTO EmpRecord;
EXIT WHEN EmpInfoCursor%NOTFOUND;
DBMS_OUTPUT.PUT_LINE('Employee Name :  '||EmpRecord.Ename);
DBMS_OUTPUT.PUT_LINE('Employee Job : '||EmpRecord.Job);
DBMS_OUTPUT.PUT_LINE('Employee Salary :  '||EmpRecord.Sal);
DBMS_OUTPUT.PUT_LINE('Employee Comission : '||EmpRecord.Comm);
DBMS_OUTPUT.PUT_LINE('************************************');
TSalary := TSalary + EmpRecord.Sal;
NEmployees := NEmployees + 1;
IF EmpRecord.Sal > MAXSalary THEN
MAXSalary := EmpRecord.Sal;
END IF;
END LOOP;
AVGSalary := TSalary / NEmployees;
DBMS_OUTPUT.PUT_LINE('Number of Employees :  '||NEmployees);
DBMS_OUTPUT.PUT_LINE('Total Salary :  '||TSalary);
DBMS_OUTPUT.PUT_LINE('Maximum Salary :  '||MAXSalary);
DBMS_OUTPUT.PUT_LINE('Average Salary :  '||AVGSalary);
CLOSE EmpInfoCursor;
END EmpInfo;

FUNCTION EmpExp(V_Empno NUMBER) RETURN NUMBER
IS
V_HireDate Emp.HireDate%TYPE;
V_Exp NUMBER(4,2) := 1;
BEGIN
SELECT HireDate INTO V_HireDate FROM Emp WHERE Empno = V_Empno;
V_Exp := MONTHS_BETWEEN(SYSDATE,V_HireDate) / 12;
RETURN V_Exp;
END EmpExp;

FUNCTION EmpGrade(I_Grade NUMBER)
RETURN VARCHAR2
```

```
IS
V_Num NUMBER(4);
BEGIN
SELECT COUNT(*) INTO V_Num FROM Emp, SalGrade WHERE Sal BETWEEN
                    LoSal AND HiSal AND  Grade = I_Grade;
RETURN 'The Total Employees For The Grade Given By You Are : '||V_Num;
END EmpGrade;
END EmpPackage;
/
```

Working With Triggers

- A DATABASE TRIGGER is A Named PL/SQL Block Stored in A Database And Executed Implicitly When TRIGGERING EVENT OCCURS.
- An Act of Executing A Trigger is Referred To As Firing A Trigger.
- A TRIGGEING EVENT is A DML Statement Associated With "INSERT", "UPDATE" OR "DELETE" Statement Executed Against A DATABASE TABLE.
- A Trigger Can Fire BEFORE OR AFTER A Triggering Event.
- A Trigger is Auto Executable Programmed Block, Waiting For The Event To Take Place.

Syntax

```
CREATE OR REPLACE TRIGGER TriggerName
{BEFORE OR AFTER} EventType ON TableName
[FOR EACH ROW]
[WHEN Condition]
DECLARE
        Declaration Statements;
BEGIN
        Executable Statements;
EXCEPTION
        Exception Handling;
END;
```

- When A Database Table is Dropped Then The Associated Triggers Are Also Dropped Automatically.

Advantages

- Enforcing Complex Business Rules That Cannot Be Defined By Using Integrity Constraints.
- Maintaining Complex Security Rules.
- Automatically Generating Values For Derived Columns OR PRIMARY KEY Columns.
- Collecting Statistical Information On Table Access.
- Preventing Invalid Transactions.
- Providing Value Auditing.

Restrictions

- A TRIGGER May Not issue A Transactional Control Statement Like COMMIT, SAVEPOINT And ROLLBACK.
- Any FUNCTION OR PROCEDURE Called By A Trigger Cannot issue A Transaction Control Statement.
- It is Not Permitted To Declare LONG OR LONG RAW Variables in The Body of A Trigger.

Before Triggers

- These TRIGGERS Fire BEFORE Any Transactions Are Implemented
- These TRIGGERS Can Be Classified As
  - BEFORE INSERT.
  - Before UPDATE.
  - Before DELETE.

Usage

- When A Trigger Provides Values For Derived Columns BEFORE The INSERT OR UPDATE Statement is Completed.
- When A Trigger Determines Whether An INSERT, UPDATE OR DELETE Statement Should Be Allowed To Complete.

After Triggers

- These TRIGGERS Fire AFTER Any Transaction is Implemented.
- These Triggers Can Be Classified As
  - After INSERT.
  - After DELETE.
  - After UPDATE.

Usage

- When A TRIGGER Should Fire After A DML Statement is Executed For Acknowledgement Purpose OR Auditing.
- When A Trigger Should Perform Actions Not Specified in A BEFORE Trigger .

```
SQL> CREATE OR REPLACE TRIGGER WeekEndCheck
        AFTER INSERT OR UPDATE OR DELETE ON Emp
        DECLARE
        V_WeekDay VARCHAR2(100);
        BEGIN
        V_WeekDay := TO_CHAR(SYSDATE,'DY');
        IF V_WeekDay = 'SAT' OR V_WeekDay = 'SUN' THEN
        RAISE_APPLICATION_ERROR(-20010,'An Illegal Intrusion into the System was
                                                                    Detected.');
        END IF;
        END;
        /

SQL> CREATE OR REPLACE TRIGGER IllegalTime
        BEFORE INSERT OR UPDATE OR DELETE ON Emp
        DECLARE
        V_Time NUMBER;
        BEGIN
        V_Time := TO_CHAR(SYSDATE, 'HH24');
        IF V_Time NOT BETWEEN 10 AND 17 THEN
        RAISE_APPLICATION_ERROR(-20011,' Illegal Intrusion, Not Business Hours...');
        END IF;
        END;
        /
```

Instead of Triggers

- The INSTEAD OF TRIGGERS Are Special Kind That Are Defined On DATABASE Views.
- The INSTEAD OF TRIGGER is Created As ROW TRIGGERS Only.
- The INSTEAD OF TRIGGER Fires Instead of The TRIGGERING STATEMENT, That Has Been Issued Against A View.

Syntax

```
SQL> CREATE OR REPLACE TRIGGER TriggerName
        INSTEAD OF TriggerEvent ON ViewName
        FOR EACH ROW
        BEGIN
        Execution Statement;
        END;

SQL> CREATE VIEW DeptView AS SELECT Deptno, Dname, Loc FROM Dept
```

```
SQL> CREATE OR REPLACE TRIGGER DeptDel
        INSTEAD OF DELETE ON DeptView
        FOR EACH ROW
        BEGIN
        DELETE FROM Dept
        WHERE Deptno = :OLD.Deptno;
        DBMS_OUTPUT.PUT_LINE('Department Deleted...');
        END;
        /
```

Row Triggers

- A Row Triggers is Fired As Many Times As There Are Rows Affected By Triggering Event.
- When The Statement FOR EACH ROW is Present in The CREATE TRIGGER Clause, The Trigger is A ROW Trigger.

Illustration

```
SQL> CREATE OR REPLACE TRIGGER DeptUpdate
        AFTER UPDATE ON Dept
        FOR EACH ROW
```

- The ROW LEVEL TRIGGER Fires As Many Times As The Number of ROWS Are Affected When The Transaction Takes Place.

Statement Level Triggers

- A Statement Trigger is Fired Once For The Entire Triggering Statement.
- This TRIGGER Fires Once, Regardless of The Number of Rows Affected By The TRIGGERING Statement.

Illustration

```
SQL> CREATE  OR REPLACE TRIGGER
        DeptDelete  AFTER DELETE ON Dept
```

- Statement Trigger Should Be Used When The Operations Performed By The Trigger Do Not By On The Data in The Individual Records.

Correlation Identifiers in Row Level

- As A ROW LEVEL TRIGGER Fires Once Per Row Inside The TRIGGER We Can Access The Data In The "ROW THAT IS CURRENTLYTY BEING PROCESSED".
- The Two Correlation Identifiers Provided By PL/SQL Are ":OLD" And ":NEW".
- A Correlation Identifier is A Special Kind of PL/SQL Bind Variable.
- The PL/SQL Compiler Treats The Correlation Identifiers As RECORDS of Type "Triggering_Table%ROWTYPE";

The Specifications

Transaction	:OLD	:NEW
INSERT	Undefined: All Fields Are NULL	Values That Will Be Inserted When The Statement is Complete.
UPDATE	Original Values For The Row Before The Update.	New Values That Will Be Updated When The Statement is Complete.
DELETE	Original Values Before The Row is Deleted	Undefined: All Fields Are NULL

```
SQL> CREATE TABLE RecycleBin( Empno NUMBER(6), Ename VARCHAR2(20),
            Job VARCHAR(20), MGR NUMBER(6), HireDate DATE,Sal NUMBEr(7,2),
            Comm NUMBER(7,2), Deptno NUMBER(2) )
SQL> CREATE OR REPLACE TRIGGER EmpRBin
        BEFORE DELETE  ON Emp
        FOR EACH ROW
        BEGIN
```

```
INSERT INTO RecycleBin VALUES(:OLD.Empno, :OLD.Ename, :OLD.Job, :OLD.MGR,
                              :OLD.HireDate, :OLD.Sal, :OLD.Comm, :OLD.Deptno);
END;
/
```

Some Good Stuff To Practice

Q.1) Display the details of all employees
Q.2) Display the depart information from department table
Q.3) Display the name and job for all the employees
Q.4) Display the name and salary for all the employees
Q.5) Display the employee no and totalsalary for all the employees
Q.6) Display the employee name and annual salary for all employees.
Q.7) Display the names of all the employees who are working in department number 10.
Q.8) Display the names of all the employees who are working as clerks and drawing a salary more than 3000.
Q.9) Display the employee number and name who are earning comm.
Q.10) Display the employee number and name who do not earn any comm.
Q.11) Display the names of employees who are working as clerks, salesman or analyst and drawing a salary more than 3000.
Q.12) Display the names of the employees who are working in the company for the past 5 years;
Q.13) Display the list of employees who have joined the company before 30-jun-90 or after 31-dec-90.
Q.14) Display current date.
Q.15) Display the list of all users in your database(use catalog table).
Q.16) Display the names of all tables from current user;
Q.17) Display the name of the current user.
Q.18) Display the names of employees working in depart number 10 or 20 or 40 or employees working as clerks,salesman or analyst.
Q.19) Display the names of employees whose name starts with alphabet s.
Q.20) Display the employee names for employees whose name ends with alphabet s.
Q.21) Display the names of employees whose names have second alphabet a in their names.
Q.22) Select the names of the employee whose names is exactly five characters in length.
Q.23) Display the names of the employee who are not working as managers.
Q.24) Display the names of the employee who are not working as salesman or clerk or analyst.
Q.25) Display all rows from emp table. the system should wait after every screen full of information.
Q.26) Display the total number of employee working in the company.
Q.27) Display the total salary being paid to all employees.
Q.28) Display the maximum salary from emp table.
Q.29) Display the minimum salary from emp table.
Q.30) Display the average salary from emp table.
Q.31) Display the maximum salary being paid to clerk.
Q.32) Display the maximum salary being paid to depart number 20.
Q.33) Display the minimum salary being paid to any salesman.
Q.34) Display the average salary drawn by managers.
Q.35) Display the total salary drawn by analyst working in depart number 40.
Q.36) Display the names of the employee in order of salary i.e the name of
Q.37) The employee earning lowest salary should salary appear first.
Q.38) Display the names of the employee in descending order of salary.
Q.39) Display the names of the employee in order of employee name.
Q.40) Display empno,ename,deptno,sal sort the output first base on name and
Q.41) Within name by deptno and with in deptno by sal.

Q.42) Display the name of the employee along with their annual salary(sal*12).the name of the employee earning highest annual salary should appear first.

Q.43) Display name,salary,hra,pf,da,total salary for each employee. The output should be in the order of total salary,hra 15% of salary,da 10% of salary,pf 5% salary,total salary will be(salary+hra+da)-pf.

Q.44) Display depart numbers and total number of employees working in each department.

Q.45) Display the various jobs and total number of employees within each job group.

Q.46) Display the depart numbers and total salary for each department.

Q.47) Display the depart numbers and max salary for each department.

Q.48) Display the various jobs and total salary for each job

Q.49) Display the depart numbers with more than three employees in each dept.

Q.50) Display the various jobs along with total salary for each of the jobs where total salary is greater than 40000.

Q.51) Display the various jobs along with total number of employees in each job. the output should contain only those jobs with more than three employees.

Q.52) Display the name of the employee who earns highest salary.

Q.53) Display the employee number and name for employee working as clerk and earning highest salary among clerks.

Q.54) Display the names of salesman who earns a salary more than the highest salary of any clerk.

Q.55) Display the names of clerks who earn a salary more than the lowest salary of any salesman.

Q.56) Display the names of employees who earn a salary more than that of jones or that of salary greater than that of scott.

Q.57) Display the names of the employees who earn highest salary in their respective departments.

Q.58) Display the names of the employees who earn highest salaries in their respective job groups.

Q.59) Display the employee names who are working in accounting department.

Q.60) Display the employee names who are working in Chicago.

Q.61) Display the job groups having total salary greater than the maximum salary for managers.

Q.62) Display the names of employees from department number 10 with salary greater than that of any employee working in other department.

Q.63) Display the names of the employees from department number 10 with salary greater than that of all employee working in other departments.

Q.64) Display the names of the employees in uppercase.

Q.65) Display the names of the employees in lowercase.

Q.66) Display the names of the employees in proper case.

Q.67) Display the length of your name using appropriate function.

Q.68) Display the length of all the employee names.

Q.69) Select name of the employee concatenate with employee number.

Q.70) Use appropriate function and extract 3 characters starting from 2 characters from the following string 'oracle'. I.e the out put should be 'ac'.

Q.71) Find the first occurrence of character 'a' from the following string i.e 'computer maintenance corporation'.

Q.72) Replace every occurrence of alphabet a with b in the string allens(use translate function)

Q.73) Display the information from emp table. where job manager is found it should be displayed as boos(use replace function).

Q.74) Display empno,ename,deptno from emp table. instead of display department numbers display the related department name(use decode function).

Q.75) Display your age in days.

Q.76) Display your age in months.

Q.77) Display the current date as 15th august Friday nineteen ninety seven.

Q.78) Display the following output for each row from emp table.

Q.79) Scott has joined the company on Wednesday 13th august nineteen ninety.

Q.80) Find the date for nearest Saturday after current date.
Q.81) display current time.
Q.82) Display the date three months before the current date.
Q.83) Display the common jobs from department number 10 and 20.
Q.84) Display the jobs found in department 10 and 20 eliminate duplicate jobs.
Q.85) Display the jobs which are unique to department 10.
Q.86) Display the details of those who do not have any person working under them.
Q.87) Display the details of those employees who are in sales department and grade is 3.
Q.88) Display those who are not managers and who are managers to any one.
Q.89) Display the managers names
Q.90) Display the who are not managers
Q.91) Display those employee whose name contains not less than 4 characters.
Q.92) Display those department whose name start with "s" while the location
Q.93) Display employees whose names end with "k".
Q.94) Display those employees whose manager name is jones.
Q.95) Display those employees whose salary is more than 3000 after giving 20% increment.
Q.96) Display all employees while their dept names;
Q.97) Display ename who are working in sales dept.
Q.98) Display employee name,deptname,salary and comm for those sal in between 2000 to 5000 while location is chicago.
Q.99) Display those employees whose salary greter than his manager salary.
Q.100) Display those employees who are working in the same dept where his manager is work.
Q.101) Display those employees who are not working under any manager.
Q.102) Display grade and employees name for the dept no 10 or 30 but grade is not 4 while joined the company before 31-dec-82.
Q.103) Update the salary of each employee by 10% increment who are not eligible for commission.
Q.104) Select those employee who joined the company before 31-dec-82 while their dept location is network or chicago.
Q.105) Display employee name,job,department,location for all who are working as manager?
Q.106) Display those employees whose manager name is jones? [and also display their manager name]?
Q.107) Display name and salary of ford if his salary is equal to hisal of his grade
Q.108) Display employee name,job,depart name ,manager name,his grade and make out an order under department wise?
Q.109) List out all employees name,job,salary,grade and depart name for every
Q.110) One in the company except 'clerk'. sort on salary display the highest salary?
Q.111) Display the employee name,job and his manager. display also employee who are without manager?
Q.112) Find out the top 5 earners of company?
Q.113) Display name of those employee who are getting the highest salary?
Q.114) Display those employee whose salary is equal to average of maximum and minimum?
Q.115) Select count of employee in each department where count greater than 3?
Q.116) Display dname where at least 3 are working and display only department name?
Q.117) Display name of those managers name whose salary is more than average salary of his company?
Q.118) Display those managers name whose salary is more than average salary of his employee?
Q.119) Display employee name,sal,comm and net pay for those employee
Q.120) Whose net pay is greter than or equal to any other employee salary of the company?
Q.121) Display all employees names with total sal of company with each employee name?
Q.122) Find out last 5(least)earners of the company.?
Q.123) Find out the number of employees whose salary is greater than their
Q.124) Manager salary?
Q.125) Display those department where no employee working?
Q.126) Display those employee whose salary is odd value?

Q.127) Display those employee whose salary contains atleast 3 digits?
Q.128) Display those employee who joined in the company in the month of DEC?
Q.129) Display those employees whose name contains "a"?
Q.130) Display those employee whose deptno is available in salary?
Q.131) Display those employee whose first 2 characters from hiredate - last 2 characters of salary?
Q.132) Display those employee whose 10% of salary is equal to the year of joining?
Q.133) Display those employee who are working in sales or research?
Q.134) Display the grade of jones?
Q.135) Display those employees who joined the company before 15 of the month?
Q.136) Display those employee who has joined before 15th of the month.
Q.137) Delete those records where no of employees in a particular department is less than 3.
Q.138) Display the name of the department where no employee working.
Q.139) Display those employees who are working as manager.
Q.140) Display those employees whose grade is equal to any number of sal but not equal to first number of sal?
Q.141) Print the details of all the employees who are sub-ordinate to Blake?
Q.142) Display employee name and his salary whose salary is greater than
Q.143) Highest average of department number?
Q.144) Display the 10th record of emp table(without using rowid)
Q.145) Display the half of the ename's in upper case and remaining lowercase?
Q.146) Display the 10th record of emp table without using group by and rowid?
Q.147) Create a copy of emp table;
Q.148) Select ename if ename exists more than once.
Q.149) Display all enames in reverse order?(smith:htims).
Q.150) Display those employee whose joining of month and grade is equal.
Q.151) Display those employee whose joining date is available in deptno.
Q.152) Display those employees name as follows
Q.153) A → allen
Q.154) B → blake
Q.155) List out the employees ename,sal,pf(20% of sal) from emp;
Q.156) Create table emp with only one column empno;
Q.157) Add the column to emp table with name as ename varachar2(20).
Q.158) Oops i forgot give the primary key constraint. Add it now.
Q.159) Now increase the length of ename column to 30 characters.
Q.160) Add salary column to emp table.
Q.161) I want to give a validation saying that salary cannot be greater 10,000 (note give a name to this constraint)
Q.162) For the time being i have decided that i will not impose this validation. my boss has agreed to pay more than 10,000.
Q.163) My boss has changed his mind. Now he doesn't want to pay more than 10,000.so revoke that salary constraint.
Q.164) Add column called as mgr to your emp table;
Q.165) Oh! This column should be related to empno. Give a command to add this constraint.
Q.166) Add deptno column to your emp table;
Q.167) This deptno column should be related to deptno column of dept table; give the command to add the constraint.
Q.168) Create table called as newemp. Using single command create this table as well as get data into this table(use create table as);
Q.169) Create table called as newemp. This table should contain only empno,ename,dname.
Q.170) Delete the rows of employees who are working in the company for more than 2 years.
Q.171) Provide a commission(10% comm of sal) to employees who are not earning any commission.
Q.172) If any employee has commission his commission should be incremented by 10% of his salary.
Q.173) Display employee name and department name for each employee.

Q.174) Display employee number,name and location of the department in which he is working.
Q.175) Display ename,dname even if there are no employees working in a particular department(use outer join).
Q.176) Display employee name and his manager name.
Q.177) Display the department name and total number of employees in each department.
Q.178) Display the department name along with total salary in each department.
Q.179) Display itemname and total sales amount for each item.
Q.180) Write a query to delete the repeated rows from emp table;
Q.181) Write a query to display 5 to 7 rows from a table
Q.182) Write a query to display top n rows from table?
Q.183) Write a query to display top 3 salaries from emp;
Q.184) Write a query to display 9th from the emp table?
Q.185) Write a query to list all the employees who are working as clerk
Q.186) Write a query to list the employees who are working as clerks or managers with the minimum salary of 4000
Q.187) Write a query o list the employees who are having experience more than 4 years
Q.188) Write a query to list the employees who hired in the last seven days
Q.189) Write a query to list the employees whose salaries are within the range of 5000 and 10000
Q.190) Write a query to list the employees who have joined in the month of march'99
Q.191) Write a query to list the employees whose salaries are not within the range of 10000 and 13000
Q.192) Write a query who are getting 1000, 2000, 3000
Q.193) Write a query to list all employees except who are joined on 11- sep-2001, 31-mar-2001 and 30-jan-2001(it also compares time).
Q.194) Write a query to list all the employees whose names are having 'o' as second character
Q.195) Write a query to list all the employees whose names are having 'r' as last character
Q.196) Write a query to list all the employees whose names are starting with 'r' and ending with 'o'
Q.197) Display those employees who's joining of month and grade is equal?
Q.198) Display the half of the ename's in upper case and the remaining in lower case?
Q.199) Use the variable in a statement which finds all employees who can earn $30,000 a year of more?
Q.200) Find out how many managers are there without listing them?
Q.201) Find out the average salary and average total remuneration for each job type remember salesman earn comm?
Q.202) Find out the job that was filled in the first half of 1983 and same job that was filled during the same period on 1984?
Q.203) Findout all the employees who joined the company before their manager?
Q.204) List out all the employees by name and number along with their managers name and number, also display king who has no manager?
Q.205) Find out the employees who earn the highest salary in each job type. sort in descending sal order?
Q.206) Find out the most recently hired employees in each department. Order by hiredate?
Q.207) Display ename salary and deptno for each employee who earn a salary greater than the average for their department order by deptno?
Q.208) Display the department where there are no employees?
Q.209) Display deptno with highest annual remuneration bill as compensation?
Q.210) Display average salary figure for the department?
Q.211) Display employees who can earn more than lowest salary in department 30?
Q.212) Find employees who can earn more than every employee in department 30?
Q.213) Select department name,deptno and sum of salary?
Q.214) Find all departments which have more than three employees?
Q.215) List lowest paid employees working for each manager? Exclude any
Q.216) Groups where the minimum salary is less than 1000 sort the output by salary?
Q.217) Delete those records from emp table whose deptno not available in dept table?

Q.218) Display name of those emp who are going to retire 31-dec-1999. I the maximum job period is 30 years?
Q.219) Display those emp whose salary is odd value?
Q.220) Display those employees whose salary is contains at least 3 digits?
Q.221) Display those emp who join in the company in the month of DEC?
Q.222) Display those emp who's name contains "a"?
Q.223) Display those emp who's deptno is available in salary?
Q.224) Display those emp who's 10% of salary is equals to the year of joining?
Q.225) Display those emp who joined the company before 15th of the month?
Q.226) Delete those emp who joined company of 10years back from today?
Q.227) Display those emp who are working as manager?
Q.228) Find the errors if any.
1. SELECT ,ename,job,sal salary FROM emp;
2. SELECT * FROM salgrade
3. SELECT empno,ename, sal*12 annual salary FROM emp;
4. SELECT empno,ename,distinct sal, job FROM emp;
5. SELECT empno,ename,sal,job*12 FROM emp;
Q.229) Create a query to display unique jobs from the emp table.
Q.230) Display the name concatenated with the job, separated by a comma and space, and name the column as employees and title.
Q.231) Create a query to display all the data from emp table. Separate each Column by a comma. Name the column the_output.
Q.232) Display all records in following format (for all record) on
Employee<name>works in department <number> and appointed on <date>

Some Common Questions Asked on DBMS Theory

Q.1) What is a database?
A dbms is a complex softwaresystem that is used to manage,store and manipulate data and metadata used to describe the data.
Q.2) What is a key?what are different keys in database?
A key is nothing but a attribute or group of attributes.they are used to perform some specific operation depending on their operation.the keys are classified into primary key,secondary key,alternative keysuper key,candidate key,compound or concatinated or composite key.
Q.3) What is a primary key?
Primary key:an attribute to identify a record uniquely is considered to be primary key.for eg in the student table student_no is the primary key because it can be used to identify unique record or unique student.
Q.4) What is a secondary key?
An attribute used to identify a group of records satisfying a given condition is said to be a secondary key.in the employee table designation is a secondary key because more than one employee can have the same designation.
Q.5) What is a candidate key?
Register no usually alloted in the exams is also unique for each student in that case for identifying a student uniquely either student_no or register_no can be used.here two different candidates are contesting for primary key post.any of them can be selected as primary key.
Q.6) What is an alternate key?
If any one of the candidate keys among the different candidate keys available is selected as primary key then remaining keys are called alternate key.
Q.7) What is a super key?
With primary key if any other attribute is added then that combination is called super key in other words,primary key is the minimum possible super key.in the student table student_no+student_name is one the super key.
Q.8) What is a composite key?
If the primary key is combination of more than one key then it is called the composite key.in the table called marks student_no+subject is the composite key.

Q.9) What is a relation?
A relation consists of a homogeneous set of tuples.

Q.10) What is a table?
It is the representation of a relation having records as rows and attributes as columns.

Q.11) What is an attribute?
An object or entity is characterized by its properties or attributes. in relational database systems attributes corresponds to fields.

Q.12) What is a domain?
The set of allowable value for the attribute is the domain of the attribute.

Q.13) What is a tuple?
Tuples are the members of a relation. an entity type having attributes can be represented by set of these attributes called tuple.

Q.14) What is a selection?
An operation that selects only some of the tuples in the relation is known as selection operation.the selection operation yields a horizontal subset of a given relation.

Q.15) What is a join operation?
The join operation allows the combination of two relations to form a new relation.

Q.16) What are base operations in relational algebra?
Union:
The term of the relation as performed by combining the tuples from one relation with those a second relation to produce a third relation. duplicate tuples are eliminated. the relation must be union compatible.
Difference:
The difference of two relations is a third relation having tuples that occur in the first relation but not in the second relation.
Intersection:
The intersection operation selects the common tuples from the two relations.
Cartesian product:
The Cartesian product of two relations is the concatenation of tuples belonging to the two relations. a new resultant scheme is created consisting of concatenation of all possible combination of tuples.

Q.17) What are different dbms facilities? OR How many types of facilities are provided by a dbms?
1)the data definition facility or data definition language(DDL)
2)the data manipulation facility or data manipulation language(DML)
3)the data control facility(DCL)

Q.18) What is data definition language?
Data scheme is specified by a set of definitions which are expressed b a special language called a DDL.

Q.19) What is a data directory or data dictionary?
The result of compilation of DDL statements is a set of tables which are stored in a special file called data dictionary or data directory.
A data directory is a file that contains metadata i.e data about data. this file is consulted before actual is read or modified in the database system.

Q.20) What is a DML?
A DML is a language that enables users to access or manipulate data as organised by the appropriate data mobel.there are basically two types:
1)procedural DML require a user to specify what data is needed and how to get it.
2)non procedural dimly require a user to specify what data is needed without specifying how to get it.

Q.21) What is a query?
A query is a statement requesting the retrieval of information.

Q.22) What is a query language?
The portion of dimly that involves information retrieval is called a query language.

Q.23) What are the advantages of dbms?

Reduction of redundancies,integrity,security,conflict resolution,data independence,shared data,data quality enhanced.

Q.24) What is a SQL?
Structured query language(SQL) originated in 1974 at IBM. SQL was the data definition and manipulation language.

Q.25) What are the features of SQL?
Portability, client server architecture,dynamic data definition,multiple views of data,complete data base language,interactive,high level structure and SQL standards.

Q.26) How SQL organizes the data?
SQL organizes data as DATABASES,TABLES,INDEXES,VIEWS.

Q.27) What is data definition?
SQL lets a user to define the data structure and relationship at the stored data.

Q.28) What is data retrieval?
Allows a user or an application program to retrieve the stored data.

Q.29) What is data sharing?
Data can be shared by more than one user.

Q.30) What are data manipulation operations?
Remove,append,create,delete.

Q.31) Data definition is done through which statement?
Data definition in SQL is via the create statement. the statement can be used to create a table, index or view.

Q.32) What is the command to alter the structure of the table?
The definition of the existing relation can be altered by using alter statement. this statement allows a new column to be added to an existing relation.

Q.33) What is a view?
It is an object of SQL. a query can be defined,storedand named. this is called view.

Q.34) What is a first normal form?
A relation which contains no mulivalued attributes.

Q.35) What is a second normal form?
A relation is in second normal form for if it is first normal form and every nonkey attribute is fully functionally dependent on primary key.

Q.36) What is a third normal form?
A relation is in third normal form if for every functional dependency
f : x->y is a Key.

Q.37) What is BCNF?
Boyce-code normal form.

Q.38) What is fifth normal form?
A relation which eliminates join dependencies.

Q.39) What is the command to delete a record in the table?
DELETE.

Q.40) What is the command to delete a table?
DROP Table.

Q.41) What is the command to insert a record?
INSERT INTO.

Q.42) What is the command to alter table values in SQL?
UPDATE.

Q.43) What is time stamping?
In the time stamping based method, a serial order is created among the concurrent transactions by assigning to each transaction a unique nondecreasing numbers. you will be allocating fixed time for each transaction.

Q.44) What is data base schema?
It is the description of the database i.e its data structure and not the detail.

Q.45) What is a self join?
Joining the table to the same table.

Q.46) What are the different aggregate functions in SQL?

AVG(), MIN(), MAX(), COUNT(), SUM().

Q.47) What is data integrity?
Data must satisfy the integrity constraints of the system for Data Quality.

Q.48) What is data independence?
A database system keeps data separate from software data structures.

Q.49) What is dead locking?
It is the situation where two transactions are waiting for other to release a lock on an item.

Q.50) what is decryption?
Taking encoded text and converting it into text that you are able to read.

Q.51) What is a distributed database?
A database in which the data is contained with in a number of separate subsystems usually in different locations.

Q.52) What is an entity?
It represents a real world object.

Q.53) What is a conceptual data model?
A conceptual data model is concerned with the general description of the data base without concern for how the data may be organised.

Q.54) What is two phase locking?
It is a most common mechanism that is used to control concurrency in two phases for achieving the serializability. the two phases are growing and shrinking.
1)a transaction acquires locks on data items it will need to complete the transaction. this is called growing process. a transaction may obtain lock but may not release any lock.
2)one lock is released no other lock may be acquired. this is called shrinking process. a transaction may release locks but may not obtain any new locks.

Q.55) What is projection?
The projection of a relation is defined as projection of all its tuples over a set of attributes. it yields vertical subset of the relation. the projection operation is used to View the number of attributes in the resultant relation or to reorder attributes.

Q.56) What is encryption?
Encryption is the coding or scrambling of data so that humans can not read them directly.

Q.57) What is cardinality?
The no of instances of each entity involved in an instance of a relation of a relation ship describe how often an entity can participate in relation ship.(1:1,1:many,many:many)

Attempt These Concepts

Q.1) Which is the subset of SQL commands used to manipulate Oracle Database structures, including tables?
Q.2) What operator performs pattern matching?
Q.3) What operator tests column for the absence of data?
Q.4) Which command executes the contents of a specified file?
Q.5) What is the parameter substitution symbol used with INSERT INTO command?
Q.6) Which command displays the SQL command in the SQL buffer, and then executes it?
Q.7) What are the wildcards used for pattern matching?
Q.8) State true or false. EXISTS, SOME, ANY are operators in SQL.
Q.9) State true or false. !=, <>, ^= all denote the same operation.
Q.10) What are the privileges that can be granted on a table by a user to others?
Q.11) What command is used to get back the privileges offered by the GRANT command?
Q.12) Which system tables contain information on privileges granted and privileges obtained?
Q.13) Which system table contains information on constraints on all the tables created?
Q.14) TRUNCATE TABLE EMP;
Q.15) DELETE FROM EMP;
Q.16) Will the outputs of the above two commands differ?
Q.17) What is the difference between TRUNCATE and DELETE commands?
Q.18) What command is used to create a table by copying the structure of another table?

Q.19) What will be the output of the following query?
SELECT REPLACE(TRANSLATE(LTRIM(RTRIM('!! ATHEN !!','!'), '!'), 'AN', '**'),'*','TROUBLE') FROM DUAL;
Q.20) What will be the output of the following query?
SELECT DECODE(TRANSLATE('A','1234567890','1111111111'), '1','YES', 'NO');
Q.21) What does the following query do?
SELECT SAL + NVL(COMM,0) FROM EMP;
Q.22) Which date function is used to find the difference between two dates?
Q.23) Why does the following command give a compilation error?
Q.24) What is the advantage of specifying WITH GRANT OPTION in the GRANT command?
Q.25) What is the use of the DROP option in the ALTER TABLE command?
Q.26) What is the value of 'comm' and 'sal' after executing the following query if the initial value of 'sal' is 10000?
Q.27) UPDATE EMP SET SAL = SAL + 1000, COMM = SAL*0.1;
Q.28) What is the use of DESC in SQL?
Q.29) What is the use of CASCADE CONSTRAINTS?
Q.30) When this clause is used with the DROP command, a parent table can be dropped even when a child table exists.
Q.31) Which function is used to find the largest integer less than or equal to a specific value?
Q.32) What is the output of the following query?
SELECT TRUNC(1234.5678,-2) FROM DUAL;

Study The Following SCHEMAS
Table 1: STUDIES
PNAME (VARCHAR), SPLACE (VARCHAR), COURSE (VARCHAR), CCOST (NUMBER)
Table 2 : SOFTWARE
PNAME (VARCHAR), TITLE (VARCHAR), DEVIN (VARCHAR), SCOST (NUMBER), DCOST (NUMBER), SOLD (NUMBER)
Table 3 : PROGRAMMER
PNAME (VARCHAR), DOB (DATE), DOJ (DATE), SEX (CHAR), PROF1 (VARCHAR), PROF2 (VARCHAR), SAL (NUMBER)
LEGEND
PNAME – Programmer Name, SPLACE – Study Place, CCOST – Course Cost, DEVIN– Developed in, SCOST – Software Cost, DCOST – Development Cost, PROF1 –Proficiency 1

Attempt The Following Queries

Q.1) Find out the selling cost average for packages developed in Oracle.
Q.2) Display the names, ages and experience of all programmers.
Q.3) Display the names of those who have done the PGDCA course.
Q.4) What is the highest number of copies sold by a package?
Q.5) Display the names and date of birth of all programmers born in April.
Q.6) Display the lowest course fee.
Q.7) How many programmers have done the DCA course.
Q.8) How much revenue has been earned through the sale of packages developed in C.
Q.9) Display the details of software developed by Rakesh.
Q.10) How many programmers studied at Pentafour.
Q.11) Display the details of packages whose sales crossed the 5000 mark.
Q.12) 12. Find out the number of copies which should be sold in order to recover the development cost of each package.
Q.13) 13. Display the details of packages for which the development cost has been recovered.
Q.14) What is the price of costliest software developed in VB?
Q.15) How many packages were developed in Oracle ?
Q.16) How many programmers studied at PRAGATHI?
Q.17) How many programmers paid 10000 to 15000 for the course?

Q.18) What is the average course fee?
Q.19) Display the details of programmers knowing C.
Q.20) How many programmers know either C or Pascal?
Q.21) How many programmers don't know C and C++?
Q.22) How old is the oldest male programmer?
Q.23) What is the average age of female programmers?
Q.24) Calculate the experience in years for each programmer and display along with their names in descending order.
Q.25) Who are the programmers who celebrate their birthdays during the current month?
Q.26) How many female programmers are there?
Q.27) What are the languages known by the male programmers?
Q.28) What is the average salary?
Q.29) How many people draw 5000 to 7500?
Q.30) Display the details of those who don't know C, C++ or Pascal.
Q.31) Display the costliest package developed by each programmer.
Q.32) Produce the following output for all the male programmers Programmer Mr. Arvind – has 15 years of experience

Attempt These Combinations on SCOTT User

Q.1) Display MGR and ENAME from EMP table.
Q.2) Display the records where the job is either SALESMAN or DEPT NO =20.
Q.3) Display the record where MGR is blank.
Q.4) Display all the records where names begin with 'A'.
Q.5) Display the records whose ENAME begins with the letter in between 'K' to 'M'.
Q.6) Display all the records whose ENAME contain 5 characters.
Q.7) Display total number of employees department wise.
Q.8) Display the department that contains maximum employees.
Q.9) Display the employee who getting maximum salary.
Q.10) Display the last 4 character of each employee
Q.11) Display the year and the total number of employee hired during a year.
Q.12) Display the year and the total number of employee for the year in which maximum numbers of employee were hired.
Q.13) Display the record for all employees who have the same job as EMP NO=7838.
Q.14) Display all the records where job contain the letter "E"
Q.15) Display the record where the salary of the employee is greater than the salary obtained by 'TURNER"
Q.16) Select all employee who are either 'clerk' or 'salesman' and have salary greater than 1500.
Q.17) Select all the employee whose names start with 'Th' or 'Lh'.
Q.18) Display names ,annual salary ,comm. of those salesperson whose monthly salary >comm..
Q.19) Select employees who are with the company for more than 25 years
Q.20) Display details of those employees who have join the company on same date.
Q.21) Display details of highest paid employee in the EMP table.
Q.22) Display the 2nd highest paid employee in the EMP table
Q.23) Display DEPT NO which do not have clerk.
Q.24) Update sal of all employees by 2000 who are working in the company for more than 15 years and drawing sal<4000..
Q.25) Display ENAME , HIREDATE in ascending order.
Q.26) Suppose a table contain thousand data and you are needed to select the 2nd last record from the table .write the query.
Q.27) Print the list of employees displaying last salary if exactly 1500 .display on target if less than 1500.
Q.28) Display DEPT NO and the name of the employee who gets minimum salary in each department.
Q.29) Display ENAME ,annual salary and comm. Of every employee in DEPT NO 30 and fill the comm. Field with 0 if it doesn't contain any value.

Q.30) Display details of that employee who have no manager.
Q.31) Display details of those employees who are clerk and whose salary lies between 1000 and 2000
Q.32) List the employee by name , salary, and department name for every employee in the company except clerks , sort on salary , displaying the highest salary first.
Q.33) Display Name and Total Remuneration for all employees.
Q.34) List the employee name and salary increased by 20%.
Q.35) Display each employee name with hire date and salary review date .assuming review date is first year after hire date
Q.36) Find out how many managers are there without listing them.
Q.37) Find the average salary and avg Total Remuneration for each job types. Remember Salesman earn comm..
Q.38) Find the job that was filled in the first half of 1983 and the same job that was filled during the same period of 1984.
Q.39) List all employees by NAME and NUMBER along with their manager's details.
Q.40) Write the query to display details for any employee who earns a salary greater that the average for their department .Sort in Department Number order.
Q.41) Find out the employee name , salary , dept no, who earns greater than every employee in Department no 30.
Q.42) Find the Department having maximum employees.
Q.43) Find the employee who earns more than 'MILLER"
Q.44) Find all the person who are not MANAGERS
Q.45) Find the name of person getting same salary in different department .
Q.46) Find the job whose average salary is equal to maximum average salary of any job.
Q.47) Find out the details of all the person who have been assigned same job in different DEPT NO.
Q.48) Find the department which has more than 3 employees.
Q.49) Write a query which will return the day of the week ,for any date entered in format :DD:MM:YY.
Q.50) Check whether all employee number are indeed unique.
Q.51) List lowest paid employees working for each manager. Exclude any group where the min salary is <1000.Sort the output by salary.
Q.52) Show the records of employees working in DALLAS location.
Q.53) List the following details for employees who earn $36,000 a year or who are clerk.
Q.54) List all the employees by name, and number along with their manager's name and number.
Q.55) Find the employees who joined the company before their managers.
Q.56) Find the job with the highest average salary.
Q.57) Find the employees who have at least one person reporting to them.
Q.58) Find all employees whose department is not in DEPT table.
Q.59) Find Department which has no employees.
Q.60) Display the following information for the department with the highest annual remuneration bill.
Q.61) In which year did most people join the company? Display the year and number of employees.