Oracle – SQL

Chapter 1: Introduction

What is a Database?

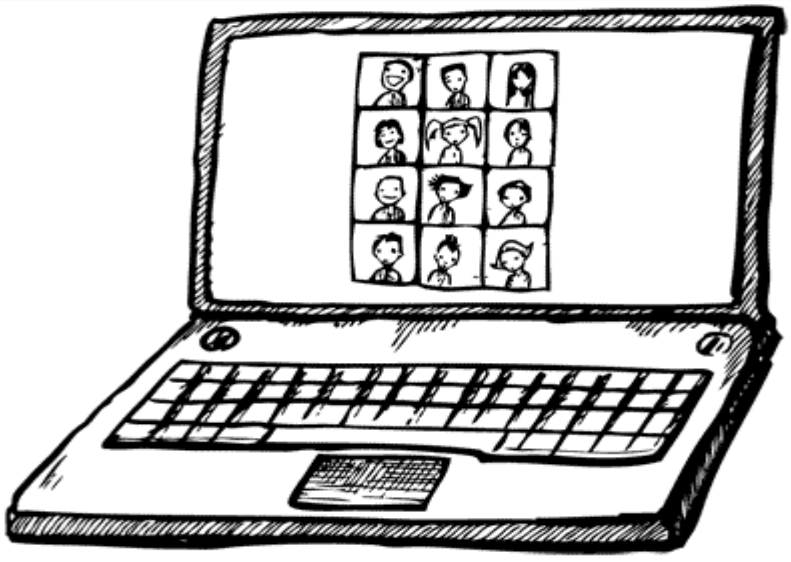
(Book Definition) – A **Data Base** is an Organized Collection of *Data* which can be easily accessed, managed and updated.

In today's World, Data plays a vital role in every business. In our day to day life, we see or interact with many applications and Software's, every application that we see or work with will have two parts:

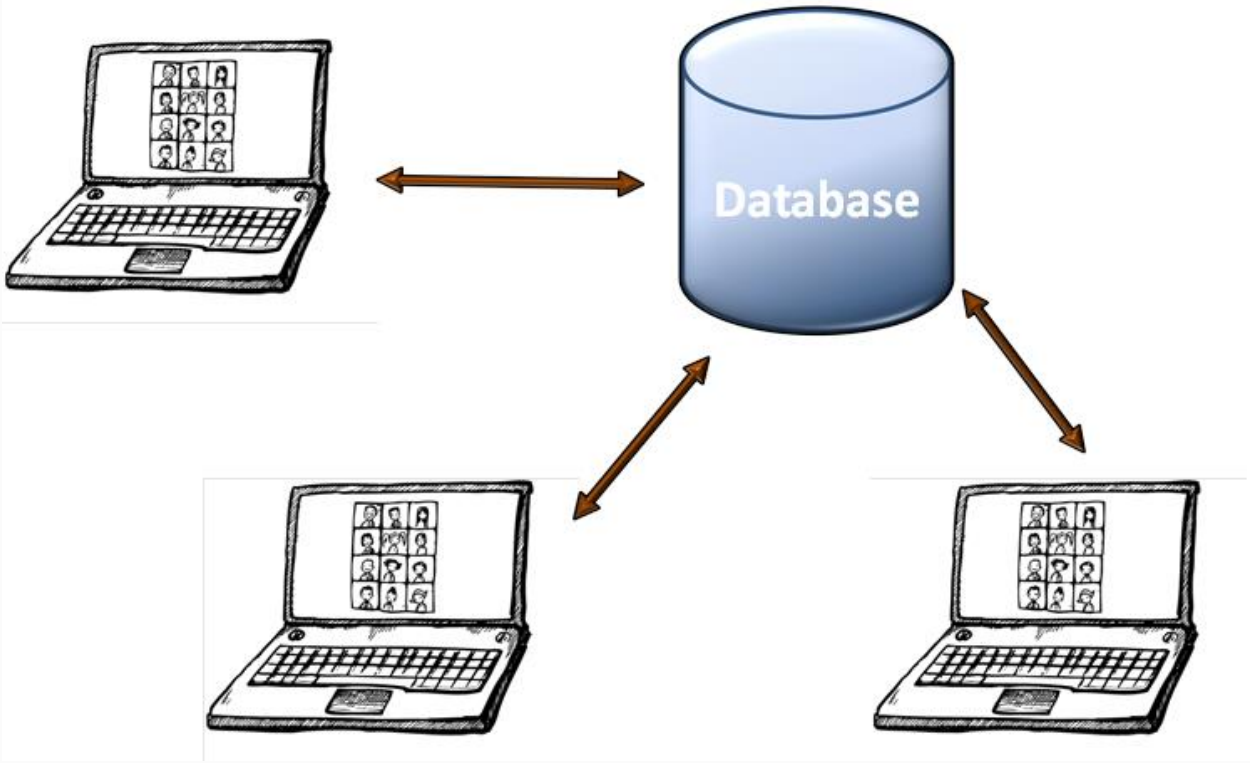
1. GUI (Graphical User Interface / Front end)

2. Database

To keep it simple GUI is the part where user interacts with (like Facebook applications – look and feel) and the Data that we see in the application (like Facebook profile, messages, images and videos) are pulled from Database.



End User who interacts with the application may not know how the data is being fetched and where so much of information is stored. Internally all the dynamic content that we see in the application is fetched from Database.



Database and all its components should be designed and configured at the time of application development. Once the application is developed we will not be able to make changes to the database structure as every change will hugely affect the business application GUI code.

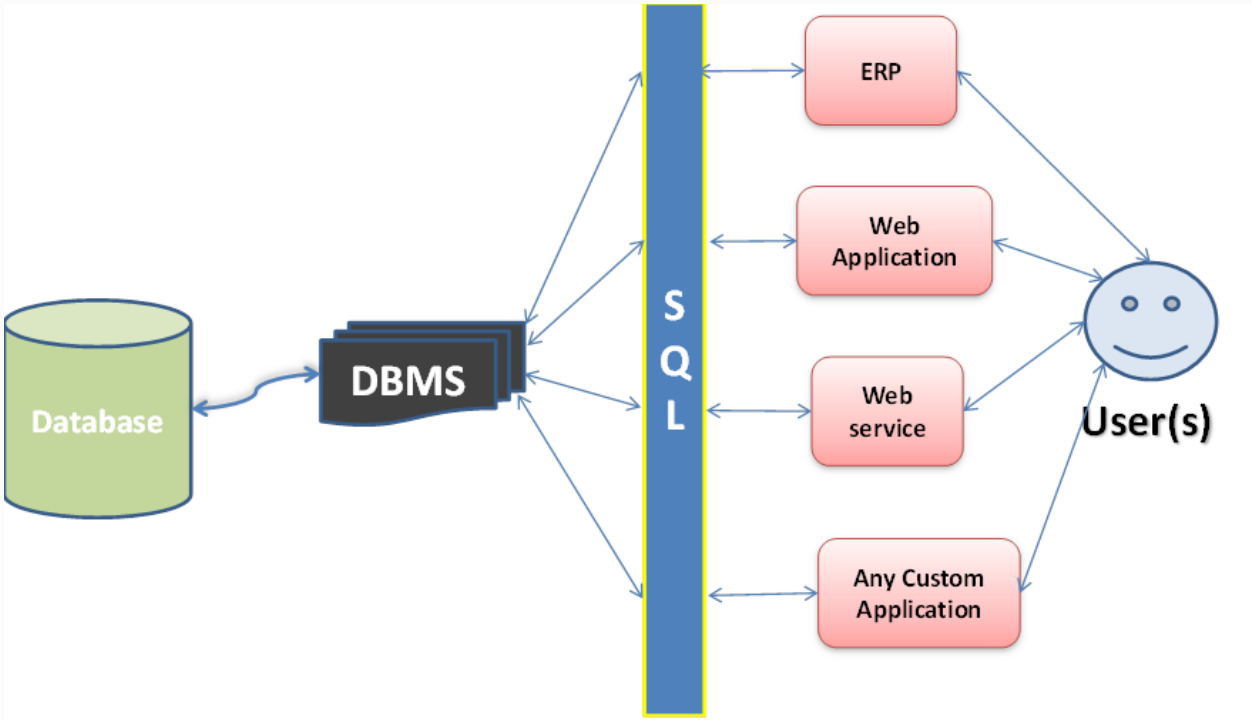
It is very important to make sure that data is securely maintained and accurately stored. So to maintain security and accuracy in database a set of rules / software system is defined and that we call it as DBMS (Data Base Management System – which performs all the operations on the database)

What is DBMS?

DBMS (Database Management System) is a software tool that is used to store and manage data in the Database.

A database management system contains a set of programs that control the creation, maintenance, and use of a database. Like:

- Adding new data to the table.
- Modifying existing data.
- Deleting unwanted data.



DBMS allows different user application programs to concurrently access the same database.

Before Database and DBMS were introduced, traditional approach was to store data in flat files (physical files in system) but there were some disadvantages with it.

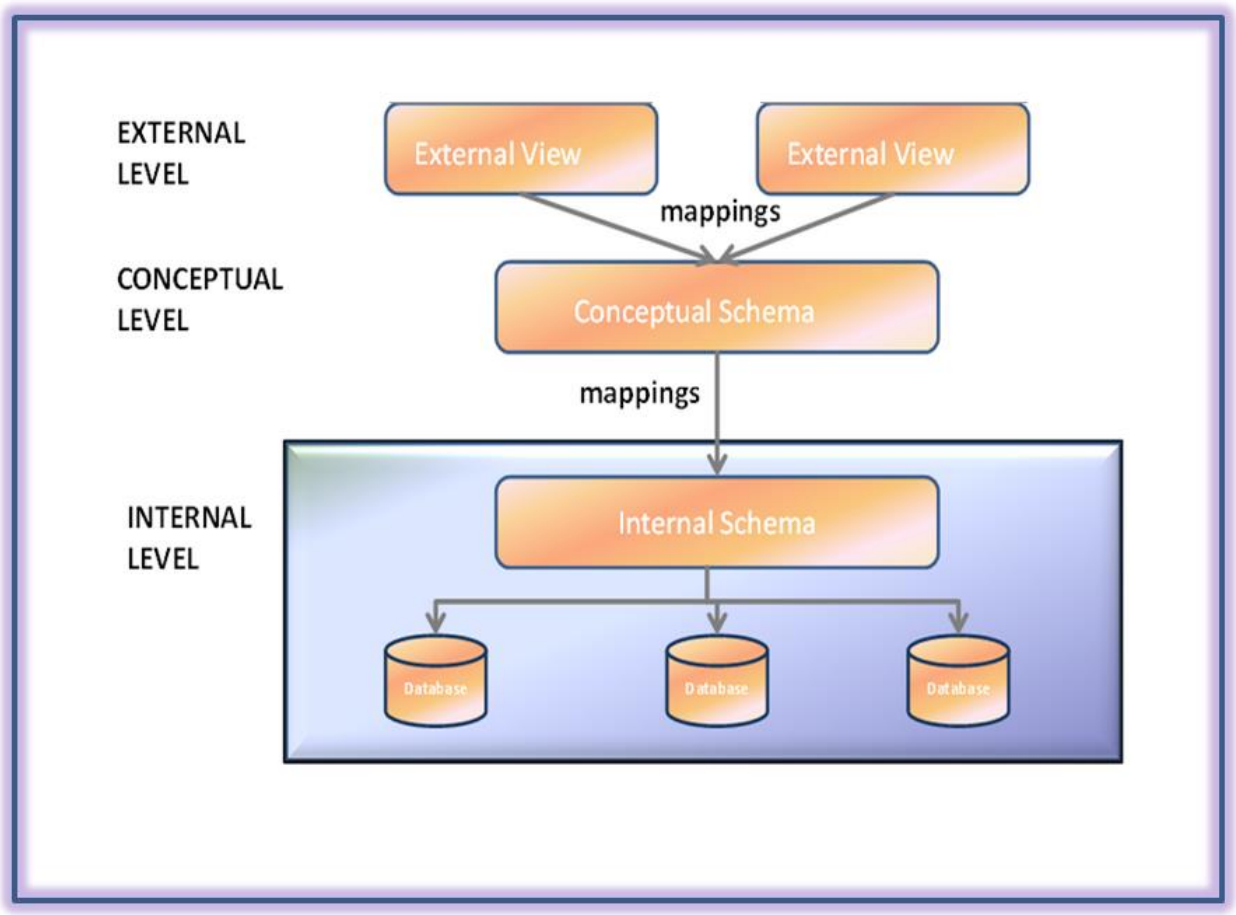
Following are some of the advantages of using a database over a traditional file-processing system:

- Potential for enforcing standards.
- Flexibility.
- Fast response to information requests.
- Availability of up-to-date information to all users.
- Less storage.

Architecture of the DBMS

The architecture of the DBMS can be viewed as a 3-level system comprising the following:

- The external level (this is where application developer or end user interacts with database)
- The conceptual level (At this level all the DBMS functions resides, that is functions for adding , modifying and deleting data)
- The internal or the physical level is the level where the data resides. (the physical storage happens)



Facts about DBMS:

- DBMS came into picture when database was introduced (i.e. just a simple software to interact with database)
- According to DBMS data can be stored in any format. (not necessary that data should be stored in form of tables)
- DBMS was a simple system to access database. There were issues for huge data.
- In DBMS, there were no proper set of rules defined on how data should be stored, how to avoid redundancy (duplicate records) and how to maintain data relationships.

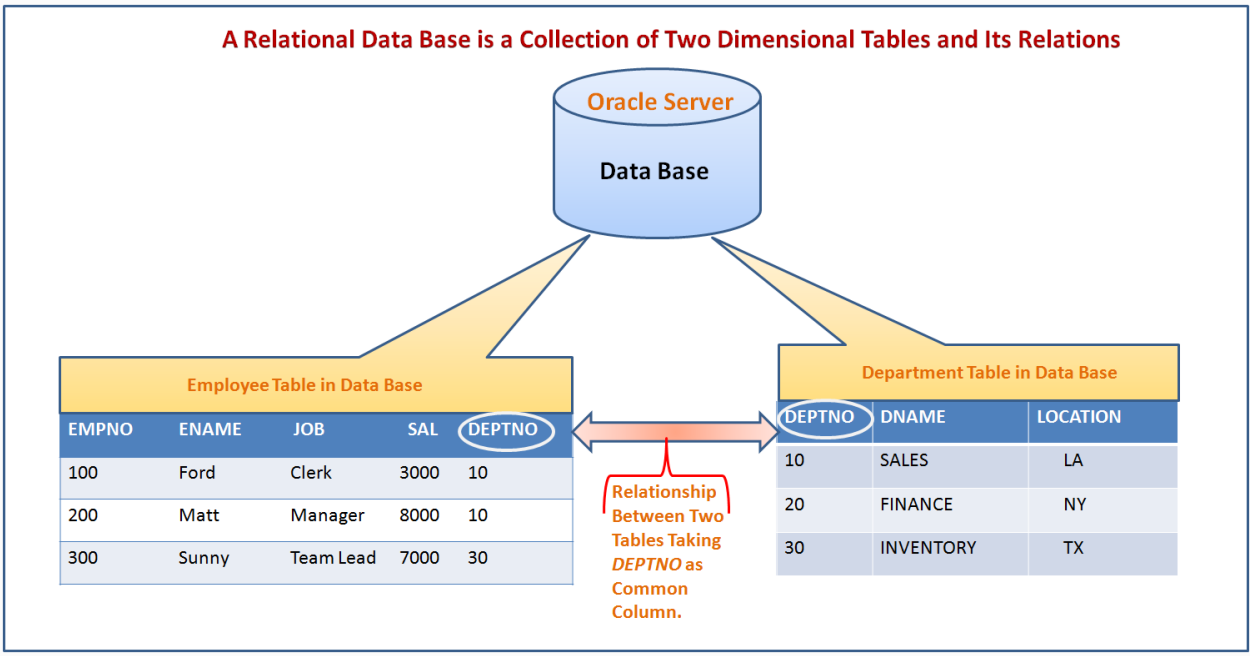
As the time progressed data complexity increased as usual, time and technology brains have introduced **RDBMS** and came up with some set of rules.

What is RDBMS?

A relational database management system (RDBMS) is a Database Management System (DBMS) that is based on the relational model introduced by E. F. Codd and most popular databases currently in use are based on the relational database model. To put in different words RDBMS is built on top of in which data is stored in tables and the relationships among the data are maintained. The data stored in a table is organized into rows and columns. Each row in a table represents an individual record and each column represents a field. A record is an individual entry in the database.

Difference between DBMS and RDBMS

For example, consider the database which stores employee information. In DBMS, all details like empno, ename, job, salary, deptno, dname, location, street, city, state, phone will be stored in a single table. But in RDBMS, the data model will be designed in such a way that like the empno, ename, job, salary and deptno will be stored in emp table and deptno, dname, location will be stored in dept table and location, street, city, state, phone will be stored under locations table. Any information to be obtained is done by properly relating the 'emp', 'dept' and 'locations' tables.



E. F. Codd Rules : Codd's twelve rules are a set of thirteen rules (numbered zero to twelve) proposed by **Edgar F. Codd,** a pioneer of the relational model for databases, designed to define what is required from a database management system in order for it to be considered relational, i.e., a relational database management system (RDBMS). As a beginner it

might be little difficult to understand all the rules defined below but however once we start using the database and database programming languages it will be very easy to correlate the rules and understand better.

Rules:

Rule (0): ***The system must qualify as relational, as a database, and as a management system***.
For a system to qualify as a relational database management system (RDBMS), that system must use its relational facilities (exclusively) to manage the database.

Rule 1: ***The information rule***:
All information in a relational database (including table and column names) is represented in only one way, namely as a value in a table.

Rule 2: ***The guaranteed access rule***:
All data must be accessible. This rule is essentially a restatement of the fundamental requirement for primary keys. It says that every individual scalar value in the database must be logically addressable by specifying the name of the containing table, the name of the containing column and the primary key value of the containing row.

Rule 3: ***Systematic treatment of null values***:
The DBMS must allow each field to remain null (or empty). Specifically, it must support a representation of "missing information and inapplicable information" that is systematic, distinct from all regular values (for example, "distinct from zero or any other number", in the case of numeric values), and independent of data type. It is also implied that such representations must be manipulated by the DBMS in a systematic way.

Rule 4: ***Active online catalog based on the relational model***:
The system must support an online, inline, relational catalog that is accessible to authorized users by means of their regular query language. That is, users must be able to access the database's structure (catalog) using the same query language that they use to access the database's data.

Rule 5: ***The comprehensive data sub language rule***:
The system must support at least one relational language that

1. Has a linear syntax
2. Can be used both interactively and within application programs.

3. Supports data definition operations (including view definitions), data manipulation operations (update as well as retrieval), security and integrity constraints, and transaction management operations (begin, commit, and rollback).

Rule 6: *The view updating rule*:
All views that are theoretically updatable must be updatable by the system.

Rule 7: *High-level insert, update, and delete*:
The system must support set-at-a-time insert, update, and delete operators. This means that data can be retrieved from a relational database in sets constructed of data from multiple rows and/or multiple tables. This rule states that insert, update, and delete operations should be supported for any retrievable set rather than just for a single row in a single table.

Rule 8: *Physical data independence*:
Changes to the physical level (how the data is stored, whether in arrays or linked lists etc.) must not require a change to an application based on the structure.

Rule 9: *Logical data independence*:
Changes to the logical level (tables, columns, rows, and so on) must not require a change to an application based on the structure. Logical data independence is more difficult to achieve than physical data independence.

Rule 10: *Integrity independence*:
Integrity constraints must be specified separately from application programs and stored in the catalog. It must be possible to change such constraints as and when appropriate without unnecessarily affecting existing applications.

Rule 11: *Distribution independence*:
The distribution of portions of the database to various locations should be invisible to users of the database. Existing applications should continue to operate successfully:

1. when a distributed version of the DBMS is first introduced; and
2. When existing distributed data are redistributed around the system.

Rule 12: The *nonsubversion rule*:
If the system provides a low-level (record-at-a-time) interface, then that interface cannot be used to subvert the system, for example, bypassing a relational security or integrity constraint.

As a result of this **CODD** rules a RDBMS (relational database management system) ensures and enforce the database to follow ACID properties.

Now let's see what ACID rules are?

What are ACID Rules?

ACID (atomicity, consistency, isolation, durability) is a set of properties that guarantee that database transactions are processed reliably. In the context of databases, a single logical operation on the data is called a transaction. For example, a transfer of funds from one bank account to another, even though that might involve multiple changes (such as debiting one account and crediting another), is a single transaction.

Atomicity: -

Atomicity requires that each transaction is "all or nothing": if one part of the transaction fails, the entire transaction should fails, and the database state is left unchanged. An atomic system must guarantee atomicity in each and every situation, including power failures, errors, and crashes.

Consistency states that only valid data will be written to the database. If, for some reason, a transaction is executed that violates the database's consistency rules, the entire transaction will be rolled back and the database will be restored to a state consistent with those rules. On the other hand, if a transaction successfully executes, it will take the database from one state that is consistent with the rules to another state that is also consistent with the rules.

Isolation: -

The isolation property ensures that the concurrent execution of transactions results in a system state that could have been obtained if transactions are executed serially i.e. one after the other.[citation needed] The isolation property ensures that the concurrent execution of transactions should not impact each other's execution.

For **example**, if Mark issues a transaction against a database at the same time that Scott issues a different transaction; both transactions should operate on the database in an isolated manner. The database should either perform Mark's entire transaction before executing Scott's or vice-versa. This prevents Mark's transaction from reading intermediate data produced as a side effect of part of Scott's transaction that will not eventually be committed to the database. Note that the isolation property does not ensure which transaction will execute first, merely that they will not interfere with each other.

Durability: -

Durability means that once a transaction has been committed, it will remain so, even in the event of power loss, crashes, or errors. In a relational database, for instance, once a group of SQL statements execute, the results need to be stored permanently (even if the database crashes immediately thereafter).

What is SQL?

SQL is a standard language designed for accessing and managing data in Relational Database Management Systems (RDBMS).
SQL is based on ANSI (American National Standards Institute) standard introduced in 1986 and there are many SQL database software's available in the market which follows the ANSI standard like:

- Oracle
- Mysql
- Sqlserver
- DB2 ……etc

Note: Most of the SQL database programs currently available in the market also have their own proprietary extensions in addition to the SQL standard.

What Can SQL do?

- SQL can execute queries against a database
- SQL can retrieve data from a database
- SQL can insert records in a database
- SQL can update records in a database
- SQL can delete records from a database
- SQL can create new databases
- SQL can create new tables in a database
- SQL can create stored procedures in a database
- SQL can create views in a database
- SQL can set permissions on tables, procedures, and views

Using SQL we execute different commands on RDBMS software's to perform different operations on the data available in the database.
Whenever we execute a SQL command the RDBMS software translates our command into understandable format of the database and do the necessary operation on data.

To make it simple SQL Commands are divided into five categories, depending upon what they do.

1. Data Retrieval Language (DRL)
2. Data Manipulation Language (DML)
3. Data Definition language (DDL)
4. Transaction Control Language (TCL)
5. Data Control Language (DCL)

SQL Sub Languages	SQL Statements
Data Retrieval Language (**DRL**)	*SELECT*
Data Manipulation Language (**DML**)	*INSERT, UPDATE, MERGE, DELETE*
Data Definition language (**DDL**)	*CREATE, ALTER, DROP, TRUNCATE, RENAME*
Transaction Control Language (**TCL**)	*COMMIT, ROLLBACK, SAVEPOINT*
Data Control Language (**DCL**)	*GRANT, REVOKE*

SQL has a special version named PL/SQL, which is sometimes termed as a superset of SQL. It bridges the gap between database technology and procedural languages as programmers can use PL/SQL to create programs for validation and manipulation of table, something that was not possible with SQL.

Newton Apples has separate training course for PL/SQL, Please go through PL/SQL content for more details.

RDBMS Products in the Market:

Starting from 1980's there are many **RDBMS products** introduced in the market. However I would like to introduce top 10 products available in today's market. All these products are highly competitive, and come packed with advanced features. These database systems range in price from free to thousands of dollars.

1. Oracle

Oracle began its journey in 1979 and it is the first commercially available relational database management system (RDBMS). There is no doubt that Oracle ranks top among all the enterprise database systems as it delivers industry leading performance, scalability, security and reliability and handles complex database solutions with ease.

Most of the ERP systems that are available in the market use Oracle as the database because of it performance and ability to handle huge data.

Product Owner	First Public Release Date	Latest Stable Version	Latest Release Date
Oracle Corporation	Nov-1979	11g Release 2	Aug-2009

2. SQL Server

SQL server is a Microsoft. Microsoft SQL Server is widely used for small and mid level industries. SQL server is mostly used in developing countries because of it easy to use and easy to maintain features. SQL Server has tight Windows operating system integration and makes it an easy choice for firms that choose Microsoft products for their enterprises.

Product Owner	First Public Release Date	Latest Stable Version	Latest Release Date
Microsoft	1989	v11	2012

3. DB2

DB2 is an IBM product and it is mostly used in IBM products like IBM Tivoli TSM etc. DB2 runs on Linux, UNIX, Windows and mainframes. IBM pits its DB2 9.7 system squarely in competition with Oracle's 11g, via the International Technology Group, and shows significant cost savings for those that migrate to DB2 from Oracle.

Product Owner	First Public Release Date	Latest Stable Version	Latest Release Date
IBM	1983	10.1	Apr-2012

4. Sybase

Sybase, an SAP company, is an enterprise software and services company offering software to manage, analyze, and mobilize information using relational databases. Although its market share slowed for few years, it's returning with powerful positioning in the next-generation transaction processing space. Sybase has also thrown a considerable amount of weight behind the mobile enterprise by delivering partnered solutions to the mobile device market. Sybase now works with other industry leaders in infrastructure, data

storage and virtualization to optimize technologies for delivery into public and virtual private cloud environments that provide greater technology availability and flexibility to Sybase customers looking to unwire their enterprise.

Product Owner	First Public Release Date	Latest Stable Version	Latest Release Date
SAP	1987	Sybase 365	Feb-2012

5. My SQL

MYSQL is most preferred choice for Web based and low cost products, this is an open source product. Using a free / open source minimize a product cost by significant amount. My SQL sold to Sun Microsystems in 2008 and now My SQL is currently part of the Oracle empire (January 2010) as Sun Microsystems is acquired by Oracle. More than just a niche database now, My SQL powers commercial websites by the hundreds of thousands and a huge number of internal enterprise applications. Although My SQL's community and commercial adopters had reservations about Oracle's ownership of this popular open source product, Oracle has publicly declared its commitment to ongoing development and support.

Product Owner	First Public Release Date	Latest Stable Version	Latest Release Date
Sun Microsystems(now Oracle Corporation)	Nov-1995	5.5.17	Oct-2011

6. PostgreSQL

PostgreSQL is an open source database and widely used in online gaming applications, data center automation suites and domain registries. The vast majority of Linux distributions have PostgreSQL available in supplied packages. Mac OS X, starting with Lion, has PostgreSQL server as its standard default database in the server edition and PostgreSQL client tools in the desktop edition.

Product Owner	First Public Release Date	Latest Stable Version	Latest Release Date
PostgreSQL Global Development Group	Jun-1989	9.1.3	Jun-2012

7. Teradata

Teradata is mainly used for for data warehouses and analytic applications because its products are meant to consolidate data from different sources and make the data easily available for analysis and reporting. Teradata was incorporated in 1979 and was formerly a division of NCR Corporation, with the spinoff from NCR on October 1, 2007.

Product Owner	First Public Release Date	Latest Stable Version	Latest Release Date
Teradata (now NCR corp.)	1984	13.10	

8. Informix

Informix is another IBM product. The Informix products were originally developed by Informix Corporation, whose Informix Software subsidiary was acquired by IBM in 2001. IBM has continued active development and marketing of the key Informix products and offers several Informix. Informix database is often associated with universities and colleges. Informix customers often speak of its low cost, low maintenance and high reliability.

Product Owner	First Public Release Date	Latest Stable Version	Latest Release Date
IBM	1980	11.70.xC5	May-2012

9. Ingres

Ingres Database is fully open source with a growing global community of contributors. Ingres is all about choice and choosing might mean lowering your total cost of ownership for an enterprise database system. Ingres was first created as a research project at the University of California, Berkeley, starting in the early 1970s and ending in the early 1980s. Ingres prides itself on its ability to ease your transition from costlier database systems. Ingres also incorporates security features required for HIPPA and Sarbanes Oxley compliance.

Product Owner	First Public Release Date	Latest Stable Version	Latest Release Date
Ingres Corp.	1974	Ingres Database 10	Oct-2010

10. MaxDB

MaxDB is an ANSISQL-92 (entry level) compliant relational database management system (RDBMS) from SAP AG. MaxDB is targeted for large SAP environments e.g. mySAP Business Suite and other applications that require enterprise-level database functionality. It is able to run terabyte-range data in continuous operation.

Product Owner	First Public Release Date	Latest Stable Version	Latest Release Date
SAP AG	May-2003	7.6	Jan-2008

Chapter 2: Basics of Oracle SQL

Users and Schemas:

In Oracle database terminology, a user is someone who can connect to a database and create different database objects. However the user should have necessary privileges to connect and create.

Many of us get confused between the terms 'user' and 'schema'. Let's try to understand it better, In Oracle terminology both User and Schema are one and the same. All the objects that user owns are collectively called as schema. A schema, on its part, is always bound to exactly one user. Because there is obviously a 1 to 1 relationship between a user and a schema, these two terms are often used inter changeable.

We can use **dba_users** table to find out all the users created on the database.

Create User

Use the **CREATE USER** statement to create and configure a database user, which is an account through which we can log in to the database and perform database operations

```
CREATE USER user1 IDENTIFIED BY user1;
```

In the above syntax **'user1'** is the user name and **'user1'** is the password.

Within each database, a user name must be unique with respect to other user names and roles. Furthermore, each user has an associated schema. Within a schema, each schema object must have a unique name.

A newly created user cannot connect to the database until you grant the user the **CREATE SESSION** system privileges. So, immediately after you create the user account, use the **GRANT** SQL statement to grant the user these privileges.

When a user is created a default table space will be assigned. When a schema object is created in the user's schema, Oracle Database stores the object in the default user's table space.

```
GRANT ALL privileges TO user1;
```

Here we have given all the privileges to user 'user1'.

Introduction to Data Modeling:

In our information society, data storage has become an important aspect of every business and much of the world's computing power is now dedicated to maintaining and using databases. All kinds of data like contact information, financial data and records of sales should be stored in some form of a database.

While considering development of a business application, it is imperative that we have a good database design. A data model helps us achieve this by factoring various parameters to develop a functional, reliable and efficient database system.

What is a Data Model?

A data model is a conceptual representation of data structures that are required by a database. These data structures include data objects, associations between data objects, and the rules which govern operations on these objects. As the name implies, data model focuses on what data is required and how it should be organized rather than what operations will be performed on the data. A data model is independent of hardware or software constraints.

During software development, the data model gets its inputs from the planning and analysis stage. Here the modeler, along with Analysts, collects information about the requirements of the database by reviewing existing documentation and interviewing end-users.

Why is Data Modeling Important?

Data modeling is probably the most labor intensive and time consuming part of the development process. A common response by experts says that you should no more build a database without a model it is like building a house without blueprints.

The goal of the data model is to make sure that the all data objects required by the database are completely and accurately represented. Because the data model uses easily understood notations and natural language, it can be reviewed and verified as correct by the end-users.

The data model is also detailed enough to be used by the database developers to use as a "blueprint" for building the physical database. The information contained in the data model will be used to define the relational tables, primary and foreign keys, stored procedures, and triggers.

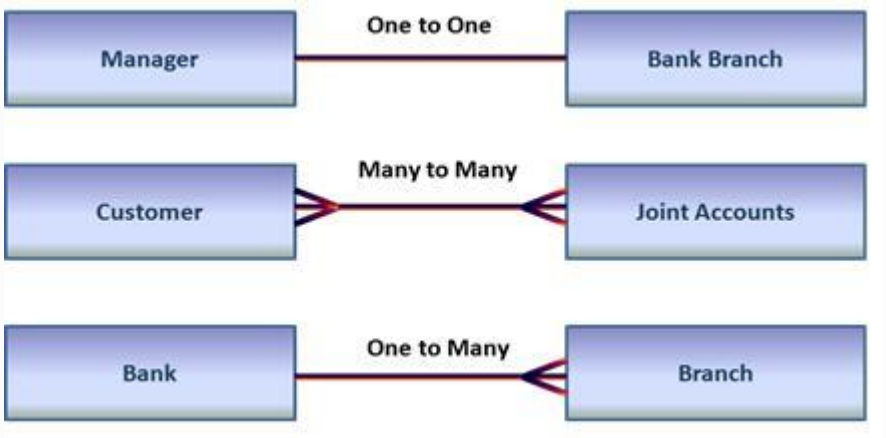
A poorly designed database will require more time in the long-term. Without careful planning we may create a database that omits data required to create critical reports and produces results that are incorrect or inconsistent.

There are two major methodologies used to create a data model: the Entity-Relationship (ER) approach and the Object Model. In this lesson we will discuss Entity-Relationship approach.

The Entity-Relationship Model

The Entity-Relation Model (ER) is the most common method used to build data models for relational databases. A basic component of the model is the Entity-Relationship diagram which is used to visually represent data objects.

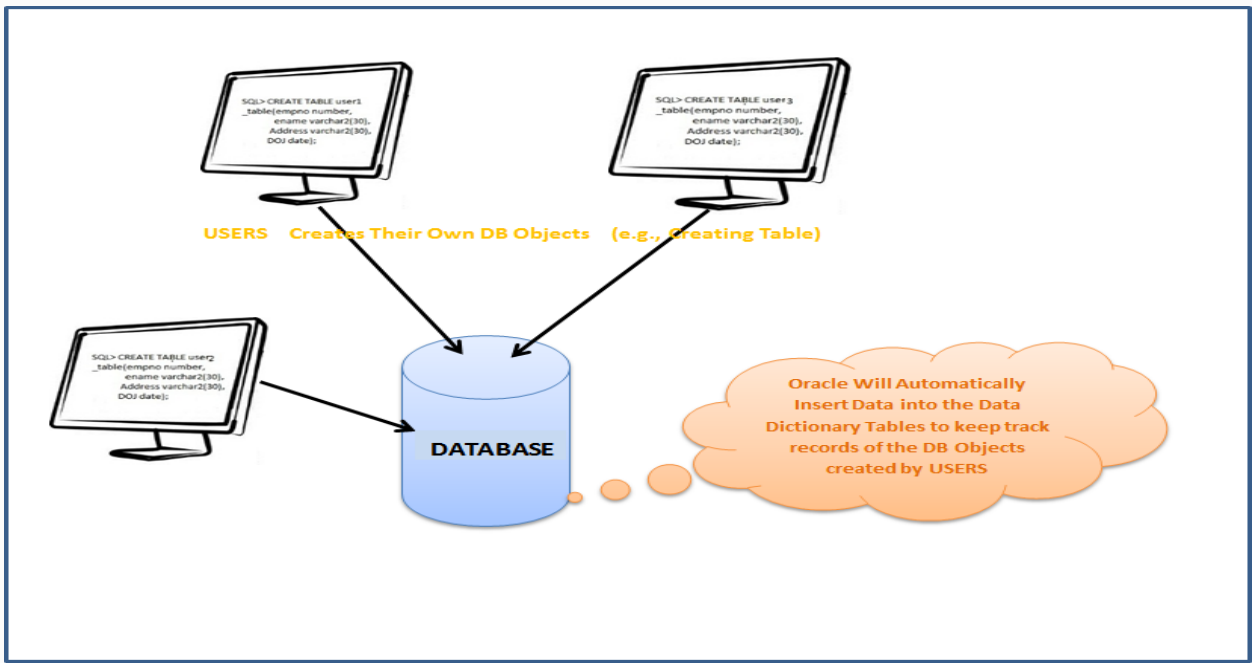
From the information collected during requirement analysis, data objects are identified and classified as entities, attributes, or relationship; assigned names; and defined using terms familiar to the end-users. The objects are then modeled and analyzed using an ER diagram. The diagram can be reviewed by the modeler and the end-users to determine its completeness and accuracy. If the model is not correct, it is modified, which sometimes requires additional information to be collected. The review and edit cycle continues until the model is certified as correct.



This lesson is just to give a heads up on the basics and need of a Data Model. However a separate course will be posted to discuss in detail about all topics of Data Model.

Introduction to Data Dictionary:

- **Data Dictionary** is the main source of information of every RDBMS.
- A **Data Dictionary** is a collection of pre-defined set of tables called as dictionary tables (or) System Tables (or) Pre-defined tables.
- These **Data Dictionary** tables are created by executing some readymade scripts provided by Oracle. All these tables remain same in all Oracle Software's.
- All these **Data Dictionary** tables belong to the user accounts "sys" and "system".
- The owner of "sys" and "system" accounts is DBA i.e., the owner of **Data Dictionary** table is **DBA.**
- The Super User who controls all the resources of the Data Base is called the **DBA** including starting and stopping the Data Base Server.
- **Data Dictionary** is the central source of **RDBMS.**
- All the **Data Dictionary** tables get the data (populated) from Oracle Itself.
- Oracle user "*Recursive SQL*" to enter the data into the Dictionary Tables.



Recursive SQL:

The SQL generated by Oracle on its own and executed by Oracle is called as Recursive SQL. The data available in the Data Dictionary is called as **"Meta Data".**

Meta Data:

The data about the user and the users created data which is generated by the Recursive SQL is called Meta Data. In simple words a data about the data is called Meta data.

Let's say in example a user created one table and inserted some data into that table, now what Oracle will do, here in this case oracle will keep track record of the table which is created by user in another table without user knowledge (here the another table is not created by any other user those are predefined tables created by Oracle not by user), and those track record maintaining tables are called as Data Dictionary Tables.

Note:

Not only table records all the Data Base objects which are created by Individual users are maintained by Data Dictionary tables of Oracle.

Note to Remember:

All the users of the Data Base are having a read permission i.e., a SELECT permission on most of the Data Dictionary tables. The tables created by the individual users for their own project purpose are called as User Tables and the data is Users Data.

Oracle SQL Data Types:

As part of SQL basics, in this lesson we will discuss about data types.

Data types are used to classify the type of data that is being stored or used. In SQL we use data types when creating the tables to classify what type of data should be stored in the columns.

For example we wanted to add 'hire date' column in EMP table, As per my business I am aware that this column will store DATE information only at any given point of time. So as confirmed we will assign this column with data type – **'DATE'.**

Once the column is added and start inserting data into the EMP table, 'hire date' column will only accept DATE information.

Following are the frequently used data types:

Char(size) –

This data type is used to store character information. Size specifies maximum size of character information that variable can hold. Columns defined with char data type will have fixed size allocation.

Example*:*-

Name **CHAR**(7)

If value 'Smith' is inserted into the column then (even it is a 5 character string) system will consider it as 7 character string with 2 spaces after the last character. i.e. 'Smith '
Using this char data type will have performance issues as it is fixed length and will assign unnecessary space if short. To overcome this Oracle has introduced another data type called 'varchar/varchar2'

Varchar(size) or Varchar2(size) -

Varchar and varchar2 datatypes both are used to store character information. However, 'Varchar' type should not be used as it is reserved for future usage.

Varchar2 stores exact length of the string that is inserted.

Example:-

Name varchar2(7)

If value 'Smith' is inserted into the column then it will store the exact string and will not concatenate any spaces even size is 7.

Number –

Number data type is used to store numeric data. All numeric data like +ve integers, -ve integers, decimals will come under number datatype. When defining a column we can optionally specify size and precision.

Example: -

Number(2) mean it can store a numeric values between 00 and 99
Number(8,2) means it can store values whose after decimal points values can go up to two places

Date -

Date data type is used to store date information. This Date datatype can understand and identify different date formats.
Till now we discussed most commonly used datatypes. Below is the table containing all the data types available in Oracle SQL.

Type	Description
CHAR[(length [BYTE | CHAR])]	Fixed-length character data of length bytes or characters and padded with trailing spaces. Maximum length is 2,000 bytes.
VARCHAR2(length [BYTE | CHAR])	Variable-length character data of up to length bytes or characters. Maximum length is 4,000 bytes.
NCHAR[(length)]	Fixed-length Unicode character data of length characters. Number of bytes stored is 2 * length for AL16UTF16 encoding and 3 * length for UTF8. Maximum length is 2,000 bytes.
NVARCHAR2(length)	Variable-length Unicode character data of length characters. Number of bytes stored is 2 * length for AL16UTF16 encoding and 3 * length for UTF8 encoding. Maximum length is 4,000 bytes.
NUMBER(precision, scale)	Variable-length number; precision is the maximum number of digits (in front of and behind a decimal point, if used) that may be used for the number. The maximum precision supported is 38; scale is the maximum number of digits to the right of a decimal point (if used). If neither precision nor scale is specified, then a number with up to a precision and scale of 38 digits may be supplied (meaning you can supply a number with up to 38 digits, and any of those 38 digits may be in front of or behind the decimal point).

DATE	Date and time with the century, all four digits of year, month, day, hour (in 24-hour format), minute, and second. May be used to store a date and time between January 1, 4712 B.C. and December 31, 4712 A.D. Default format is specified by the NLS_DATE_FORMAT parameter (for example: DD-MON-RR).
CLOB	Variable length single-byte character data of up to 128 terabytes.
NCLOB	Variable length Unicode national character set data of up to 128 terabytes.
BLOB	Variable length binary data of up to 128 terabytes.
BFILE	Pointer to an external file.
LONG	Variable length character data of up to 2 gigabytes. Super-ceded by CLOB and NCLOB types, but supported for backwards compatibility.
RAW(length)	Variable length binary data of up to length bytes. Maximum length is 2,000 bytes. Superseded by BLOB type, but supported for backwards compatibility.
LONG RAW	Variable length binary data of up to 2 gigabytes. Super-ceded by BLOB type but supported for backwards compatibility.
ROWID	Hexadecimal string used to represent a row address.
UROWID[(length)]	Hexadecimal string representing the logical address of a row of an index-organized table; length specifies the number of bytes. Maximum length is 4,000 bytes (also default).
VARRAY	Variable length array. This is a composite type and stores an ordered set of elements.

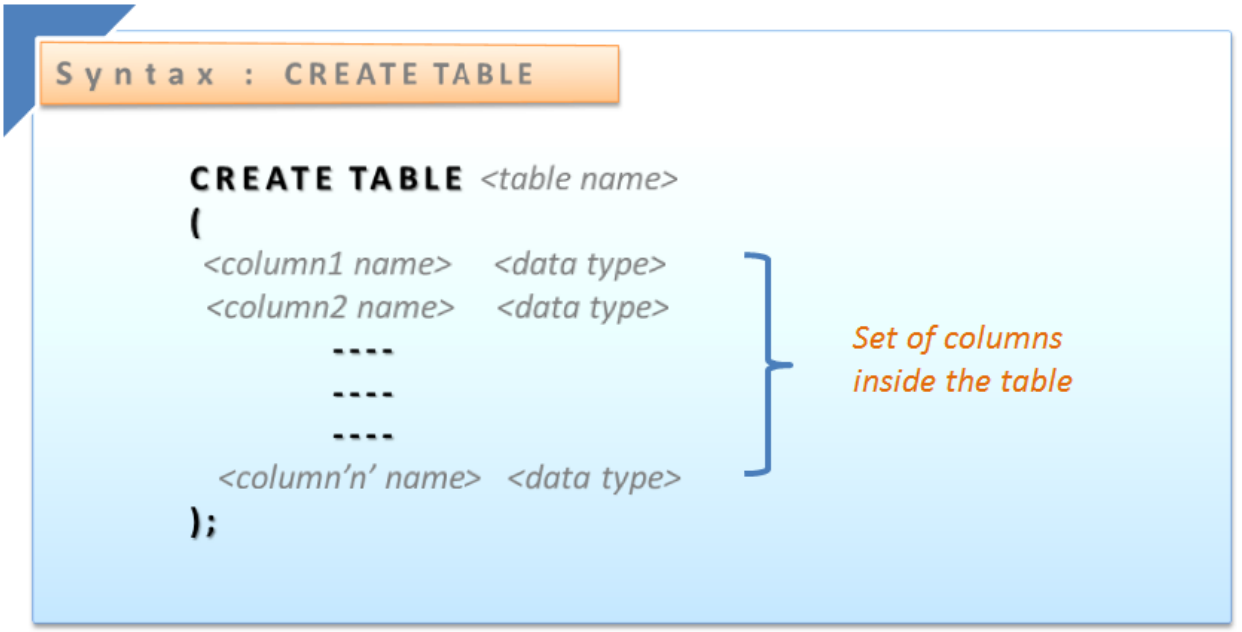
NESTED TABLE	Nested table. This is a composite type and stores an un ordered set of elements.
XMLType	Stores XML data.
User defined object type	You can define your own object type and create objects of that type.

Chapter 3: DDL

Create Table:

CREATE TABLE command is used to create a new table. 'CREATE' is part of Data Definition Language (DDL).
While creating a table we provide the basic information for each column together with their data type and sizes.

The Syntax for the CREATE TABLE Statement is:



- **table_name** - is the name of the table.
- **column_name1, column_name2….** – is the name of the columns
- **datatype** – is the type of data that column holds.

Example: If we want to create the employee table, the statement would be like

```
CREATE TABLE emp
(  empno    NUMBER(5),
   ename    CHAR(20),
   job      CHAR(10),
   sal      NUMBER(20),
   comm     NUMBER(10),
```

```
  deptno    NUMBER(10)
);
```

Output:

Once the table is created we will receive a message “Table Created”.
If a table of that name is already available then we will not be allowed to create that table\

Table Naming Conventions:

The name we choose for our table should follow these standard rules:

- The name must begin with a letter A-Z or a-z.
- Can contain numbers and underscores.
- Can be in UPPER or lower case.
- Can be up to 30 characters in length.
- Cannot use the same name of another existing object in our schema.

Examples of Object Names:

Name	Valid?
Emp or emp	Yes
1EMP or 1emp	No – should not begin with a number
Employee_details	Yes
Employee details	No – should not contain a blank space in between the name
UPDATE	No – ‘UPDATE’ is a SQL reserved word

Creating a Table from another Table/View:

Oracle provides another way for creating a table that is ‘copying another table’. This method is useful when we wanted to select the data out of a table for temporary modifications or if we

wanted to create a table that is similar to the existing table or view using similar data.
The syntax is as follows:

```
CREATE TABLE na_emp AS
SELECT * FROM emp;
```

Here, na_emp table is created with the same number of columns and with datatypes as employee table.

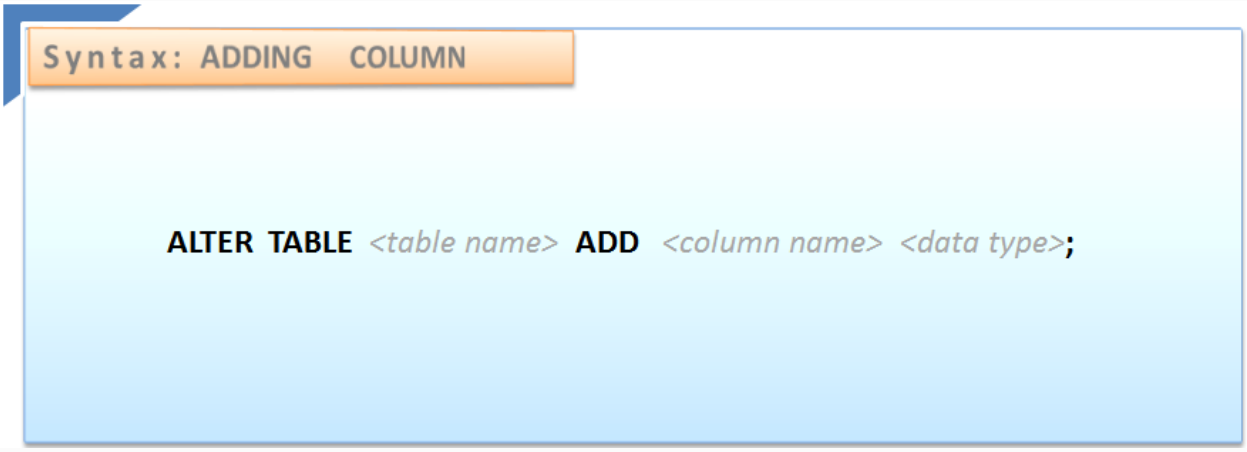
Alter Table:

Alter command is part of Data Definition Language (DDL). The SQL ALTER TABLE command is used to modify the definition (structure) of a table by modifying the definition of its columns. The ALTER command is used to perform the following functions.

1) Add, drop and modify table columns
2) Add and drop constraints
3) Enable and Disable constraints (will be discussed in constraints lesson)

Adding a column in table

Below is the syntax used to add a column in table.



Example:-

```
SQL>ALTER TABLE emp ADD na_column DATE;
```

Once the table is successfully altered then we will receive a message saying “Table altered”
Let’s check the “emp” table structure.

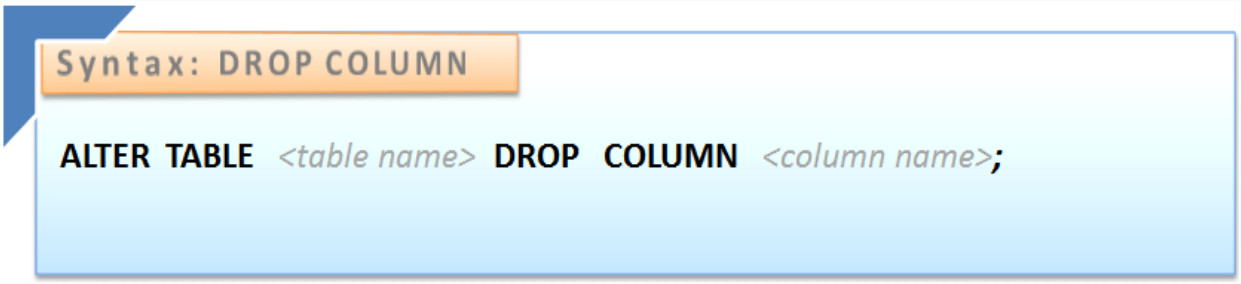
```
SQL>SELECT * FROM  emp;
```

-- A new column is added in emp table.

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO	NA_COLUMN
7369	SMITH	CLERK	7902	12/17/1980	800		20	
7499	ALLEN	SALESMAN	7698	2/20/1981	1600	300	30	
7521	WARD	SALESMAN	7698	2/22/1981	1250	500	30	
7566	JONES	MANAGER	7839	4/2/1981	2975		20	
7654	MARTIN	SALESMAN	7698	9/28/1981	1250	1400	30	
7698	BLAKE	MANAGER	7839	5/1/1981	2850		30	
7782	CLARK	MANAGER	7839	6/9/1981	2450		10	

Removing a column from table

Below is the syntax used to remove a column from a table



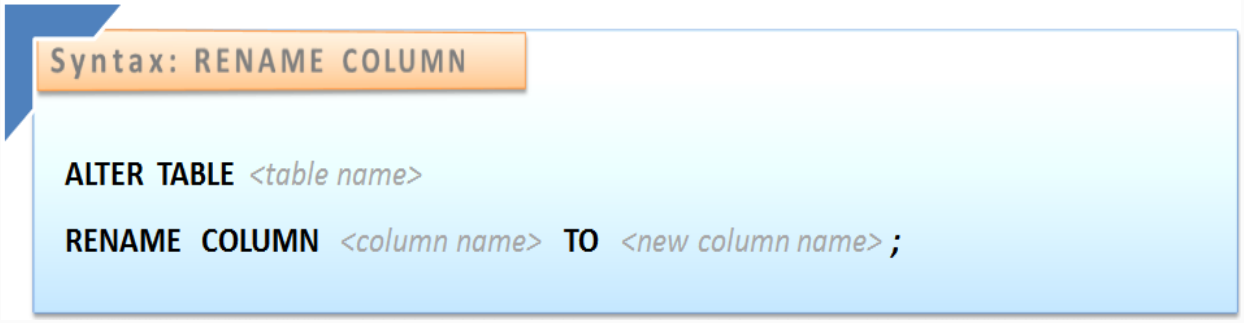
Let's check the "emp" table structure.

```
SELECT * FROM emp;
-- 'na_column' is removed from emp table.
```

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7369	SMITH	CLERK	7902	12/17/1980	800		20
7499	ALLEN	SALESMAN	7698	2/20/1981	1600	300	30
7521	WARD	SALESMAN	7698	2/22/1981	1250	500	30
7566	JONES	MANAGER	7839	4/2/1981	2975		20
7654	MARTIN	SALESMAN	7698	9/28/1981	1250	1400	30
7698	BLAKE	MANAGER	7839	5/1/1981	2850		30
7782	CLARK	MANAGER	7839	6/9/1981	2450		10

Renaming a Column in Table

Below is the syntax used to rename a column in table.



Example:-

```
ALTER TABLE emp RENAME COLUMN sal TO salary;
```

Let's check the "emp" table structure.

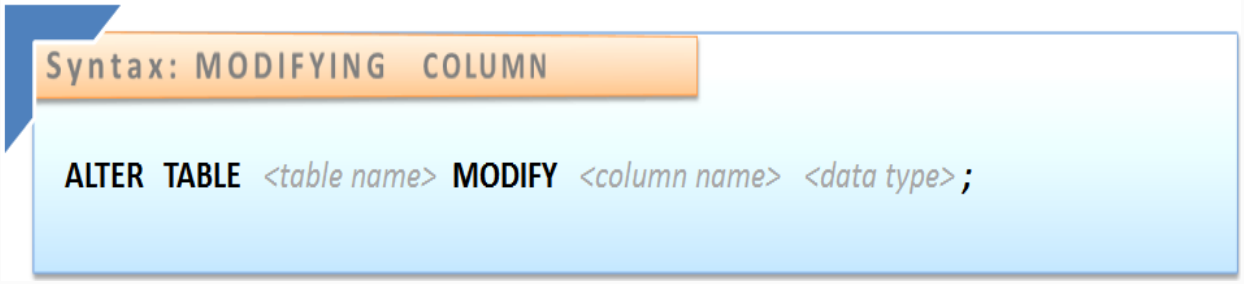
```
SELECT * FROM emp;

-- We can observe that "sal" column is renamed as "salary"
```

EMPNO	ENAME	JOB	MGR	HIREDATE	SALARY	COMM	DEPTNO
7369	SMITH	CLERK	7902	12/17/1980	800		20

Increasing or Decreasing Precision of a column

Below is the syntax used to increasing the precision of a column in table.



– Before we run the alter command lets first check the emp table description

Column Name	Col ID	Pk	Data Type	Null?	Default
EMPNO	1		NUMBER (4)	N	
ENAME	2		VARCHAR2 (10)	Y	
JOB	3		VARCHAR2 (9)	Y	
MGR	4		NUMBER (4)	Y	
HIREDATE	5		DATE	Y	
SALARY	6		NUMBER (7,2)	Y	
COMM	7		NUMBER (7,2)	Y	
DEPTNO	8		NUMBER (2)	Y	

– Here ENAME datatype size is 10

Example:- Run alter command to increase the precision.

SQL>**ALTER TABLE** emp **MODIFY** ename **VARCHAR**(25);

Let's check the "emp" table structure.

Column Name	Col ID	Pk	Data Type	Null?	Default
EMPNO	1		NUMBER (4)	N	
ENAME	2		VARCHAR2 (25)	Y	
JOB	3		VARCHAR2 (9)	Y	
MGR	4		NUMBER (4)	Y	
HIREDATE	5		DATE	Y	
SALARY	6		NUMBER (7,2)	Y	
COMM	7		NUMBER (7,2)	Y	
DEPTNO	8		NUMBER (2)	Y	

We can now observe that ENAME column data size is changed to 25.
Similarly we can decrease the precision of column but to decrease the precision, column should be empty.

Truncate Table:

Truncate command is part of **DDL** (*Data Definition Language*).Truncate command is used to delete the data permanently from the database table.

System does AUTO commit after the deletion and hence the data deleted cannot be rolled back.

Using Truncate we cannot delete partial records from the database. Once we run truncate command, system will delete all the records permanently from the database.

Example:-

```
SQL> TRUNCATE TABLE emp;
```

Once the table is successfully truncated then we will receive a message saying 'Table Truncated'
– Lets run the emp table after truncate

```
SELECT * FROM emp;
```

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO

– We can check the count as well

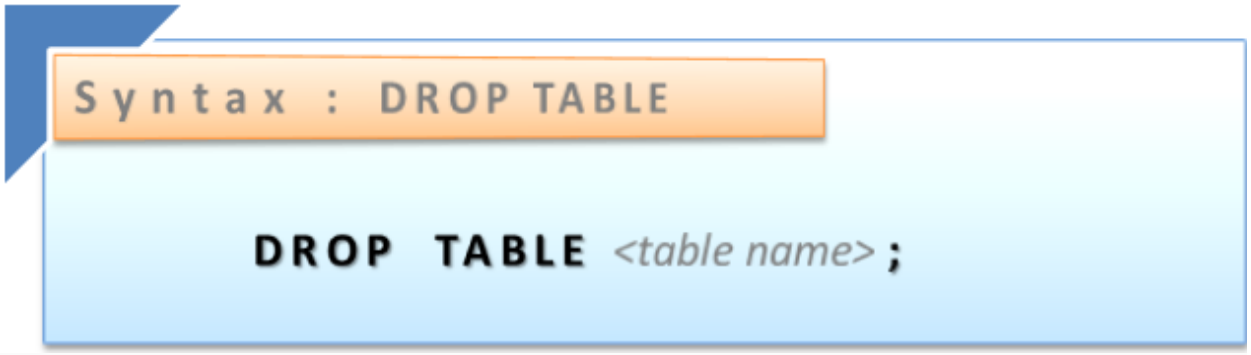
```
SELECT COUNT(*) FROM emp;
```

COUNT(*)
0

Drop Table:

Drop command is used to remove the object from the database.
The DROP command not only removes the table but also removes table based indexes, privileges and constraints.

At a later stage even if we wanted we will not be able to get the table or its data, as it is permanently removed.



Example:-

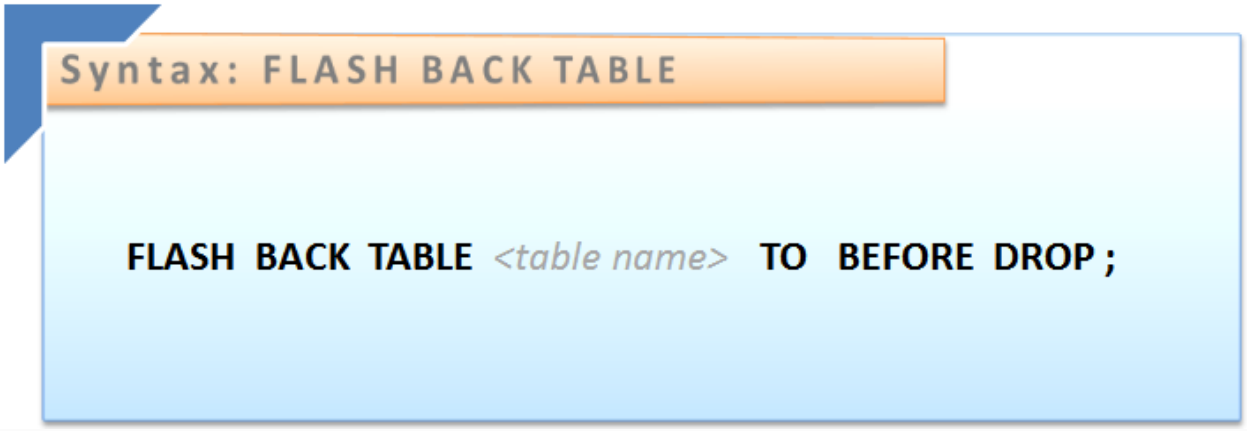
```
SQL>DROP TABLE emp;
```

If table is successfully dropped then we will receive a message saying 'Table dropped'

Even after dropping the emp table whenever we are try to select the emp table then system throws an error saying 'table or view does not exist'.

```
Error at line 1:
ORA-00942: table or view does not exist
```

From Oracle 10g version on wards we can get back the table which we dropped using FLASH BACK, here is the following syntax for that:

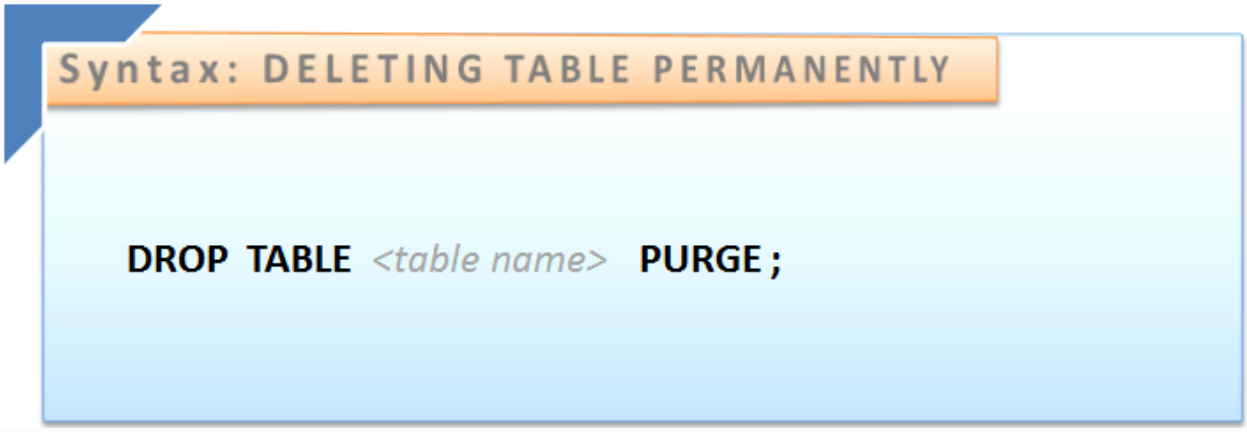


Example:-

```
SQL>FLASH BACK emp TO BEFORE DROP;
```

Now if we try to use the SELECT command we can see the data from 'emp' table.

This process is similar like our recycle bin in our Operating Systems. It means from Oracle 10g on wards the DROP command will not remove our table permanently it will store in Recycle bin. So therefore if we want to delete the table permanently here is the following syntax which will drop the table permanently and which cannot be retrieved back.



Example:-

```
SQL>DROP TABLE emp PURGE;
```

Now if we try to select the emp table then system throws an error like :

```
Error at line 1:
ORA-00942: table or view does not exist
```

Rename Table:

Rename command is used to rename the object in database. Rename command is part of DDL (Data Definition Language)

Example:-

SQL>**RENAME** emp **TO** na_emp;

Once the table is successfully renamed then we will receive a message saying 'Table renamed'.
– after rename if we try to access 'emp' table system will not recognize the table.

SELECT * **FROM** emp;

Output:-

ORA-00942: **TABLE OR VIEW** does **NOT** exit

SQL>**SELECT** * **FROM** na_emp;

Output:

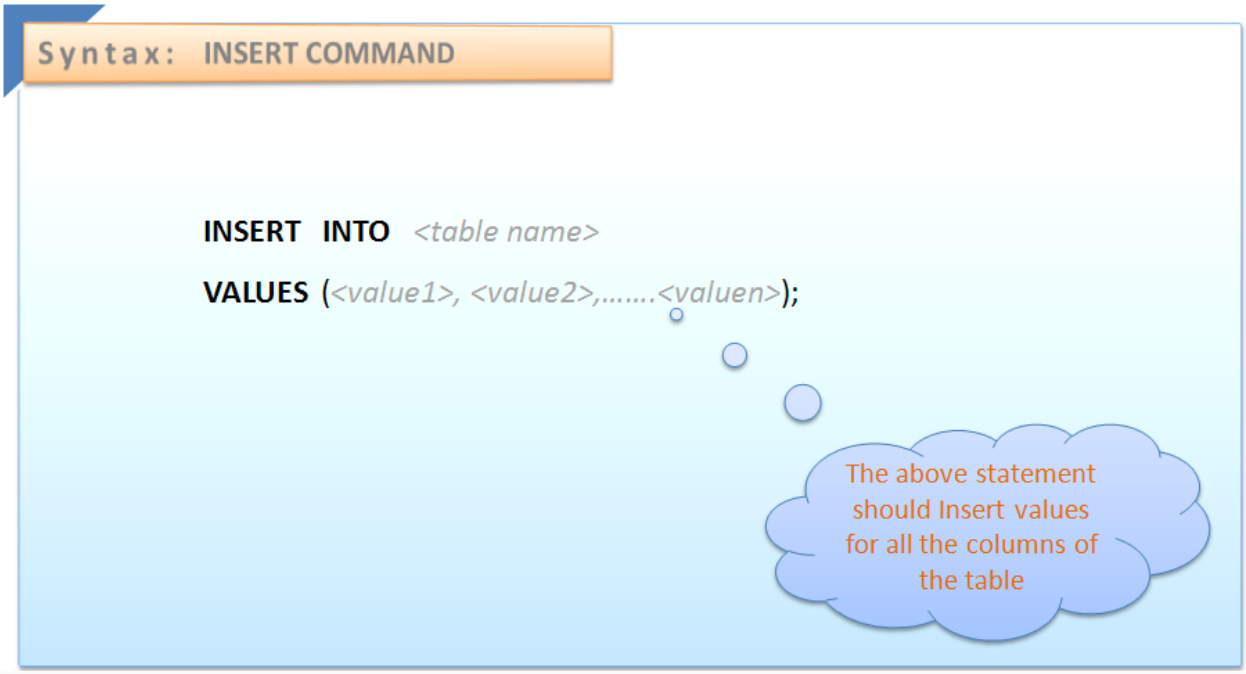
EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7369	SMITH	CLERK	7902	12/17/1980	800		20
7499	ALLEN	SALESMAN	7698	2/20/1981	1600	300	30
7521	WARD	SALESMAN	7698	2/22/1981	1250	500	30
7566	JONES	MANAGER	7839	4/2/1981	2975		20
7654	MARTIN	SALESMAN	7698	9/28/1981	1250	1400	30
7698	BLAKE	MANAGER	7839	5/1/1981	2850		30
7782	CLARK	MANAGER	7839	6/9/1981	2450		10

Chapter 4: DML

Insert Command:

Insert command is part of **DML** (Data Manipulation Language). Insert Statement is used to add new rows of data to an existing database table.

To insert data, table should be first available in the database and below are the ways to insert data into a table:



Note: For better demonstration let's assume EMP table is empty.

Example:

INSERT INTO EMP **VALUES** (7369,'SMITH','CLERK',7902,'10-JUN-1980',800,**NULL**,20);

After successfully executing the above statements then we will receive a message saying

'1 row inserted'

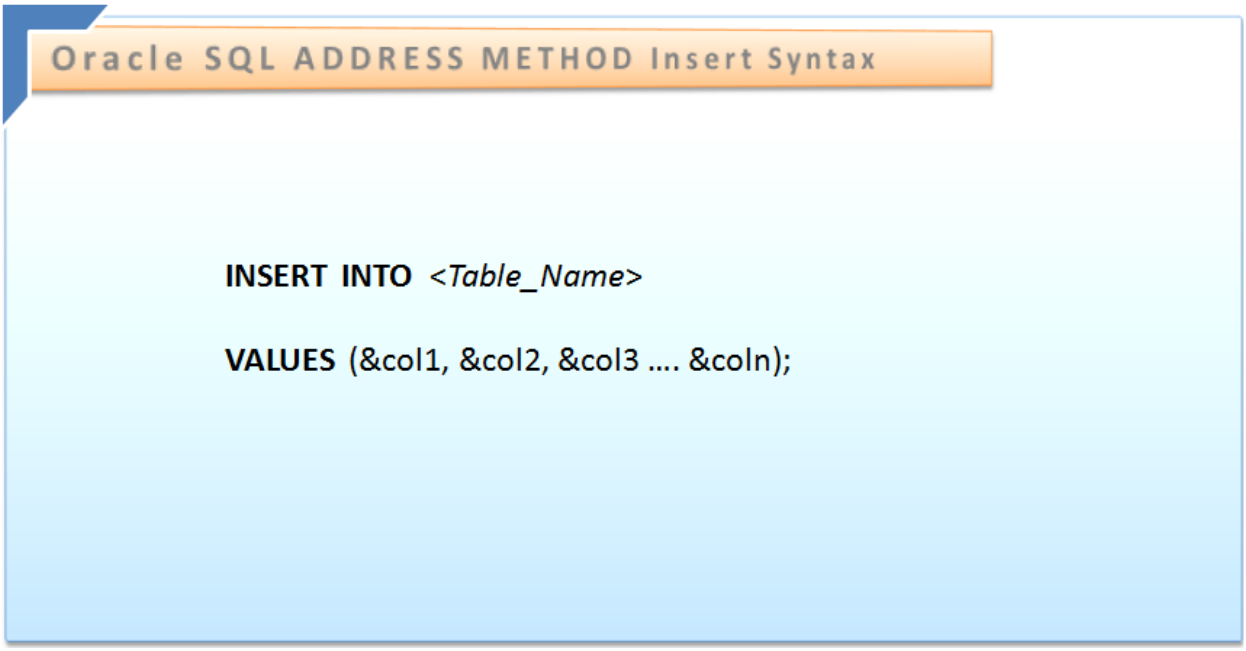
Now let's check the EMP table

SELECT * **FROM** emp

EMPNO	ENAME	JOB	MGR	HIREDATE	SALARY	COMM	DEPTNO
7369	SMITH	CLERK	7902	6/10/1980	800		20

To insert another record we have to type the entire insert statement, if there are lot of records to be inserted then it will become difficult. To avoid this we can use address method.

ADDRESS METHOD



Example:

```
INSERT INTO EMP VALUES
(&empno,'&ename','&job',&mgr_id,'&hire_date',&sal,&comm,&deptno);
```

– After successfully executing the above statement we will see a message saying

```
Enter value for empno:
Enter value for ename:
Enter value for job:
Enter value for mgr_id:
Enter value for hire_date:
Enter value for sal:
Enter value for comm:
Enter value for deptno:
```

System will prompt us to enter data for these variables (&empno,'&ename','&job',&mgr_id,'&hire_date',&sal,&comm,&deptno).
After entering all the values, system will insert records into table

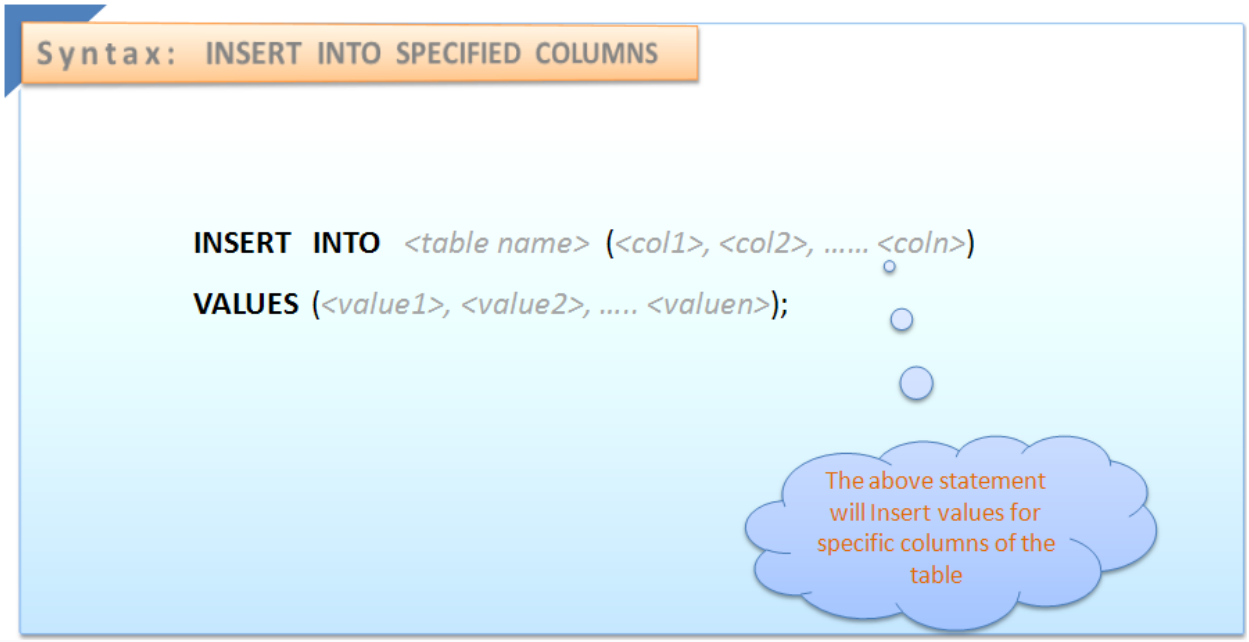
```
Enter value for empno: 7370
Enter value for ename: JACK
Enter value for job: CLERK
Enter value for mgr_id: 7903
Enter value for hire_date: 10-FEB-1981
Enter value for sal: 2000
Enter value for comm: NULL
Enter value for deptno: 10
old   1: INSERT INTO EMP VALUES (&empno,'&ename','&job',&mgr_id,'&hire_date',&sal,&comm,&deptno)
new   1: INSERT INTO EMP VALUES (7370,'JACK','CLERK',7903,'10-FEB-1981',2000,NULL,10)
```

SELECT * **FROM** emp

EMPNO	ENAME	JOB	MGR	HIREDATE	SALARY	COMM	DEPTNO
7369	SMITH	CLERK	7902	6/10/1980	800		20
7370	JACK	CLERK	7903	2/10/1981	2000		10

INSERTING DATA INTO SPECIFIED COLUMNS USING VALUE METHOD

Using this process we can insert the data into the table for specific columns.



Example:-

INSERT INTO emp (empno, ename, job) **VALUES** (1024, 'SCOTT', 'Manager');

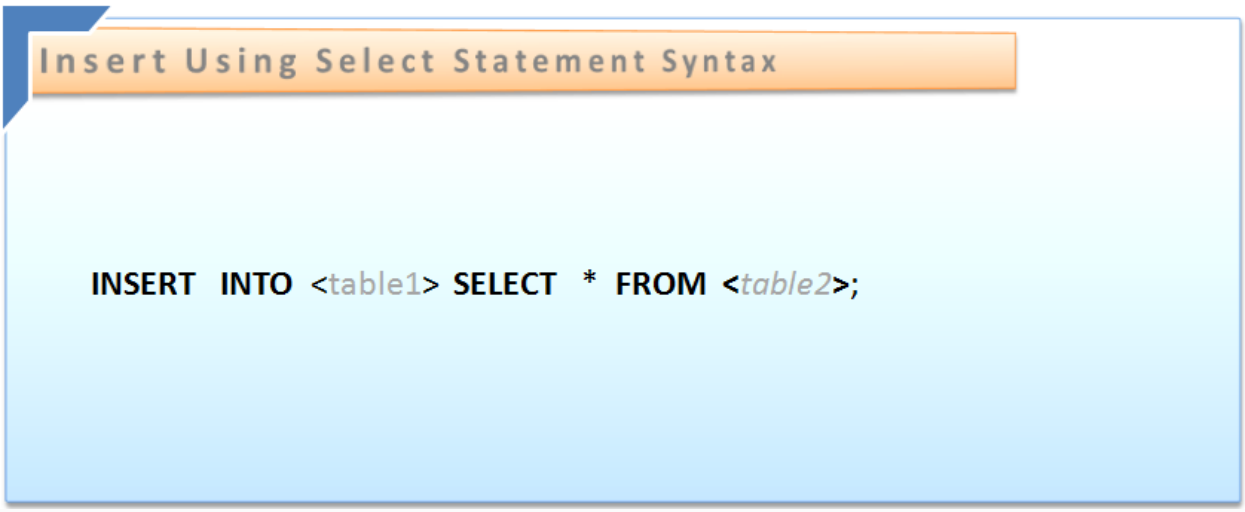
– After entering the values. lets checkout the emp table

SELECT * **FROM** emp;

EMPNO	ENAME	JOB	MGR	HIREDATE	SALARY	COMM	DEPTNO
7369	SMITH	CLERK	7902	6/10/1980	800		20
7370	JACK	CLERK	7903	2/10/1981	2000		10

INSERT USING SELECT STATEMENT

Using this process we can insert existing table data to another table in a single statement. But the table structure should be same.



Example:-

– Let's assume there is a empty table named 'emp1' is available in database.

```
SQL> INSERT INTO emp1 SELECT * FROM emp;
```

– Now check emp1 table data

```
SQL> SELECT * FROM emp1
```

EMPNO	ENAME	JOB	MGR	HIREDATE	SALARY	COMM	DEPTNO
7369	SMITH	CLERK	7902	6/10/1980	800		20
7370	JACK	CLERK	7903	2/10/1981	2000		10
1024	SCOTT	Manager					

Inserting data into specified columns

```
SQL> INSERT INTO emp1(empno, ename)
                        SELECT empno, ename FROM emp;
```

7369	SMITH	CLERK	7902	6/10/1980	800		20
7370	JACK	CLERK	7903	2/10/1981	2000		10
1024	SCOTT	Manager					
7369	SMITH						
7370	JACK						
1024	SCOTT						

INSERT USING INSERT ALL STATEMENT

Insert all statement is used when we wanted to execute multiple insert statements in a single command.

Syntax:-

```
INSERT ALL
    INTO tbl_name VALUES (value1, value2, value3 …. Valuen)
    INTO tbl_name VALUES (value1, value2, value3 …. Valuen)
    INTO tbl_name VALUES (value1, value2, value3 …. Valuen)
    |
    |
    INTO tbl_name VALUES (value1, value2, value3 …. Valuen)
SELECT COLUMNS FROM tbl_name WHERE condition;
```

'Insert all' statement will contain 'select statement' in the end and this select statement defines how many number of iterations should these insert statements get executed.
i.e. if a select statement returns 10 records then all the insert statements in a 'Insert All' command will be executed 10 times.
Assume: IN DEPT table we have following columns and data, however EMP and DEPT table do not have any constraint. Please refer to the table script for more info.

```
DEPTNO    DNAME       LOC
--------- --------        ----
10        Accounting      New York
20        Research        Dallas
30        Sales           Chicago
40        Operations      Boston
```

a) 'INSERT ALL' WITH ALL FIELDS

Using this process we will insert data for all columns in the table.

Before running insert all command lets empty the emp table. (i.e. truncate table emp;)

```
SELECT * FROM emp;
```

EMPNO	ENAME	JOB	MGR	HIREDATE	SALARY	COMM	DEPTNO

```
SELECT * FROM dept;
```

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

```
SQL> INSERT ALL
        INTO emp VALUES(7369,'SMITH','CLERK',7902,'10-JUN-1980',800,NULL,20)
        INTO emp VALUES(7999,'MARK','MGR',NULL,'14-JUN-1970',2000,NULL,30)
        INTO emp VALUES(7100,'SCOTT','CLERK',7902,'10-JUN-1980',900,NULL,40)
      SELECT * FROM dept WHERE deptno=10;
```

Output:

EMPNO	ENAME	JOB	MGR	HIREDATE	SALARY	COMM	DEPTNO
7369	SMITH	CLERK	7902	6/10/1980	800		20
7999	MARK	MGR		6/14/1970	2000		30
7100	SCOTT	CLERK	7902	6/10/1980	900		40

When above 'Insert all' statement is executed then system first runs the select statement.

```
SELECT * FROM dept WHERE deptno=10;
```

Execution process:

'SELECT * FROM dept WHERE deptno=10' statement returns only one record so compiler executes 3 insert statements only once and hence 3 records will be inserted into the emp table.

```
SQL> INSERT ALL
        INTO emp VALUES(7369,'SMITH','CLERK',7902,'10-JUN-1980',800,NULL,20)
        INTO emp VALUES(7999,'MARK','MGR',NULL,'14-JUN-1970',2000,NULL,30)
        INTO emp VALUES(7100,'SCOTT','CLERK',7902,'10-JUN-1980',900,NULL,40)
      SELECT * FROM dept WHERE deptno>20;
```

– this insert all statement inserts 6 rows in emp table as 'SELECT * FROM dept WHERE deptno>20' fetches 2 records (i.e. deptno 30 and 40), so every insert statement is executed twice.

Output:

EMPNO	ENAME	JOB	MGR	HIREDATE	SALARY	COMM	DEPTNO
7369	SMITH	CLERK	7902	6/10/1980	800		20
7369	SMITH	CLERK	7902	6/10/1980	800		20
7999	MARK	MGR		6/14/1970	2000		30
7999	MARK	MGR		6/14/1970	2000		30
7100	SCOTT	CLERK	7902	6/10/1980	900		40
7100	SCOTT	CLERK	7902	6/10/1980	900		40

b) 'INSERT ALL' BASED ON CONDITION

Using this process, Insert statements are executed based on conditions.

```
SQL>INSERT ALL
      WHEN deptno > 10 THEN
      INTO emp VALUES(7369,'SMITH','CLERK',7902,'10-JUN-1980',800,NULL,20)
      WHEN dname = 'SALES' THEN
      INTO emp VALUES(7999,'MARK','MGR',NULL,'14-JUN-1970',2000,NULL,30)
      WHEN loc = 'XXX' THEN
      INTO emp VALUES(7100,'SCOTT','CLERK',7902,'10-JUN-1980',900,NULL,40)
    SELECT * FROM dept WHERE deptno > 10;
```

– After execution of the above insert all statement, total 4 rows inserted in the emp table.

EMPNO	ENAME	JOB	MGR	HIREDATE	SALARY	COMM	DEPTNO
7369	SMITH	CLERK	7902	6/10/1980	800		20
7369	SMITH	CLERK	7902	6/10/1980	800		20
7369	SMITH	CLERK	7902	6/10/1980	800		20
7999	MARK	MGR		6/14/1970	2000		30

Execution process:

Compiler first executes 'SELECT * FROM dept WHERE deptno > 10' which will returns 3 records.(i.e. 3 iterations for insert statements)

- Condition 'WHEN deptno > 10' becomes true for every record of select statement so first insert statement i.e. 'INTO emp values(7369,'SMITH','CLERK',7902,'10-JUN-1980',800,NULL,20)' executes 3 times inserting 3 records into emp table.
- Condition 'WHEN deptno = 'SALES' becomes true for only one record of select statement so second insert statement i.e. 'INTO emp values(7999,'MARK','MGR',NULL,'14-JUN-1970',2000,NULL,30') executes only once so inserting only one record into emp table
- Condition 'WHEN loc = 'XXX' becomes false for every record of select statement so this insert statement will never get executed

c) 'INSERT ALL' INTO MULTIBLE TABLES

Using this process, we can insert into multiple tables at once.

To demonstrate Insert All into multiple tables, I have created 3 empty tables emp1, emp2, emp3.

```
SELECT * FROM emp1
```

EMPNO	ENAME	JOB	MGR	HIREDATE	SALARY	COMM	DEPTNO

```
SELECT * FROM emp2
```

EMPNO	ENAME	JOB	MGR	HIREDATE	SALARY	COMM	DEPTNO

SELECT * **FROM** emp3

EMPNO	ENAME	JOB	MGR	HIREDATE	SALARY	COMM	DEPTNO

– In this insert all we are trying to insert into different tables i.e. emp1, emp2, emp3.

```
SQL>INSERT ALL
        INTO emp1 VALUES(7369,'SMITH','CLERK',7902,'10-JUN-1980',800,NULL,20)
        INTO emp2 VALUES(7100,'SCOTT','CLERK',7902,'10-JUN-1980',900,NULL,40)
        INTO emp3 VALUES(7100,'SCOTT','CLERK',7902,'10-JUN-1980',900,NULL,40)
      SELECT * FROM dept WHERE deptno=10;
```

Ouput:

– The above statement inserts 3 rows but into different tables.

SELECT * **FROM** emp1

EMPNO	ENAME	JOB	MGR	HIREDATE	SALARY	COMM	DEPTNO
7369	SMITH	CLERK	7902	6/10/1980	800		20

SELECT * **FROM** emp2

EMPNO	ENAME	JOB	MGR	HIREDATE	SALARY	COMM	DEPTNO
7100	SCOTT	CLERK	7902	6/10/1980	900		40

SELECT * **FROM** emp3

EMPNO	ENAME	JOB	MGR	HIREDATE	SALARY	COMM	DEPTNO
7100	SCOTT	CLERK	7902	6/10/1980	900		40

Update Command:

Update command is part of DML (Data Manipulation Language).Update statement is used to modify the existing table data.

Syntax: UPDATING TABLE

```
UPDATE <table name>
SET <col1>=<value1>, <col2>=<value2>, .... <coln>=<valuen>
WHERE <condition> ;
```

Specifying condition is not mandatory and if condition is not specified then system will consider all records in the table for update.

Before Updating table see the details of employee:-

SQL> **SELECT** * **FROM** emp;

EMPNO	ENAME	JOB	MGR	HIREDATE	SALARY	COMM	DEPTNO
7369	SMITH	CLERK	7902	6/10/1980	800		20
7370	JACK	CLERK	7903	2/10/1981	2000		10
1024	SCOTT	Manager					

SQL> **UPDATE** emp **SET** sal = 500;

– As we did not specify any condition the above statement will update the entire table.

After Updating the table now let's see the difference selecting the details of employee:-

SQL> **SELECT** * **FROM** emp;

EMPNO	ENAME	JOB	MGR	HIREDATE	SALARY	COMM	DEPTNO
7369	SMITH	CLERK	7902	6/10/1980	800		20
7370	JACK	CLERK	7903	2/10/1981	2000		10
1024	SCOTT	Manager					

Updating Table with WHERE clause:

Before Update:

SQL> **SELECT** * **FROM** emp;

EMPNO	ENAME	JOB	MGR	HIREDATE	SALARY	COMM	DEPTNO
7369	SMITH	CLERK	7902	6/10/1980	800		20
7370	JACK	CLERK	7903	2/10/1981	2000		10
1024	SCOTT	Manager					

SQL> **UPDATE** emp **SET** sal = 500 **WHERE** JOB = 'CLERK';

– The above statement will update only the employee records whose JOB = 'CLERK'.

After Update, you can observe the difference seeing the employee table details:-

SQL> **SELECT** * **FROM** emp

EMPNO	ENAME	JOB	MGR	HIREDATE	SALARY	COMM	DEPTNO
7369	SMITH	CLERK	7902	6/10/1980	500		20
7370	JACK	CLERK	7903	2/10/1981	500		10
1024	SCOTT	Manager					

Updating multiple columns data:

Before Update:

SQL> **SELECT** * **FROM** emp;

EMPNO	ENAME	JOB	MGR	HIREDATE	SALARY	COMM	DEPTNO
7369	SMITH	CLERK	7902	6/10/1980	800		20
7370	JACK	CLERK	7903	2/10/1981	2000		10
1024	SCOTT	Manager					

SQL> **UPDATE** emp **SET** sal = 500, ename = 'SMITH' **WHERE** empno = 7369;

– The above statement will update only one employee record whose empno = 7369.

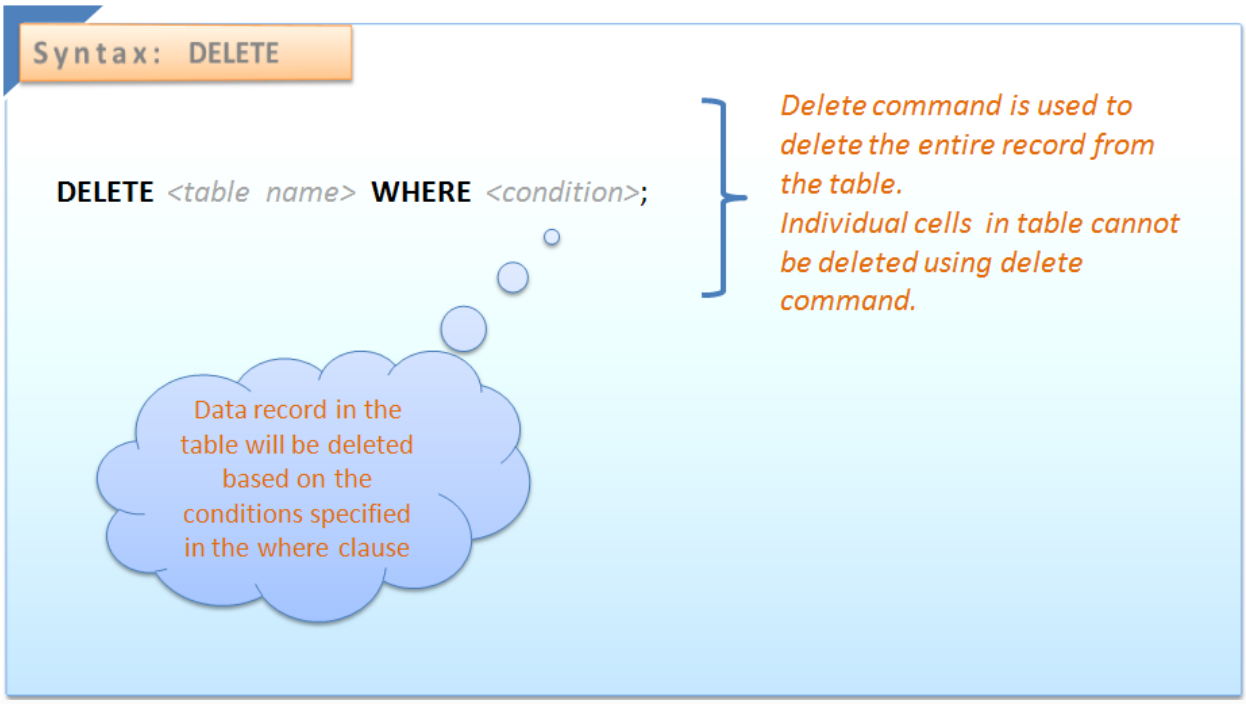
After Update:

SQL> **SELECT** * **FROM** emp;

EMPNO	ENAME	JOB	MGR	HIREDATE	SALARY	COMM	DEPTNO
7369	SMITH	CLERK	7902	6/10/1980	800		20
7370	JACK	CLERK	7903	2/10/1981	2000		10
1024	SCOTT	Manager					

Delete Command:

Delete Command is part of DML (Data Manipulation Language). Delete statement is used to *delete specific data rows* in the table. It means **DELETE** command is used to delete *only rows* in the tables. If we want to delete specific rows then we must mention in the WHERE clause using some condition.



In Delete command using a condition in where clause, we can delete required records from the table.

Before Delete, you can see the Employee details as follows:-

SQL> **SELECT** * **FROM** emp;

EMPNO	ENAME	JOB	MGR	HIREDATE	SALARY	COMM	DEPTNO
7369	SMITH	CLERK	7902	6/10/1980	800		20
7370	JACK	CLERK	7903	2/10/1981	2000		10
1024	SCOTT	Manager					

SQL> **DELETE** emp **WHERE** empno = 7369;

– The above statement will delete only one employee record whose empno = 7369.

After Deleting, row now check the Employee Details, you can observe that Employee with employee number 7369 details are removed from the table.

SQL> **SELECT** * **FROM** emp;

EMPNO	ENAME	JOB	MGR	HIREDATE	SALARY	COMM	DEPTNO
7370	JACK	CLERK	7903	2/10/1981	2000		10
1024	SCOTT	Manager					

If we don't specify any condition then the below statement will delete entire table. For Example, before deleting the Employee details you can observe the employee details as follows:-

SQL> **SELECT** * **FROM** emp;

EMPNO	ENAME	JOB	MGR	HIREDATE	SALARY	COMM	DEPTNO
7369	SMITH	CLERK	7902	6/10/1980	800		20
7370	JACK	CLERK	7903	2/10/1981	2000		10
1024	SCOTT	Manager					

Now apply the DELETE command on table as follows:-

SQL> **DELETE** emp;

– As we did not specify any condition the above statement will delete the entire table data.

After Deleting, see the details of employee table **:-**

SQL> **SELECT** * **FROM** emp;

EMPNO	ENAME	JOB	MGR	HIREDATE	SALARY	COMM	DEPTNO

Merge Command:

Merge command is used to perform insert and update in a single command. It will make our job easy instead of running multiple insert and update commands. Merge statement is introduced in oracle release 9i.

Syntax: MERGE SYNTAX

```
MERGE INTO  <Target Table> tgt
USING <Source Table> src
ON ( src.object_id = tgt.object_id )
WHEN MATCHED THEN
   UPDATE
    SET tgt.object_name = src.object_name,
        tgt.object_type = src.object_type
WHEN NOT MATCHED THEN
INSERT (tgt.object_id , tgt.object_name , tgt.object_type )
VALUES (src.object_id , src.object_name ,src.object_type);
```

- **INTO:** this is how we specify the target for the MERGE. The target must be either a table or an updateable view (an in-line view cannot be used here);
- **USING:** the USING clause represents the source dataset for the MERGE. This can be a single table (as in our example) or an in-line view;
- **ON ( ) :** the ON clause is where we supply the join between the source dataset and target table. Note that the join conditions must be in parentheses;
- **WHEN MATCHED:** this clause is where we instruct Oracle on what to do when we already have a matching record in the target table (i.e. there is a join between the source and target datasets). We obviously want an UPDATE in this case. One of the restrictions of this clause is that we cannot update any of the columns used in the ON clause (though of course we don't

need to as they already match). Any attempt to include a join column will raise an unintuitive invalid identifier exception; and

- **WHEN NOT MATCHED:** this clause is where we INSERT records for which there is no current match.

Oracle Merge Command **Example**:-

```
SQL>MERGE
    INTO emp1 e1
    USING emp2 e2
    ON (e1.empno=e2.empno)
    WHEN MATCHED
    THEN
    UPDATE
    SET sal = e2.sal
    WHEN NOT MATCHED
    THEN
    INSERT (e1.empno, e1.ename, e1.sal)
    VALUES (e2.empno, e2.ename, e2.sal);
```

In the above example, two tables are with the same structure but it is not necessary that we should use same structured tables we can even merge tables with different structure as well but the data type of the columns should be same.

Chapter 5: Constraints

- Constraint is a rule or restriction which is imposed on the table data.
- For a new table, Constraints can be specified in the create table syntax.
- For an existing table in the database, Constraints can be added by alter table syntax.
- Constraints are used to enforce business rules on data.

Why constraints are required?

- We use constraints, if we want to enforce rules whenever a row is inserted, updated or deleted from that table. I.e. For every operation on the table data to succeed, constraint must be satisfied first.
- To prevent the deletion of a table data if it has dependencies on other tables.
- Allow or restrict null values.
- Ensure unique values for columns.
- Ensure a primary identifying value for each row in a table.

Constraints are categorized as follows.

Domain integrity constraints

- Not null
- Check

Entity integrity constraints

- Unique
- Primary key

Referential integrity constraints

- Foreign key

There are three different ways to add constraints:

- Column level – along with the column definition
- Table level – after the table definition
- Alter level – using alter command

Whatever is the way we choose (Column level / Table level / alter level), Constraints are always attached to a column not to a table.

Constraint names:

Constraint names must be unique to the owner.

- The constraint name appears in messages when the constraint is violated.
- If a constraint name is not specified, the Oracle Server assigns a unique name with the format SYS_C00n.
- A suggested format for naming a constraint is

"TABLENAME_CONSTRAINT TYPE_COLNAME" example:- EMP_NN_SAL.
While adding constraints we need not specify the name, oracle will internally generate the name of the constraint.

If we want to give a name to the constraint then we have to use the **constraint** clause along with constraint name.

System tables for viewing constraints:-

- **user_constraints**:-

Information about columns in constraint definitions
Owned by the user

Syntax: USER CONSTRAINTS

```
SELECT  constraint_name, constraint_type, search_condition
FROM USER_CONSTRAINTS
WHERE table_name=<table name>;
```

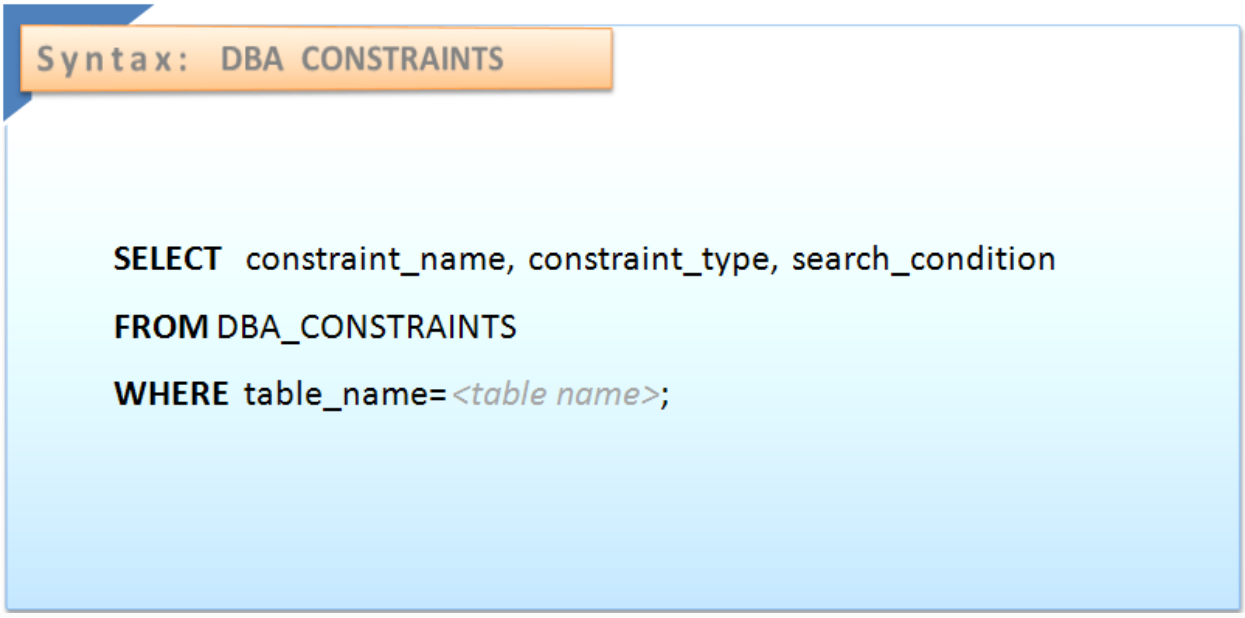
- **all_constraints**:-

Constraint definitions on accessible tables.

Syntax: ALL CONSTRAINTS

```
SELECT  constraint_name, constraint_type, search_condition
FROM ALL_CONSTRAINTS
WHERE table_name=<table name>;
```

- **dba_constraints**:-
  Constraint definitions on all tables.



Not Null – Constraint:

- 'NOT NULL' constraints are used to avoid null values for a specific column.
- We can add 'NOT NULL' constraint at column level only.
- Columns without the 'NOT NULL' constraint allow NULL values. (i.e. if no value is specified then NULL is the default) .

If Constraint name is not specified then system will generate a unique name for the constraint.

Constraint without name

```
SQL>CREATE TABLE emp (empno  NUMBER NOT NULL,
   ename   VARCHAR2 (100),
   sal     NUMBER (5));
```

If we want to give a name to the constraint, we have to use the **constraint** clause.

Constraint with name

```
SQL>CREATE TABLE emp (empno  NUMBER CONSTRAINT emp_nn NOT NULL,
   ename   VARCHAR2 (100),
```

```
  sal     NUMBER (5));
```

By using below query we can know the name of the constraint attached to a particular a table.

```
SQL>SELECT constraint_name,
   constraint_type,
   search_condition
   FROM user_constraints
   WHERE TABLE_NAME='EMP';
```

Test Constraint:-

Suppose if we try to insert null values into table column which has NOTNULL constraint then system will raise an error.

```
SQL> INSERT INTO emp (empno, ename, sal)
          VALUES (NULL,'MARK', 2000);
```

Output:

```
ERROR at line 1:
ORA-01400: cannot INSERT NULL into ("EMP"."EMPNO")
```

Check – Constraint:

Check constraint is used to insert the values based on specified condition. The Check constraint explicitly defines a condition that each row must satisfy.

We can add this constraint in all three levels.

- **Column level**
- **Table level**
- **Alter level**

COLUMN LEVEL

In the below example we are trying to add a check constraint on salary (i.e. we are allowed to insert a record only if salary is greater than 10000).

Note: If we do not specify name for the constraint then system will automatically assign a system generated name.

Constraint without name

```
SQL>CREATE TABLE emp (empno  NUMBER,
   sal NUMBER (5) CHECK (sal<10000),
   ename   VARCHAR2 (100));
```

We can specify the constraint name along with the constraint clause.

Constraint with name

```
SQL>CREATE TABLE emp (empno  NUMBER,
   sal NUMBER (5) CONSTRAINT emp_ck CHECK (sal<10000),
   name   VARCHAR2 (100));
```

TABLE LEVEL

Constraint without name

```
SQL>CREATE TABLE emp (empno  NUMBER,
   ename   VARCHAR2 (100),
   sal     NUMBER (5),
   CHECK (sal<10000));
```

Constraint with name

```
SQL>CREATE TABLE emp (empno  NUMBER,
   ename   VARCHAR2 (100),
   sal     NUMBER (5),
   CONSTRAINT emp_ck CHECK (sal<10000));
```

ALTER LEVEL

We can add constraint to an existing table by using alter command.

```
SQL> ALTER TABLE emp ADD CONSTRAINT emp_ch CHECK (sal<10000)

TABLE altered.
```

To drop a constraint from the existing table:

```
SQL> ALTER TABLE emp DROP CONSTRAINT emp_ch CHECK (sal<10000);

-- Table altered.
```

Test Constraint:-

If we try to insert the 'sal' above 10000 then system will raise an error.

```
SQL> INSERT INTO emp (empno, ename, sal) VALUES (6788,'SQL', 11000);

INSERT INTO emp (empno, ename, sal) VALUES (6788,'SQL', 11000)
*
ERROR at line 1:
ORA-02290: CHECK CONSTRAINT (SCOTT.EMP_CH) violated
```

Unique – Constraint:

Unique constraint is used to avoid duplicates values in the column data but it allows null values.

We use unique constraint to ensure that our table column has unique information.

We can add this constraint in all three levels

- **Column Level**
- **Table Level**
- **Alter Level**.

Note :- Whatever is the level we add a constraint, impact remains the same. Constraint always gets assigned to the column level.

COLUMN LEVEL

If we do not specify name for the constraint then system will automatically assign a system generated name.

Constraint without name

```
SQL>CREATE TABLE emp1 (empno NUMBER UNIQUE,
   ename  VARCHAR2 (100),
   sal UMBER (5));
```

We can specify the constraint name when defining a constraint.

Constraint with name

```
SQL>CREATE TABLE emp2 (empno NUMBER CONSTRAINT emp_ck UNIQUE,
   ename  VARCHAR2 (100),
   Sal    NUMBER (5));
```

TABLE LEVEL

Constraint without name

```
SQL>CREATE TABLE emp3 (empno NUMBER,
   ename  VARCHAR2 (100),
   sal    NUMBER (5),
   UNIQUE (empno)
  );
```

Constraint with name

```
SQL>CREATE TABLE emp4 (empno NUMBER,
   ename  VARCHAR2 (100),
   sal    NUMBER (5),
   CONSTRAINT emp_un UNIQUE (empno));
```

ALTER LEVEL

We can add constraint to an existing table using alter command.

```
SQL> ALTER TABLE emp ADD CONSTRAINT emp_un UNIQUE (empno);

-- Table altered.
```

To drop a constraint:

```
SQL> ALTER TABLE emp DROP CONSTRAINT emp_un UNIQUE (empno);

-- Table altered.
```

Test Constraint:-

If we try to insert an employee record with already existing 'empno' then system will raise an error.

```
SQL> INSERT INTO emp (empno, ename, sal)
        VALUES (7788,'SQL', 3000);
```

Output:

```
ERROR at line 1:

ORA-00001: unique constraint (SCOTT.EMP_UN) violated
```

However as it is unique constraint, system will allow null values.

```
SQL> INSERT INTO emp (empno, ename, sal)
        VALUES (NULL,'SQL', 1000);

1 ROW created.
```

Primary Key – Constraint:

- Primary Key is used to avoid duplicates and nulls. This constraint will work as a combination of both unique and not null.
- Only one primary key is allowed to a table.
- For parent child relationships across tables, Primary key is always attached to the parent table.
- If we want more than one column to work as Primary Key columns then those are called composite primary key.
- Note that we can take a maximum of 32 columns in composite primary key and these composite primary key can be declared only at Table level.
- We can add this constraint in all three levels.
  - **Column Level**
  - **Table Level**
  - **Alter Level**

Note: Whatever is the level we add a constraint, impact remains the same. Constraint always gets assigned to the column level.

COLUMN LEVEL

If we do not specify name for the constraint then system will automatically assign a system generated name.

Constraint without name

```
SQL>CREATE TABLE emp1 (empno NUMBER PRIMARY KEY,
  ename VARCHAR2 (100),
```

```
  sal NUMBER (5));
```

Here we mention the Constraint clause along with constraint name.

Constraint with name

```
SQL>CREATE TABLE emp2 (empno NUMBER CONSTRAINT emp_pk PRIMARY KEY,
  ename  VARCHAR2 (100),
  sal    NUMBER (5));
```

TABLE LEVEL

CONSTRAINT without name

```
SQL>CREATE TABLE emp3 (empno NUMBER,
  ename VARCHAR2 (100),
  sal NUMBER (5),
  PRIMARY KEY (empno));
```

Here we mention the Constraint clause along with constraint name in column level.

Constraint with name

```
SQL>CREATE TABLE emp4 (empno NUMBER,
  CONSTRAINT emp_pk PRIMARY KEY (empno),
  ename  VARCHAR2 (100),
  sal    NUMBER (5));
```

COMPOSITE PRIMARY KEY

```
SQL>CREATE TABLE employee (empno NUMBER(5),
                  name varchar2(20),
                  address varchar2(20),
                  PRIMARY KEY(empno,name);
```

ALTER LEVEL

We can add constraint to an existing table by using alter command.

Constraint without name

```
SQL> ALTER TABLE emp1 ADD PRIMARY KEY (empno);

TABLE altered.
```

Constraint with name

```
SQL> ALTER TABLE emp2 ADD CONSTRAINT emp_pk PRIMARY KEY (empno);

TABLE altered.
```

To drop a constraint:

```
SQL> ALTER TABLE emp2 DROP CONSTRAINT emp_pk PRIMARY KEY (empno);

-- Table altered.
```

Test Constraint:-

From the above commands let's assume we have primary key constraint on 'empno' column. If we try to insert an employee record with already existing 'empno' then system will raise an error.

```
SQL> INSERT INTO emp1 (empno, ename, sal) VALUES (7788,'MARK', 3000);
```

Output:

```
ERROR at line 1:

ORA-00001: unique constraint (SCOTT.EMP_UN) violated
```

Even NULL values are not allowed as constraint added is 'Primary Key'.

```
SQL> INSERT INTO emp1 (empno, ename, sal) VALUES (NULL,'MARK', 2000);
```

Output:

```
ERROR at line 1:

ORA-01400: cannot insert NULL into ("SCOTT"."EMP"."EMPNO").
```

Foreign Key – Constraint:

- Foreign keys are used to refer the parent table primary key column which does not allow duplicates.
- The Foreign key constraint provides referential integrity rules (either within a table or between tables).
  i.e. We can only place a value in TABLE B if the values exist as a primary key in TABLE A.
- For parent child relationships across tables, foreign key is always attached to the child table.
- We use foreign key if we want to ensure that for every child table record there is a reference in parent table.
- We can add this constraint in all three levels.

- **Column Level**
- **Table Level**
- **Alter Level**

To demonstrate foreign key constraint we need parent child relationship between the tables. Let's first create the parent/master table (i.e DEPT table) and make deptno column as primary key column.

```
SQL>CREATE TABLE dept (deptno NUMBER (2) PRIMARY KEY,
   dname  VARCHAR2 (10),
   loc VARCHAR2 (10));

TABLE created.
```

COLUMN LEVEL

Based on the parent/master table (DEPT) now let's create child table (i.e. EMP table) and assign the foreign key for the deptno column as shown in below.

```
SQL> CREATE TABLE emp (empno NUMBER (4),
   ename  VARCHAR2 (10),
   sal    NUMBER (5),
   deptno  NUMBER (2) REFERENCES dept(deptno));

TABLE created.
```

Constraint with name

```
SQL> CREATE TABLE emp (empno NUMBER (4),
  ename   VARCHAR2 (10),
  sal     NUMBER (5),
  deptno  NUMBER (2) CONSTRAINT emp_dept_fk REFERENCES dept(deptno));

TABLE created.
```

TABLE LEVEL

We can add foreign constraint at table level.

Constraint without name

```
SQL>CREATE TABLE emp (empno NUMBER,
  ename    VARCHAR2 (100),
  sal      NUMBER (5),
  deptno   NUMBER,
  FOREIGN KEY (deptno) REFERENCES dept (deptno));
```

We can specify the constraint name when defining the constraint.

Constraint with name

```
SQL>CREATE TABLE emp (empno NUMBER,
  Ename   VARCHAR2 (100),
  sal     NUMBER (5),
  deptno  NUMBER,
  CONSTRAINT emp_dept_fk FOREIGN KEY (deptno) REFERENCES dept (deptno));
```

ALTER LEVEL

We can add constraint to an existing table by using alter command.

Constraint without name

```
SQL> ALTER TABLE emp ADD FOREIGN KEY (deptno) REFERENCES dept (deptno);

TABLE altered.
```

Here we mention constraint name along with constraint clause.

Constraint with name

```
SQL> ALTER TABLE emp ADD CONSTRAINT emp_dept_fk FOREIGN KEY (deptno) REFERENCES dept
(deptno);

TABLE altered.
```

Once the primary key and foreign key relationship has been created then we can not remove any parent record if there are dependent records in the child table.

To drop a constraint:

SQL> **ALTER TABLE** emp **DROP CONSTRAINT** emp_dept_fk **FOREIGN KEY** (deptno) **REFERENCES** dept (deptno);

-- Table altered.

Test Constraint:-

From the above example we have created a relationship among the tables EMP and DEPT

SQL>**SELECT** * **FROM** dept;

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

SQL>**SELECT** * **FROM** emp;

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7369	SMITH	CLERK	7902	12/17/1980	840		20
7499	ALLEN	SALESMAN	7698	2/20/1981	1680	300	30
7521	WARD	SALESMAN	7698	2/22/1981	1312.5	500	30
7566	JONES	MANAGER	7839	4/2/1981	3123.75		20
7654	MARTIN	SALESMAN	7698	9/28/1981	1312.5	1400	30
7698	BLAKE	MANAGER	7839	5/1/1981	2992.5		30
7782	CLARK	MANAGER	7839	6/9/1981	2572.5		10
7788	SCOTT	ANALYST	7566	12/9/1982	3150		20
7844	TURNER	SALESMAN	7698	9/8/1981	1575	0	30
7876	ADAMS	CLERK	7788	1/12/1983	1155		20
7900	JAMES	CLERK	7698	12/3/1981	997.5		30
7902	FORD	ANALYST	7566	12/3/1981	3150		20
7934	MILLER	CLERK	7782	1/23/1982	1365		10

Now if we try to insert the values into “emp” table with deptno=60 (which is not avilable in parent table) then system will raise an error

SQL> **INSERT INTO** emp (empno, ename, deptno) **VALUES** (3578,'MARK', 60);

Output:

ERROR at line 1:

ORA-02291: integrity constraint (APPS.EMP_FK) violated - parent key not found

ON Delete Cascade:

'ON DELETE CASCADE' is an extension to 'Foreign key' constraint. If a parent-child relationship (foreign key relationship) exists between the tables then we will not be able delete the parent record when child exists for it. (I.e. we cannot delete Deptno = 10 record from DEPT table when there are employees available in EMP table for Deptno = 10).

By using 'ON DELETE CASCADE' clause we can remove the parent record even it Childs exists. Because with 'ON DELETE CASCADE' whenever you remove parent record system automatically removes all its dependent records from child table.

'ON DELETE CASCADE' clause should be used when defining the foreign key constraint in the child table.

We can define these at all three levels

- **Column Level**
- **Table Level**
- **Alter Level**

To demonstrate foreign key constraint we need parent child relationship between the tables. Let's first create the parent/master table (i.e. DEPT table) and make deptno column as primary key column.

```
SQL> CREATE TABLE dept (deptno NUMBER (2) PRIMARY KEY,
    dname  VARCHAR2 (10),
    loc    VARCHAR2 (10));

Output:-

TABLE created.
```

COLUMN LEVEL

Based on the parent/master table (DEPT) now let's create child table (i.e. EMP table) and assign the foreign key for the deptno column as shown in below.
When assigning foreign key, add key word 'ON DELETE CASCADE' in the end. This will ensure that child records will be deleted when deleting a parent record.
If we don't use 'On DELETE CASCADE' command then we will not be able to delete the parent records at all.

Constraint without name

```
SQL> CREATE TABLE emp (empno NUMBER (4),
    ename   VARCHAR2 (10),
    sal     NUMBER (5),
```

```
   deptno  NUMBER (2) REFERENCES dept(deptno) ON DELETE CASCADE) ;

Output:-

TABLE created.
```

Constraint with name

```
SQL> CREATE TABLE emp (empno NUMBER (4),
   ename  VARCHAR2 (10),
   sal    NUMBER (5),
   deptno  NUMBER (2) CONSTRAINT emp_dept_fk REFERENCES dept(deptno) ON DELETE
CASCADE);

Output:-

TABLE created.
```

TABLE LEVEL

We can add foreign constraint at table level.

Constraint without name

```
SQL>CREATE TABLE emp (empno NUMBER PRIMARY KEY,
   ename  VARCHAR2 (100),
   sal    NUMBER (5),
   deptno NUMBER,
   FOREIGN KEY (deptno) REFERENCES dept (deptno) ON DELETE CASCADE);
```

We can specify the constraint name when defining the constraint

Constraint with name

```
SQL> CREATE TABLE emp (empno NUMBER PRIMARY KEY,
   ename  VARCHAR2 (100),
   sal    NUMBER (5),
   deptno NUMBER,
   CONSTRAINT emp_dept_fk FOREIGN KEY (deptno) REFERENCES dept (deptno) ON DELETE
CASCADE);
```

ALTER LEVEL

We can add constraint to an existing table using alter command.

Constraint without name

SQL> **ALTER TABLE** emp **ADD FOREIGN KEY** (deptno) **REFERENCES** dept (deptno) **ON DELETE** CASCADE;

TABLE altered.

Constraint with name

SQL> **ALTER TABLE** emp **ADD CONSTRAINT** emp_dept_fk **FOREIGN KEY** (deptno)
REFERENCES dept (deptno) **ON DELETE** CASCADE;

Output:-

TABLE altered.

Test Constraint:-

From the above commands, let's assume we now have a 'foreign key' relationship between EMP and DEPT table with 'ON DELETE CASCADE'.

SQL>**SELECT** * **FROM** dept;

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

SQL>**SELECT** * **FROM** emp;

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7369	SMITH	CLERK	7902	12/17/1980	840		20
7499	ALLEN	SALESMAN	7698	2/20/1981	1680	300	30
7521	WARD	SALESMAN	7698	2/22/1981	1312.5	500	30
7566	JONES	MANAGER	7839	4/2/1981	3123.75		20
7654	MARTIN	SALESMAN	7698	9/28/1981	1312.5	1400	30
7698	BLAKE	MANAGER	7839	5/1/1981	2992.5		30
7782	CLARK	MANAGER	7839	6/9/1981	2572.5		10
7788	SCOTT	ANALYST	7566	12/9/1982	3150		20
7844	TURNER	SALESMAN	7698	9/8/1981	1575	0	30
7876	ADAMS	CLERK	7788	1/12/1983	1155		20
7900	JAMES	CLERK	7698	12/3/1981	997.5		30
7902	FORD	ANALYST	7566	12/3/1981	3150		20
7934	MILLER	CLERK	7782	1/23/1982	1365		10

Now let's try to delete the record from master table (Deptno = 10 from DEPT table) and check whether corresponding child records get deleted or not.

```
SQL> DELETE FROM dept
   WHERE deptno=10;

 /* Now Issue the commit */

SQL> Commit;
SQL> SLECT * FROM dept;
```

DEPTNO	DNAME	LOC
20	RESEARCH	DALLAS
30	SALES	CHICAGO
40	OPERATIONS	BOSTON

/* Parent record is deleted, let's check the child table */

```
SQL> SELECT * FROM emp;
```

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7369	SMITH	CLERK	7902	12/17/1980	800		20
7499	ALLEN	SALESMAN	7698	2/20/1981	1600	300	30
7521	WARD	SALESMAN	7698	2/22/1981	1250	500	30
7566	JONES	MANAGER	7839	4/2/1981	2975		20
7654	MARTIN	SALESMAN	7698	9/28/1981	1250	1400	30
7698	BLAKE	MANAGER	7839	5/1/1981	2850		30
7788	SCOTT	ANALYST	7566	12/9/1982	3000		20
7844	TURNER	SALESMAN	7698	9/8/1981	1500	0	30
7876	ADAMS	CLERK	7788	1/12/1983	1100		20
7900	JAMES	CLERK	7698	12/3/1981	950		30
7902	FORD	ANALYST	7566	12/3/1981	3000		20

/* from the output we can observe that all the employee records whose deptno is 10 in EMP table are deleted automatically */

Composite Keys:

- A composite key is a constraint that is created on combination of columns.
- Composite key can be defined in table and alter levels only. (no column level)
- Composite key is not a system keyword, it is just a name given to the constraints which are created on multiple columns of a table.
  Example: like creating a primary key constraint on both EMPNO and ENAME columns of EMP table. (i.e. data combination of both columns EMPNO and ENAME should be unique across the table)
- We can define composite keys on entity integrity and referential integrity constraints.

Composite Unique key:

TABLE LEVEL

Here we are trying to create a composite unique constraint at table level and if we don't specify the name system will automatically generate the name.

Constraint without name

```
SQL>CREATE TABLE emp (empno NUMBER,
   ename   VARCHAR2 (100),
   sal     NUMBER (5),
   UNIQUE (empno, ename));
```

Constraint with name

```
SQL>CREATE TABLE emp (empno NUMBER,
  ename  VARCHAR2 (100),
  sal    NUMBER (5),
  CONSTRAINT emp_cmp_un UNIQUE (empno, ename));
```

ALTER LEVEL

We use alter command if we wanted to add a composite key for an existing table.

Constraint without name

```
SQL> ALTER TABLE emp ADD UNIQUE (empno, ename)

TABLE altered.
```

Constraint with name

```
SQL> ALTER TABLE emp ADD CONSTRAINT emp_cmp_un UNIQUE (empno, ename)

TABLE altered.
```

Test constraint:-

Now let's test the unique composite constraint.
i.e. UNIQUE (empno, ename)

```
SQL>SELECT * FROM emp;
```

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7369	SMITH	CLERK	7902	12/17/1980	840		20
7499	ALLEN	SALESMAN	7698	2/20/1981	1680	300	30
7521	WARD	SALESMAN	7698	2/22/1981	1312.5	500	30
7566	JONES	MANAGER	7839	4/2/1981	3123.75		20
7654	MARTIN	SALESMAN	7698	9/28/1981	1312.5	1400	30
7698	BLAKE	MANAGER	7839	5/1/1981	2992.5		30
7782	CLARK	MANAGER	7839	6/9/1981	2572.5		10
7788	SCOTT	ANALYST	7566	12/9/1982	3150		20
7844	TURNER	SALESMAN	7698	9/8/1981	1575	0	30
7876	ADAMS	CLERK	7788	1/12/1983	1155		20
7900	JAMES	CLERK	7698	12/3/1981	997.5		30
7902	FORD	ANALYST	7566	12/3/1981	3150		20
7934	MILLER	CLERK	7782	1/23/1982	1365		10

Let's try to insert the values empno=7369 and ename = 'SMITH' into emp table which are already available in table.
This scenario will trigger the UNIQUE constraint error.

```
SQL> INSERT INTO emp (empno, ename, deptno) VALUES (7369,'SMITH', 30);
```

Output:

```
ERROR at line 1:

ORA-00001: unique constraint (APPS.EMP_CMP_UN) violated
```

Composite Primary key:

Just as above, we can add composite constraint to the table using Primary key.

TABLE LEVEL

Here we are trying to create a composite primary constraint at table level and if we don't specify the name system will automatically generate the name.

Constraint without name

```
SQL>CREATE TABLE emp (empno NUMBER,
   ename  VARCHAR2 (100),
   sal    NUMBER (5),
   PRIMARY KEY (empno, ename));
```

Constraint with name

```
SQL>CREATE TABLE emp (empno NUMBER,
   ename   VARCHAR2 (100),
   sal     NUMBER (5),
   CONSTRAINT emp_cmp_pk PRIMARY KEY (empno, ename));
```

ALTER LEVEL

We use alter command if we wanted to add a composite key for an existing table.

Constraint without name

```
SQL> ALTER TABLE emp ADD PRIMARY KEY (empno, ename);

TABLE altered.
```

Constraint with name

```
SQL> ALTER TABLE emp ADD CONSTRAINT emp_cmp_pk PRIMARY KEY (empno, ename);

TABLE altered.
```

Test constraint:-

Now let's test the primary key composite constraint.
i.e. PRIMARY KEY (empno, ename)
Example remains same as unique constraint but in Primary key system does not allow null where as unique constraint allows null.

```
SQL>SELECT * FROM emp;
```

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7369	SMITH	CLERK	7902	12/17/1980	840		20
7499	ALLEN	SALESMAN	7698	2/20/1981	1680	300	30
7521	WARD	SALESMAN	7698	2/22/1981	1312.5	500	30
7566	JONES	MANAGER	7839	4/2/1981	3123.75		20
7654	MARTIN	SALESMAN	7698	9/28/1981	1312.5	1400	30
7698	BLAKE	MANAGER	7839	5/1/1981	2992.5		30
7782	CLARK	MANAGER	7839	6/9/1981	2572.5		10
7788	SCOTT	ANALYST	7566	12/9/1982	3150		20
7844	TURNER	SALESMAN	7698	9/8/1981	1575	0	30
7876	ADAMS	CLERK	7788	1/12/1983	1155		20
7900	JAMES	CLERK	7698	12/3/1981	997.5		30
7902	FORD	ANALYST	7566	12/3/1981	3150		20
7934	MILLER	CLERK	7782	1/23/1982	1365		10

Let's try to insert the values empno=7369 and ename = 'SMITH' into emp table which are already available in table.
This scenario will trigger the PRIMARY KEY constraint error.

```
SQL> INSERT INTO emp (empno, ename, deptno) VALUES (7369,'SMITH', 30);
```

Output:

```
ERROR at line 1:

ORA-00001: unique constraint (APPS.EMP_CMP_PK) violated
```

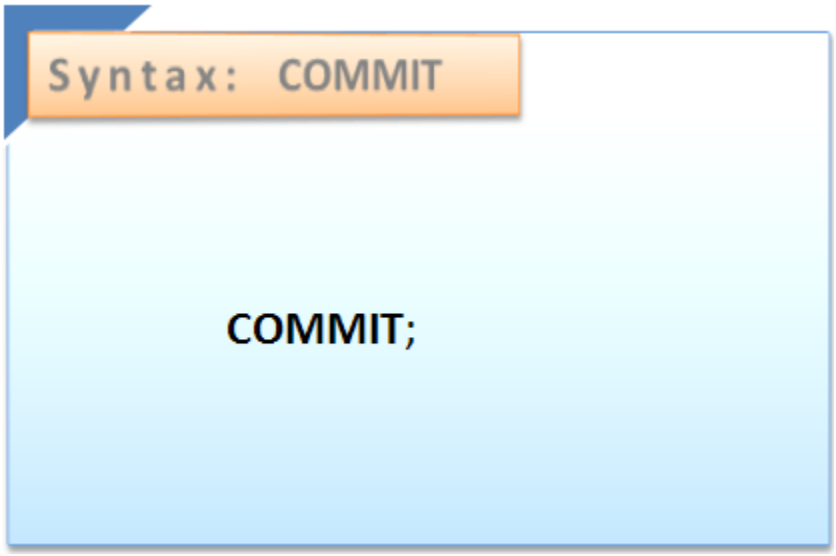
Chapter 6: TCL

Commit:

- **COMMIT** is a part of Transaction Control Language (TCL) that allows us to commit the database transactions i.e. to save the work permanently in the database.
- Commit plays a vital role when we work on DML operations because we need to explicitly save it.

Commit has two types

1. Explicit Commit – User have to Commit explicitly when ever any DML operations like insert, update, delete are performed on the database table.



Example:-For example we are inserting record into emp table Shown below

SQL> **INSERT INTO** emp **VALUES** (7125,'SCOTT','ANALYST', 7902,'22-DEC-1982', 15000, 1000, 20);

Output:

1 **ROW** created

-- *To save the record into the database execute commit.*

SQL> Commit;

Output:

```
Commit complete
```

For some reason we did not commit the record after DML operation then 'if system crashes' or 'if session is ended' – data will be lost from cache. i.e. even we inserted the record we will not be able to see it in the database table.

2. Implicit Commit – system will automatically execute commit whenever we perform any DDL Commands like create, alter, drop.
That is the reason if we perform any DDL command we will not be able to revert back the operation (like using ROLLBACK command).

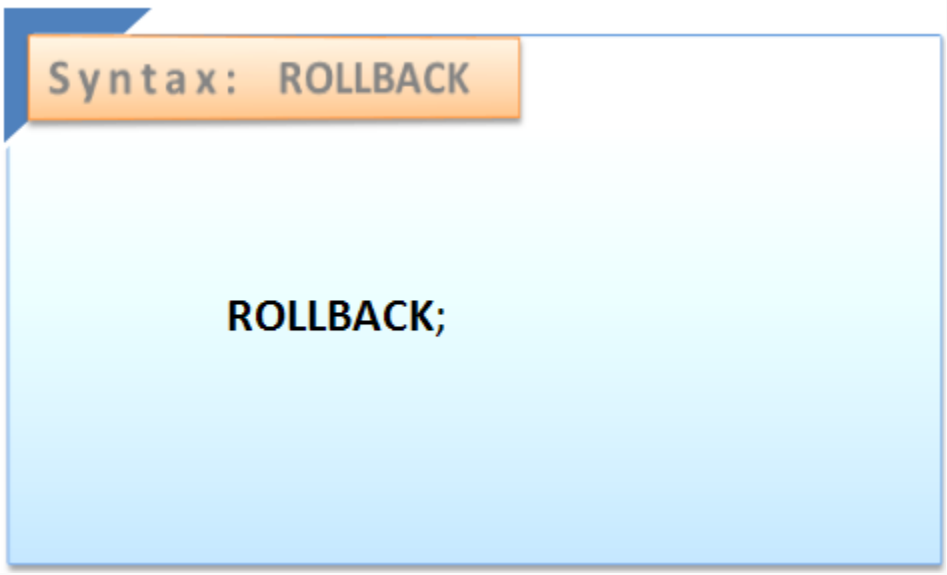
Example:

```
SQL>TRUNCATE TABLE  emp;

-- Data will be permanently deleted from EMP table. (AUTO COMMIT)
```

Rollback:

- ROLLBACK is just opposite to COMMIT.
- ROLLBACK will undo the DML operation that we perform on database which is not committed yet.



Example: – First we are updating the salary of employee record, ename='JOHN'

```
SQL> UPDATE emp SET sal=sal+1000 WHERE ename='JOHN';

 -- 1 row updated
```

After update and inside the same session if we retrieve the record from database using below query we can see the updated salary i.e. '7500'

```
SQL> SELECT *FROM emp WHERE ename='JOHN';
```

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7758	JOHN	ANALYST	7839	1/12/1981	7500	500	10

Now for some reason we wanted to revert the DML operation on the table. (Remember we have not done COMMIT).

- Run ROLLBACK command.

```
SQL> ROLLBACK;

Output :

ROLLBACK complete.
```

We need to explicitly ROLLBACK the transaction if we wanted to revert any DML operation that occurred on the database (Which is not committed yet).

Now let's retrieve the same record i.e. ename='JOHN' to check the salary.

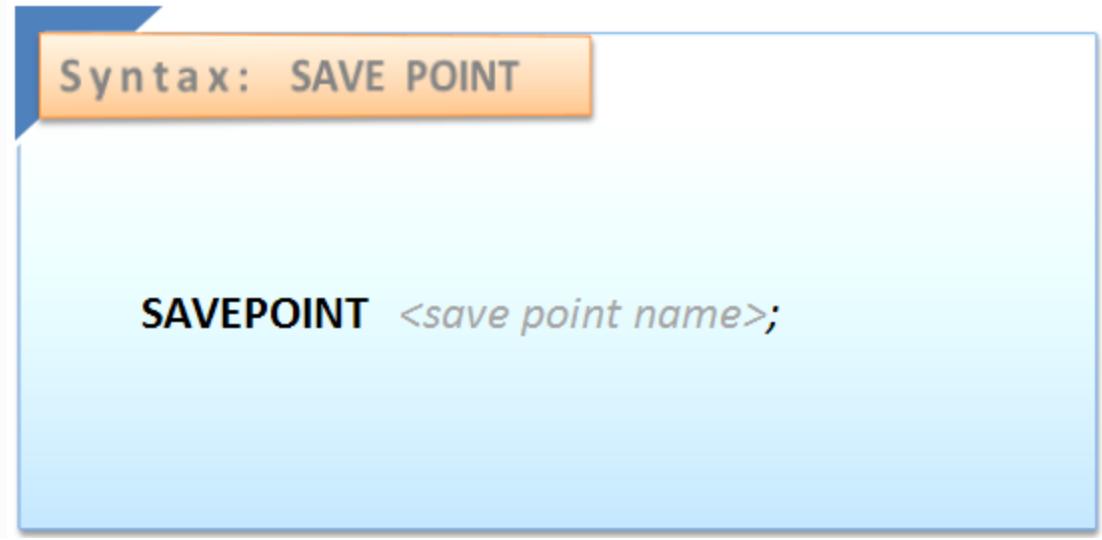
```
SQL> SELECT *FROM emp WHERE ename='JOHN';
```

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7758	JOHN	ANALYST	7839	1/12/1981	7500	500	10

Here we can see that DML operation is Roll backed (Sal from 7500 to 6500).

Save point:

- Save point is a part of Transaction Control Language (TCL).
- We use save points to rollback portions of our current set of transactions.
- These save points are effectively used when we write huge programming code for better control on the program.



Let's assume we have empty EMP table and below is the example with savepoint at different levels

Example: - Inserting records into EMP table

```
SQL>SAVEPOINT s1;

SQL>INSERT INTO emp VALUES (7744,'SMITH','CLERK', 7902,'10-JUN-1980', 800, NULL, 20);

SQL> SAVEPOINT s2;

SQL> INSERT INTO emp VALUES (7745,'MARK','MGR', NULL,'14-JUN-1970', 2000, NULL, 30);

SQL> SAVEPOINT s3;

SQL> INSERT INTO emp VALUES (7746,'SCOTT','CLERK', 7902,'10-JUN-1980', 900, NULL, 40);

SQL> SAVEPOINT s4;

SQL> INSERT INTO emp VALUES (7747,'LUKE','CONS', 7999,'17-JAN-1983', 1500, NULL, 10);
```

Before rollback

SQL>**SELECT** * **FROM** emp;

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7744	SMITH	CLERK	7902	6/10/1980	800		20
7745	MARK	MGR		6/14/1970	2000		30
7746	SCOTT	CLERK	7902	6/10/1980	900		40
7747	LUKE	CONS	7999	1/17/1983	1500		10

ROLLBACK with SAVEPOINT

SQL> **ROLLBACK TO** s3;

This will rollback last two records. i.e. Will rollback all the operations occurred after savepoint s3.

SQL> **SELECT** ***FROM** emp;

– First 2 records are fetched.

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7744	SMITH	CLERK	7902	6/10/1980	800		20
7745	MARK	MGR		6/14/1970	2000		30

Now lets rollback further, i.e. till save point s2

SQL> **ROLLBACK TO** s2;

These will rollback all operations occurred after save point s2.

SQL> **SELECT** ***FROM** emp;

– only first record is fetched.

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7744	SMITH	CLERK	7902	6/10/1980	800		20

Chapter 7: DCL

Grant:

Grant is part of Data Control Language (DCL) that allows us to grant privileges to other users.

We have two types of Grant privileges:

1. Granting permissions to users.
2. Granting privileges on database objects to the users.

To demonstrate DCL lets first create users.

Creating user-

```
SQL> CREATE USER user1 IDENTIFIED BY user1;

--User created
```

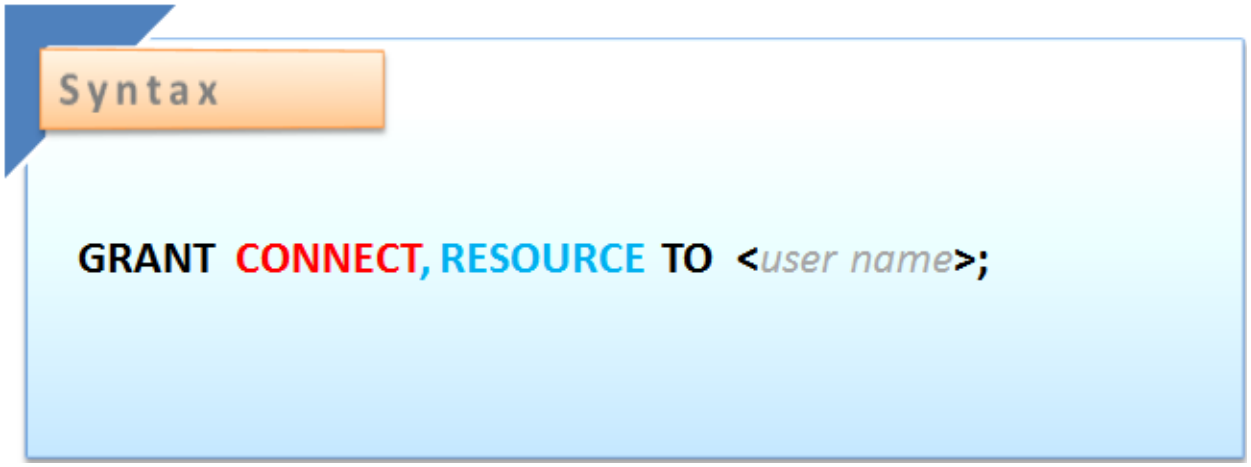
Note: ***Here IDENTIFIED BY will work as password it means for user1 password is user1***

```
SQL> CREATE USER user2 IDENTIFIED BY user2;

--User created
```

Granting permissions to users:-

For every new user created in the database, we need to explicitly provide connection and resource privileges to the user so that user can connect and create objects in the database.



Let's grant the Connect and resource privileges to the users created:-

```
SQL> GRANT CONNECT, RESOURCE TO user1;

-- Grant succeeded

SQL> GRANT CONNECT, RESOURCE TO user2;

-- Grant succeeded
```

User1 and user2 can now connect and create database objects in the database.

Granting privileges on database objects to the users.

Objects created in the database are always owned by a specific user. If a different user has to access the object then necessary privileges should be given.

- Privileges – Type of privilege
- Object name – database object name like EMP.
- User name – User to whom we wanted to provide the privilege.

Example: If user2 wants to access the table created by user1 then user1 should provide the privileges to user2 as user1 owns the object.
We can provide privileges either INDIVIDUALLY or SET OF or ALL privileges.

Individual privileges:-

```
SQL> conn user1/password
```

Let's assume EMP **TABLE IS** owned **BY** user1. **GRANT SELECT** privileges **TO** user2.

SQL> **GRANT SELECT ON** emp **TO** user2;

-- Grant succeeded

If User2 has to access the 'emp' table of user1 then user2 has to use dot operator to access table.

SQL> **SELECT** * **FROM** user1.emp;

SET OF privileges:-

SQL> **GRANT SELECT**, **INSERT**, **UPDATE**, **DELETE ON** emp **TO** user2;

-- Grant succeeded

ALL Privileges:-

SQL> **GRANT ALL ON** emp **TO** user2;

-- Grant succeeded

Privileges with 'GRANT OPTION':-

We use 'With Grant Option' if we want user2 to grant privileges to other users. i.e. User2 will be having admin rights to provide privileges.

SQL> **GRANT ALL ON** emp **TO** user2 **WITH GRANT OPTION**;

-- Grant succeeded

Now user2 can provide grants to user3.

Connect to user2:

SQL> conn user2/password;

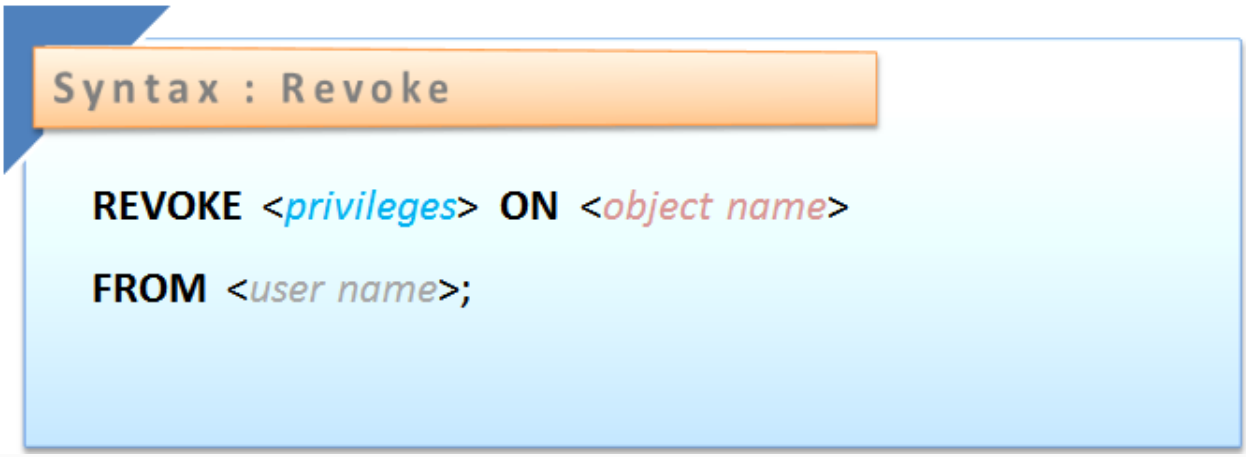
GRANT '**SELECT**' privileges **TO** user3:

SQL> **GRANT SELECT ON** emp **TO** user3;

-- Grant succeeded

Revoke:

REVOKE command is used to revoke the privileges from the users to whom we have already granted the privileges.



We can revoke privileges from user either INDIVIDUALLY or SET OF or ALL privileges.

Let's revoke the privileges that user1 have granted to user2 on EMP table in previous lesson.

Revoke Individual Privileges-

```
SQL> conn user1/password

SQL> REVOKE SELECT ON emp FROM user2

-- Revoke succeeded
```

SET OF privileges:-

```
SQL> REVOKE INSERT, UPDATE, DELETE ON emp FROM user2

-- Revoke succeeded
```

ALL Privileges:-

```
SQL> REVOKE ALL ON emp FROM user2;

-- Revoke succeeded
```

Connect to user2 and check the access to the EMP table.

```
SQL> conn user2/password
```

--If user2 tries to access the table then system will throw an error.

SQL>**SELECT** ***FROM** user1.emp;

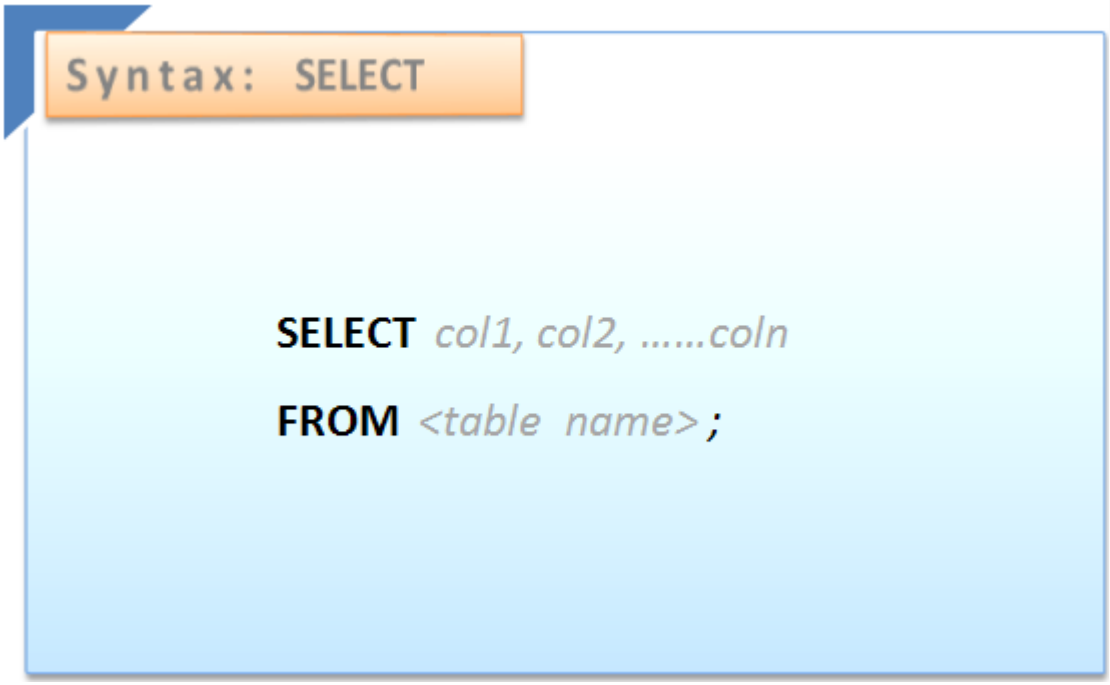
Ouput:
ORA-01031: insufficient privileges

Chapter 8: DRL & Clauses

SELECT:

- SELECT statement is used to retrieve data from a table in the database.
- Retrieval of data can be from specific columns or from all the columns in the table.
- To create a simple SQL SELECT Statement, you must specify the column(s) name and the table name.

By using SELECT we can retrieve either few columns or all columns.



- Table-name is the name of the table from which the information is retrieved.
- Col1, col2, coln means one or more columns of table from which data is retrieved.

Selecting few Columns:-

```
SQL> SELECT empno, ename, job

    FROM emp;
```

Output:-

EMPNO	ENAME	JOB
7369	SMITH	CLERK
7499	ALLEN	SALESMAN
7521	WARD	SALESMAN
7566	JONES	MANAGER
7654	MARTIN	SALESMAN
7698	BLAKE	MANAGER
7782	CLARK	MANAGER
7788	SCOTT	ANALYST
7839	KING	PRESIDENT
7844	TURNER	SALESMAN
7876	ADAMS	CLERK
7900	JAMES	CLERK
7902	FORD	ANALYST
7934	MILLER	CLERK

'**SELECT**' clause in the query specifies columns that we wanted to select.
'**FROM**' clause in the query specifies data sources. If we have multiple data sources then we should specify joins between them.

Selecting ALL COLUMNS:-

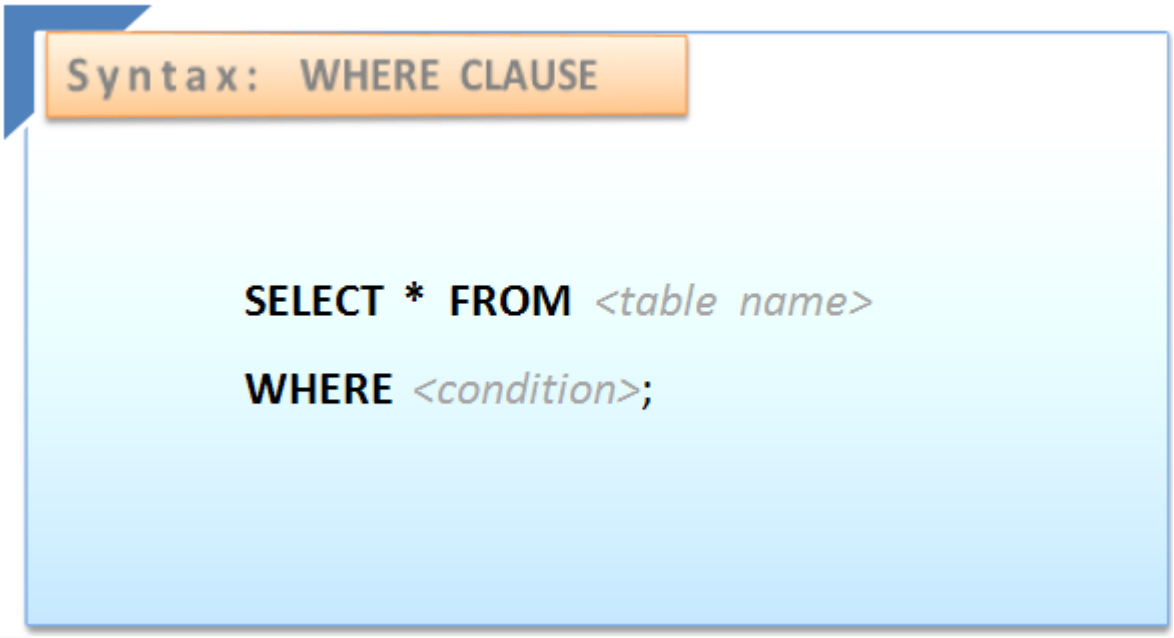
```
SQL> SELECT * FROM emp;
```

Output:-

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7369	SMITH	CLERK	7902	2/20/1981	2400		20
7499	ALLEN	SALESMAN	7698	2/20/1981	2160	300	30
7521	WARD	SALESMAN	7698	2/22/1981	1620	500	30
7566	JONES	MANAGER	7839	4/2/1981	3075		20
7654	MARTIN	SALESMAN	7698	9/28/1981	1620	1400	30
7698	BLAKE	MANAGER	7839	5/1/1981	2950		30
7782	CLARK	MANAGER	7839	6/9/1981	2550		10
7788	SCOTT	ANALYST	7566	12/9/1982	3100		20
7839	KING	PRESIDENT		11/17/1981	5100		10
7844	TURNER	SALESMAN	7698	9/8/1981	2040	0	30
7876	ADAMS	CLERK	7788	1/12/1983	1440		20
7900	JAMES	CLERK	7698	12/3/1981	1260		30
7902	FORD	ANALYST	7566	12/3/1981	3100		20
7934	MILLER	CLERK	7782	1/23/1982	1680		10

Where Clause:

WHERE clause is used to specify the conditions while retrieving or manipulating data.



It is not necessary that we should always fetch all records from the table. Using where clause we can restrict the data by using conditions.

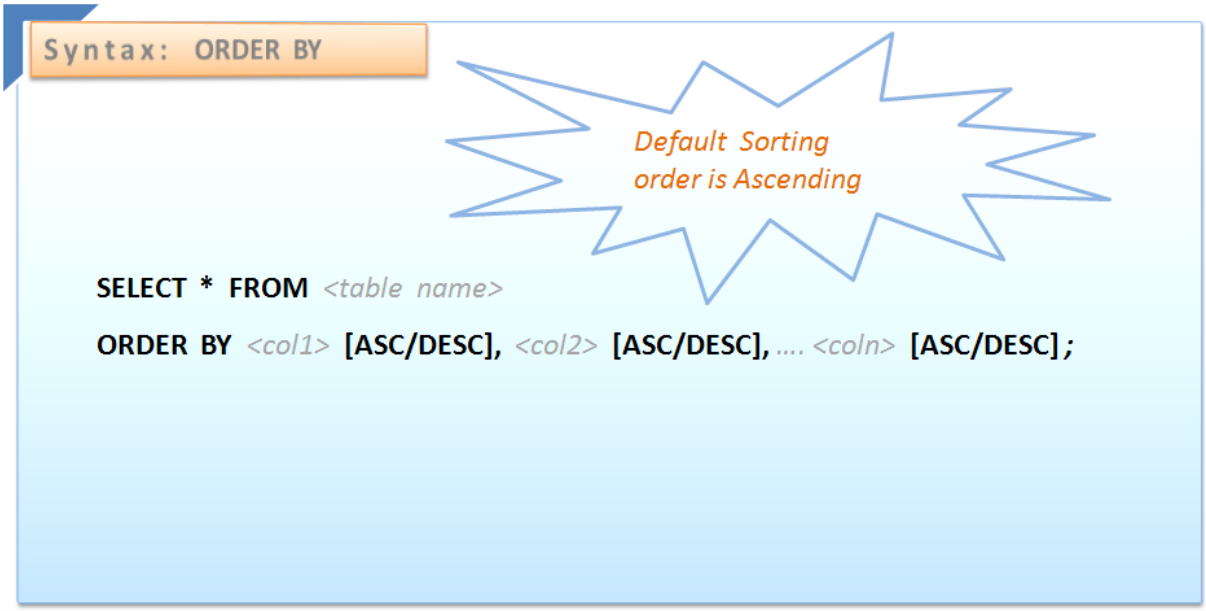
Example: - If we wanted to display whose salary is equal to 2000, below is the query.

```
SQL> SELECT *FROM emp WHERE sal =2000;
```

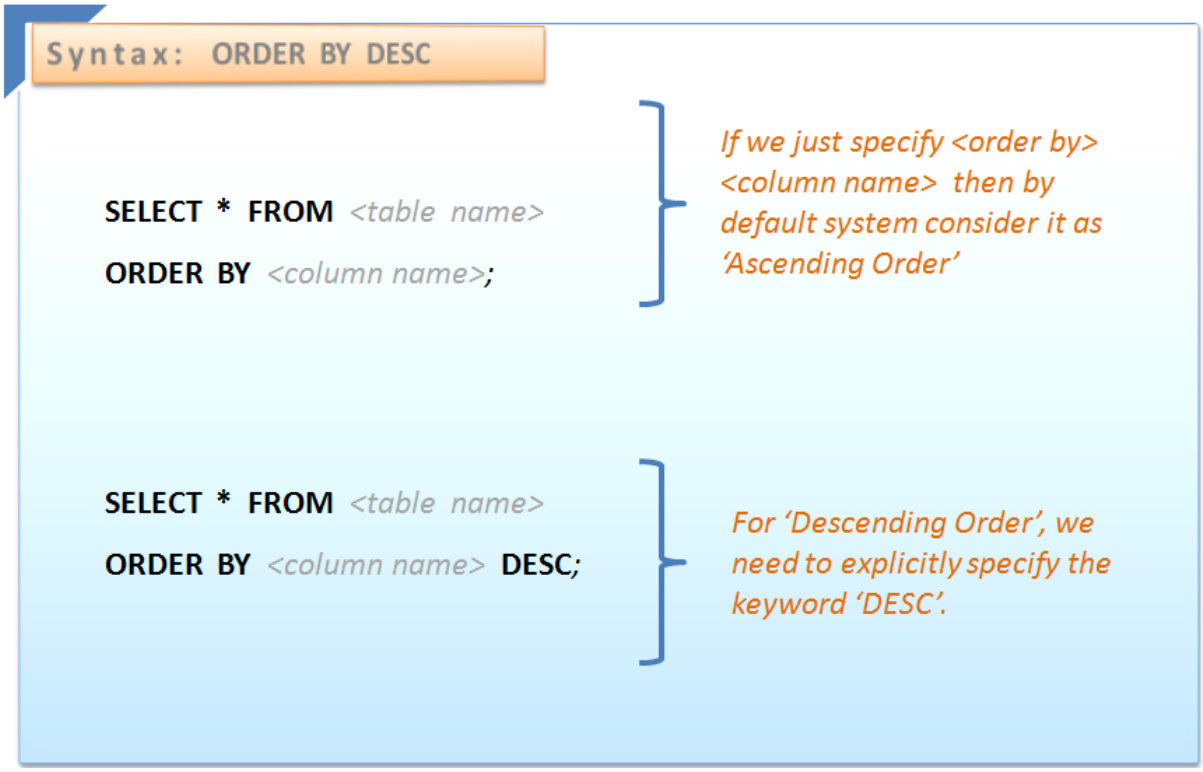
EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7369	SMITH	CLERK	7902	2/20/1981	2000		20

ORDER BY:

- 'ORDER BY' clause is used to arrange the select statement output in ASCENDING or DESCENDING order.
- 'ORDER BY' clause Supports with all types of data.



By default oracle will use ascending order. If we want to sort in descending order then specify "desc "



Example: Display employee records order by ename.

```
SQL> SELECT *
     FROM emp
     ORDER BY ename;
```

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7876	ADAMS	CLERK	7788	1/12/1983	1200		20
7499	ALLEN	SALESMAN	7698	2/20/1981	1800	300	30
7698	BLAKE	MANAGER	7839	5/1/1981	2950		30
7782	CLARK	MANAGER	7839	6/9/1981	2550		10
7902	FORD	ANALYST	7566	12/3/1981	3100		20
7900	JAMES	CLERK	7698	12/3/1981	1050		30
7566	JONES	MANAGER	7839	4/2/1981	3075		20
7839	KING	PRESIDENT		11/17/1981	5100		10
7654	MARTIN	SALESMAN	7698	9/28/1981	1350	1400	30
7934	MILLER	CLERK	7782	1/23/1982	1400		10
7788	SCOTT	ANALYST	7566	12/9/1982	3100		20
7369	SMITH	CLERK	7902	2/20/1981	2000		20
7844	TURNER	SALESMAN	7698	9/8/1981	1700	0	30
7521	WARD	SALESMAN	7698	2/22/1981	1350	500	30

Here we can see that 'ENAME' values resulted in 'ASCENDING ORDER'.

Example: Display employee records order by 'ename' in descending order. Here we need to explicitly specify the keyword 'DESC'.

```
SQL> SELECT *
     FROM EMP
     ORDER BY ename DESC;
```

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7521	WARD	SALESMAN	7698	2/22/1981	1350	500	30
7844	TURNER	SALESMAN	7698	9/8/1981	1700	0	30
7369	SMITH	CLERK	7902	2/20/1981	2000		20
7788	SCOTT	ANALYST	7566	12/9/1982	3100		20
7934	MILLER	CLERK	7782	1/23/1982	1400		10
7654	MARTIN	SALESMAN	7698	9/28/1981	1350	1400	30
7839	KING	PRESIDENT		11/17/1981	5100		10
7566	JONES	MANAGER	7839	4/2/1981	3075		20
7900	JAMES	CLERK	7698	12/3/1981	1050		30
7902	FORD	ANALYST	7566	12/3/1981	3100		20
7782	CLARK	MANAGER	7839	6/9/1981	2550		10
7698	BLAKE	MANAGER	7839	5/1/1981	2950		30
7499	ALLEN	SALESMAN	7698	2/20/1981	1800	300	30
7876	ADAMS	CLERK	7788	1/12/1983	1200		20

Ordering the result based on a numeric column.

```
SQL> SELECT *
   FROM emp
   ORDER BY sal;
```

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7900	JAMES	CLERK	7698	12/3/1981	1050		30
7876	ADAMS	CLERK	7788	1/12/1983	1200		20
7521	WARD	SALESMAN	7698	2/22/1981	1350	500	30
7654	MARTIN	SALESMAN	7698	9/28/1981	1350	1400	30
7934	MILLER	CLERK	7782	1/23/1982	1400		10
7844	TURNER	SALESMAN	7698	9/8/1981	1700	0	30
7499	ALLEN	SALESMAN	7698	2/20/1981	1800	300	30
7369	SMITH	CLERK	7902	2/20/1981	2000		20
7782	CLARK	MANAGER	7839	6/9/1981	2550		10
7698	BLAKE	MANAGER	7839	5/1/1981	2950		30
7566	JONES	MANAGER	7839	4/2/1981	3075		20
7788	SCOTT	ANALYST	7566	12/9/1982	3100		20
7902	FORD	ANALYST	7566	12/3/1981	3100		20
7839	KING	PRESIDENT		11/17/1981	5100		10

We can as well order the result using a column sequence.

```
SQL> SELECT empno,ename,job,sal
   FROM emp
   ORDER BY 2;
```

EMPNO	ENAME	JOB	SAL
7876	ADAMS	CLERK	1200
7499	ALLEN	SALESMAN	1800
7698	BLAKE	MANAGER	2950
7782	CLARK	MANAGER	2550
7902	FORD	ANALYST	3100
7900	JAMES	CLERK	1050
7566	JONES	MANAGER	3075
7839	KING	PRESIDENT	5100
7654	MARTIN	SALESMAN	1350
7934	MILLER	CLERK	1400
7788	SCOTT	ANALYST	3100
7369	SMITH	CLERK	2000
7844	TURNER	SALESMAN	1700
7521	WARD	SALESMAN	1350

Here 2 represents second column. So, data will be displayed with ename in ascending order.

GROUP BY:

- Group by clause is used to create groups of related information.
- Columns used in 'select' must be used with 'group by'; otherwise system does not recognize it as a group by expression.

Example:-

In the below example we are trying to display sum of salaries at each dept level.

When SQL statement is run, data fetched from the database is grouped first based on deptno and then sum ()

Function is applied to get the desired result.

```
SQL> SELECT deptno, SUM(sal)
  FROM emp
  GROUP BY deptno;
```

DEPTNO	SUM(SAL)
30	9400
20	10875
10	8750

In this example we are trying to group the records using two columns deptno and job

```
SQL> SELECT deptno, job, SUM(Sal)
  FROM emp
  GROUP BY deptno,job;
```

DEPTNO	JOB	SUM(SAL)
20	CLERK	1900
30	SALESMAN	5600
20	MANAGER	2975
30	CLERK	950
10	PRESIDENT	5000
30	MANAGER	2850
10	CLERK	1300
10	MANAGER	2450
20	ANALYST	6000

Suppose if we try to apply summary function on a column along displaying other columns and not using group by clause will result in error.

```
SQL> SELECT empno, ename, deptno, MAX (Sal) FROM emp;
```

Output:

ERROR at line 1:
ORA-00937: **NOT** single-**GROUP** groups **FUNCTION**

Even if we specify only one column in group by then system will raise an error

SQL> **SELECT** empno, ename, deptno, **MAX** (sal) **FROM** emp **GROUP BY** deptno;

Output:

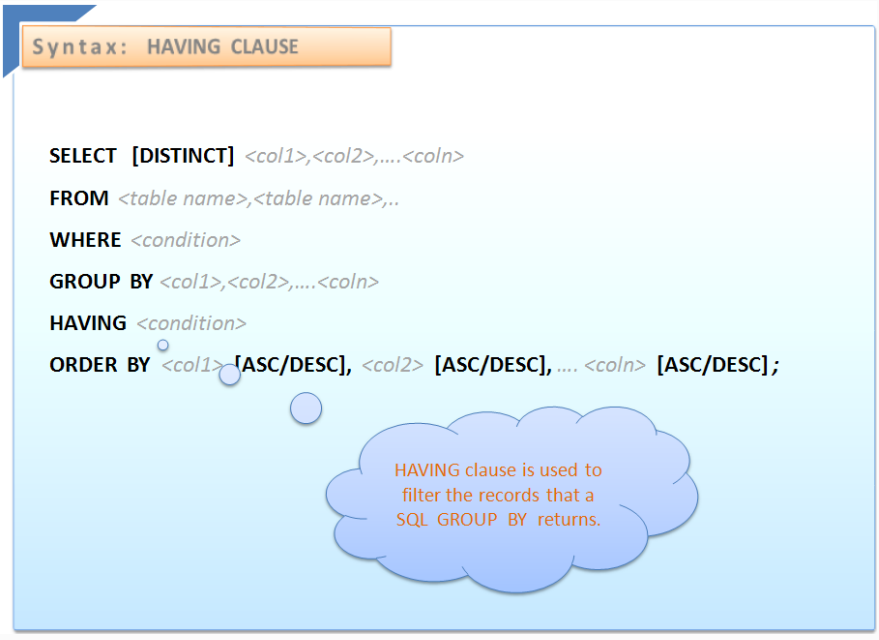
ERROR at line 1:
ORA-00979: **NOT** a **GROUP BY** expression

Rules:-

- The columns specified in the group by clause should be specified in select clause.
- If not in Group by then that column in select should be applies with summary functions.
- Summary columns and column aliases are not allowed in group by clause.

Having Clause:

- Having clause is used to restrict the records that are grouped using group by clause.
- Having clause is used to apply conditions on the summary results.
- Having clause is just like where clause but it can be used only with group by as we cannot use where clause in group by.



In the below example we are trying to fetch all deptno and job records whose aggregated salary of its

Employees is greater than 3000 (Sum(Sal) > 3000).

Having clause gets executed after the records are fetched and grouped.

```
SQL>SELECT deptno,job, SUM(sal) tsal
    FROM emp
    GROUP BY deptno,job
    HAVING SUM(sal) > 3000;
```

DEPTNO	JOB	TSAL
30	SALESMAN	5600
10	PRESIDENT	5000
20	ANALYST	6000

In this example, after applying the having clause we are trying to order the records by DEPTNO.

```
SQL> SELECT deptno,job,SUM(sal) tsal
    FROM emp
    GROUP BY deptno,job
    HAVING SUM(sal) > 3000
    ORDER BY deptno;
```

DEPTNO	JOB	TSAL
10	PRESIDENT	5000
20	ANALYST	6000
30	SALESMAN	5600

In the below example, '**where** condition' is applied first and after that **HAVING** clause is applied.

```
SQL> SELECT deptno, COUNT (*)
    FROM emp
    WHERE job='CLERK'
    GROUP BY deptno
    HAVING COUNT (*)>=2;
```

DEPTNO	COUNT(*)
20	2

ORDER OF SQL STATEMENT EXECUTION:

- Group the rows together based on group by clause.
- Calculate the group functions for each group.
- Choose and eliminate the groups based on the having clause.
- Order the groups based on the specified column.

Chapter 9: Operators

SQL Operators:

- Operators are used to express the conditions in Select statements.
- Operator manipulates individual data items and returns a result.
- If multiple operators are used in a condition then condition will be executed based on the operator precedence/priority.
- An operator manipulates individual data items and returns a result.
- The data items are called operands or arguments.
- Operators are represented by special characters or by keywords.
- Oracle SQL also supports set operators.

SQL Types of Operators:

The different types of Operators available in Oracle SQL are:-

- Relational operators
- Arithmetic operators
- Logical operators
- Special Operators
- Boolean Operators
- Concatenation Operator

We will discuss in detail about each and every operator in the next chapters.

Chapter 10: Relational Operators

Equals to (=) Operator:

Relational Operator Equals to:

= [Equalsto]

This operator is used for equality test.

Used to test the equality of two operands.

Example:-

Display the details of Employees whose salary is equal to 2000.

SQL> **SELECT** ***FROM** emp **WHERE** Sal=2000;

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7369	SMITH	CLERK	7902	2/20/1981	2000		20

Less than (<) Operator:

Relational Operator Less than:

This operator is used for less than test. Example a<b checks that operand 'a' is less than 'b' or not.

Example: Display the details of the employees whose salary is less than 3000.

SQL> **SELECT** * **FROM** emp **WHERE** Sal < 3000

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7369	SMITH	CLERK	7902	2/20/1981	200(		20
7499	ALLEN	SALESMAN	7698	2/20/1981	180(	300	30
7521	WARD	SALESMAN	7698	2/22/1981	135(	500	30
7654	MARTIN	SALESMAN	7698	9/28/1981	135(	1400	30
7698	BLAKE	MANAGER	7839	5/1/1981	295(		30
7782	CLARK	MANAGER	7839	6/9/1981	255(		10
7844	TURNER	SALESMAN	7698	9/8/1981	170(	0	30
7876	ADAMS	CLERK	7788	1/12/1983	120(		20
7900	JAMES	CLERK	7698	12/3/1981	105(		30
7934	MILLER	CLERK	7782	1/23/1982	140(		10

Here if you observe we got a result values whose salary is less than the operand 3000.

Greater than (>) Operator:

Relational Operator Greater than:

(Greater than Operator)
This operator is used for Greater than test. For example a>b checks the operand 'a' is greater than 'b' or not.

Example:

Display the details of Employees whose salary is greater than 3000

SQL> **SELECT** * **FROM** emp **WHERE** Sal > 3000;

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7566	JONES	MANAGER	7839	4/2/1981	3075		20
7788	SCOTT	ANALYST	7566	12/9/1982	3100		20
7839	KING	PRESIDENT		11/17/198:	5100		10
7902	FORD	ANALYST	7566	12/3/1981	3100		20

Less than or Equals to (<=) Operator:

Relational Operator Less than or Equals to:

Less than or Equals to Operator (<=)
This operator is used for Less than or Equal to test. For example a<=b, here checks whether operand 'a' is less than or equals to operand 'b'. If a<b then condition is true and if a=b then also condition is true but if a>b then condition is false.

Example:

Display the details of Employees whose salary is less than or equal to 3000.

SQL> **SELECT** * **FROM** EMP **WHERE** Sal <= 3000;

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7369	SMITH	CLERK	7902	2/20/1981	2000		20
7499	ALLEN	SALESMAN	7698	2/20/1981	1800	300	30
7521	WARD	SALESMAN	7698	2/22/1981	1350	500	30
7654	MARTIN	SALESMAN	7698	9/28/1981	1350	1400	30
7698	BLAKE	MANAGER	7839	5/1/1981	2950		30
7782	CLARK	MANAGER	7839	6/9/1981	2550		10
7844	TURNER	SALESMAN	7698	9/8/1981	1700	0	30
7876	ADAMS	CLERK	7788	1/12/1983	1200		20
7900	JAMES	CLERK	7698	12/3/1981	1050		30
7934	MILLER	CLERK	7782	1/23/1982	1400		10

Greater than or Equals to (>=) Operator:

Relational Operator Greater than or Equals to:

Greater than or Equals to (>=)

This operator is used to check the Greater than or equal test. For example a>=b checks the operand 'a' is greater than operand 'b' or operand 'a' is equals to the operand 'b'.

Example:

Display the details of Employees whose salary is greater than or equal to 3000.

```
SQL> SELECT * FROM emp WHERE Sal >= 3000;
```

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7566	JONES	MANAGER	7839	4/2/1981	3075		20
7788	SCOTT	ANALYST	7566	12/9/1982	3100		20
7839	KING	PRESIDENT		11/17/1981	5100		10
7902	FORD	ANALYST	7566	12/3/1981	3100		20

Not Equals to (! = / ^= / <>) Operator:

Relational Operator Not Equals to:

Not equals to (! = or ^= or <>)

This operator is used for inequality test.

Examples:

Display the details of employees whose salary is not equals to 2000.

```
SELECT * FROM emp WHERE Sal != 2000;
```

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7499	ALLEN	SALESMAN	7698	2/20/1981	1600	300	30
7521	WARD	SALESMAN	7698	2/22/1981	1250	500	30
7566	JONES	MANAGER	7839	4/2/1981	2975		20
7654	MARTIN	SALESMAN	7698	9/28/1981	1250	1400	30
7698	BLAKE	MANAGER	7839	5/1/1981	2850		30
7782	CLARK	MANAGER	7839	6/9/1981	2450		10
7788	SCOTT	ANALYST	7566	12/9/1982	3000		20
7839	KING	PRESIDENT		11/17/1981	5000		10
7844	TURNER	SALESMAN	7698	9/8/1981	1500		30
7876	ADAMS	CLERK	7788	1/12/1983	1100		20
7900	JAMES	CLERK	7698	12/3/1981	950		30
7902	FORD	ANALYST	7566	12/3/1981	3000		20
7934	MILLER	CLERK	7782	1/23/1982	1300		10

SELECT * **FROM** emp **WHERE** sal ^= 2000;

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7499	ALLEN	SALESMAN	7698	2/20/1981	1600	300	30
7521	WARD	SALESMAN	7698	2/22/1981	1250	500	30
7566	JONES	MANAGER	7839	4/2/1981	2975		20
7654	MARTIN	SALESMAN	7698	9/28/1981	1250	1400	30
7698	BLAKE	MANAGER	7839	5/1/1981	2850		30
7782	CLARK	MANAGER	7839	6/9/1981	2450		10
7788	SCOTT	ANALYST	7566	12/9/1982	3000		20
7839	KING	PRESIDENT		11/17/1981	5000		10
7844	TURNER	SALESMAN	7698	9/8/1981	1500		30
7876	ADAMS	CLERK	7788	1/12/1983	1100		20
7900	JAMES	CLERK	7698	12/3/1981	950		30
7902	FORD	ANALYST	7566	12/3/1981	3000		20
7934	MILLER	CLERK	7782	1/23/1982	1300		10

SELECT * **FROM** emp **WHERE** sal <> 2000;

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7499	ALLEN	SALESMAN	7698	2/20/1981	1600	300	30
7521	WARD	SALESMAN	7698	2/22/1981	1250	500	30
7566	JONES	MANAGER	7839	4/2/1981	2975		20
7654	MARTIN	SALESMAN	7698	9/28/1981	1250	1400	30
7698	BLAKE	MANAGER	7839	5/1/1981	2850		30
7782	CLARK	MANAGER	7839	6/9/1981	2450		10
7788	SCOTT	ANALYST	7566	12/9/1982	3000		20
7839	KING	PRESIDENT		11/17/1981	5000		10
7844	TURNER	SALESMAN	7698	9/8/1981	1500		30
7876	ADAMS	CLERK	7788	1/12/1983	1100		20
7900	JAMES	CLERK	7698	12/3/1981	950		30
7902	FORD	ANALYST	7566	12/3/1981	3000		20
7934	MILLER	CLERK	7782	1/23/1982	1300		10

SQL IN Operator:

IN:

- Returns true if value is available in given list of values
- Supports with all types of data (data types)

In the below example only employees whose empno is (7125, 7369, 7782) are fetched.

SQL> **SELECT** ***FROM** emp **WHERE** empno **IN** (7125, 7369, 7782);

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7369	SMITH	CLERK	7902	2/20/1981	2000		20
7782	CLARK	MANAGER	7839	6/9/1981	2550		10

Inside DML statements:

SQL> **UPDATE** emp **SET** Sal=sal+200 **WHERE** ename **IN** ('SMITH','ALLEN','WARD');

SQL> **DELETE FROM** emp **WHERE** hire date **IN** ('22-DEC-82',' 17-NOV-81');

SQL BETWEEN Operators:

BETWEEN

- Returns true if value specified is within the specified range.
- Supports with numbers and date values.
- **between** is an inclusive operator which includes range limits in output

Example: - in this example all employee records are fetched whose salary is between 2000 and 3000

SQL> **SELECT** ***FROM** emp **WHERE** Sal **BETWEEN** 2000 **AND** 3000;

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7369	SMITH	CLERK	7902	2/20/1981	2000		20
7698	BLAKE	MANAGER	7839	5/1/1981	2950		30
7782	CLARK	MANAGER	7839	6/9/1981	2550		10

Whenever lower bound value is larger than upper bound then it shows 'no rows selected'

Example:-

SQL> **SELECT** ***FROM** emp **WHERE** Sal **BETWEEN** 3000 **AND** 2000;

Output:

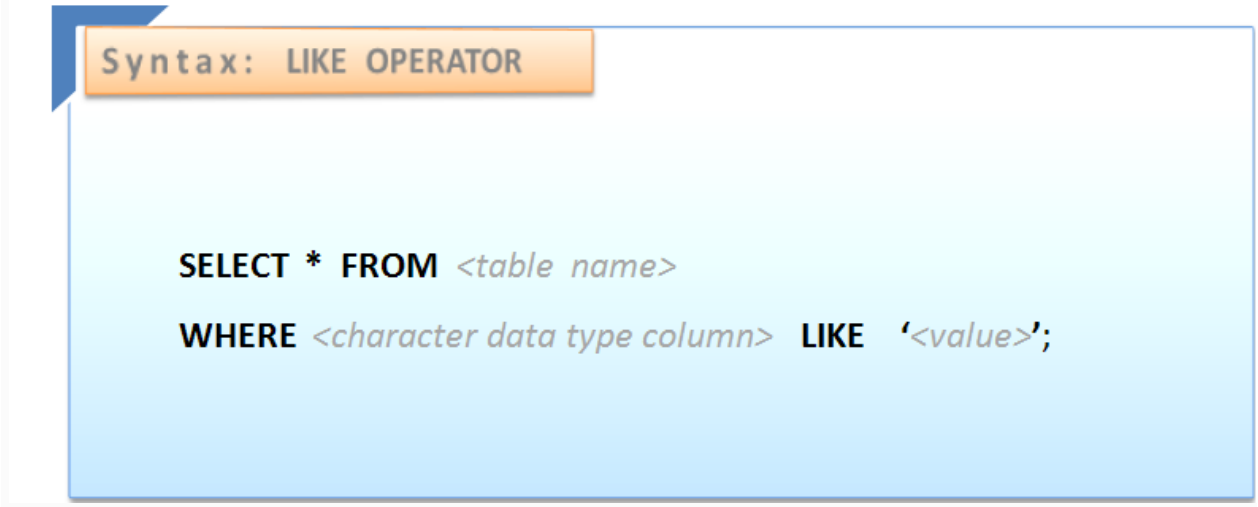
-- No rows selected

SQL LIKE Operator:

LIKE:-

- Used to search for pattern in a given input
- It is supported with character data only
- It uses two Meta characters

% (percentage) and **_ (underscore)** are two Meta characters.
% (percentage) represents "zero or more characters "in the given input
_ (underscore) represents "one character" in given input.



Example: - Display the employees whose name is starting with 'S' in EMP table.

SQL> **SELECT** * **FROM** emp **WHERE** ename **LIKE** 'S%';

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7369	SMITH	CLERK	7902	2/20/1981	2000		20
7788	SCOTT	ANALYST	7566	12/9/1982	3100		20

Display the employees whose name ends with 'S' in EMP table

SQL> **SELECT** * **FROM** emp **WHERE** ename **LIKE** '%S';

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7566	JONES	MANAGER	7839	4/2/1981	3075		20
7876	ADAMS	CLERK	7788	1/12/1983	1200		20
7900	JAMES	CLERK	7698	12/3/1981	1050		30

Display the employees whose names are having second letter as 'L' in EMP table

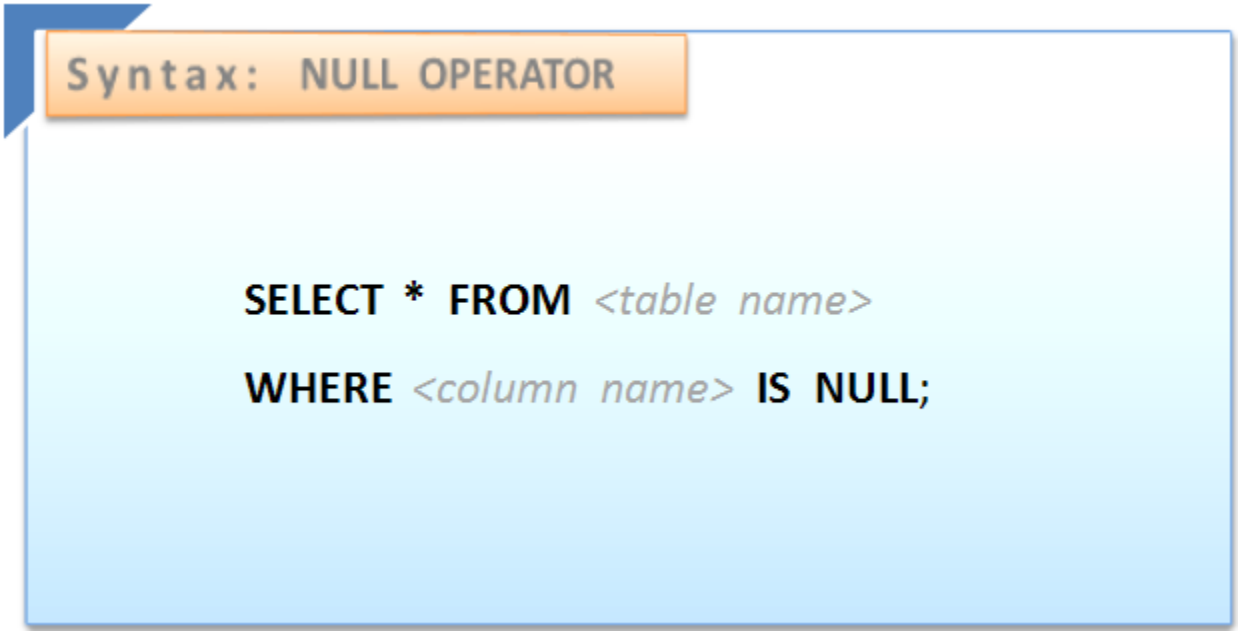
SQL> **SELECT** * **FROM** emp **WHERE** ename **LIKE** '_L%';

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7499	ALLEN	SALESMAN	7698	2/20/1981	1600	300	30
7698	BLAKE	MANAGER	7839	5/1/1981	2850		30
7782	CLARK	MANAGER	7839	6/9/1981	90445		10

SQL IS NULL Operator:

IS NULL:-

- Used to search for NULL values In the given input
- Supports with all types of data



Example:-

SQL> **SELECT** * **FROM** emp **WHERE** sal **IS NULL**;

Output:-

-- No rows selected;

AS there **IS** salary available **FOR** every employee.

Let's fetch all employees whose has commission enabled.

SQL> **SELECT** ***FROM** emp **WHERE** comm **IS NULL**;

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7369	SMITH	CLERK	7902	2/20/1981	2000		20
7566	JONES	MANAGER	7839	4/2/1981	3075		20
7698	BLAKE	MANAGER	7839	5/1/1981	2950		30
7782	CLARK	MANAGER	7839	6/9/1981	2550		10
7788	SCOTT	ANALYST	7566	12/9/1982	3100		20
7839	KING	PRESIDENT		11/17/1981	5100		10
7876	ADAMS	CLERK	7788	1/12/1983	1200		20
7900	JAMES	CLERK	7698	12/3/1981	1050		30
7902	FORD	ANALYST	7566	12/3/1981	3100		20
7934	MILLER	CLERK	7782	1/23/1982	1400		10

SQL Concatenation (||) Operator:

Character operators or Concatenation Operators are used in expressions to manipulate character strings.

Look at the below example it shows how to use the Character Operator.

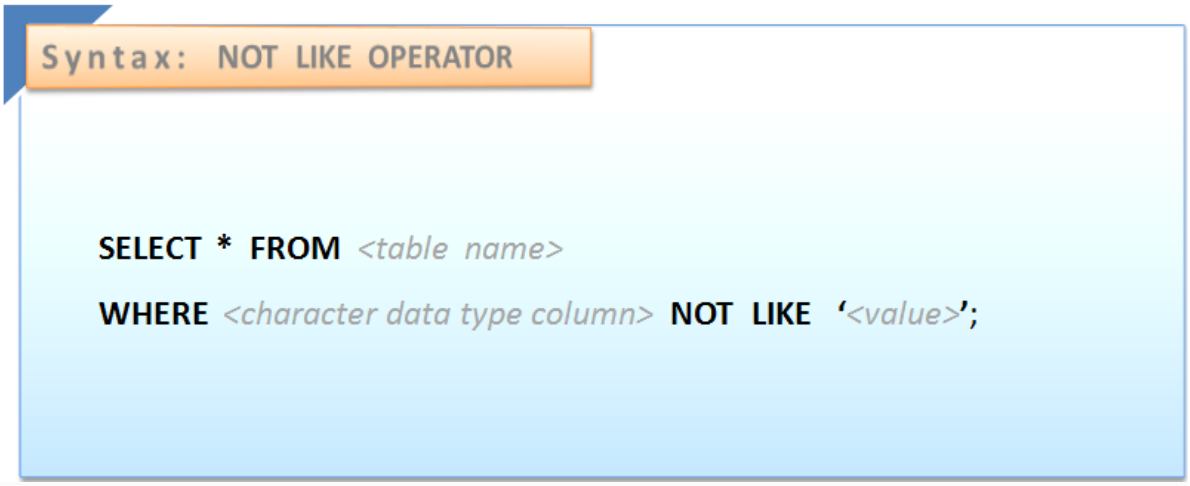
```
SQL>SELECT 'The Name of the employee is: ' || ENAME FROM EMP;
```

Output:

'THENAMEOFTHEEMPLOYEEIS:'||ENAME
The Name of the employee is: KING
The Name of the employee is: BLAKE
The Name of the employee is: CLARK
The Name of the employee is: ALLEN
The Name of the employee is: WARD
The Name of the employee is: MARTIN
The Name of the employee is: TURNER
The Name of the employee is: JAMES
The Name of the employee is: MILLER
The Name of the employee is: ANDREW
The Name of the employee is: WARN

Chapter 11: Relational Negation Operators

NOT LIKE Operator:



Display the employees whose name is not ends with 'S' in EMP table

SQL> **SELECT** ***FROM** emp **WHERE** ename **NOT LIKE** '%S';

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7369	SMITH	CLERK	7902	2/20/1981	2000		20
7499	ALLEN	SALESM	7698	2/20/1981	1800	300	30
7521	WARD	SALESM	7698	2/22/1981	1350	500	30
7654	MARTIN	SALESM	7698	9/28/1981	1350	1400	30
7698	BLAKE	MANAG	7839	5/1/1981	2950		30
7782	CLARK	MANAG	7839	6/9/1981	2550		10
7788	SCOTT	ANALYS	7566	12/9/1982	3100		20
7839	KING	PRESID		11/17/1981	5100		10
7844	TURNER	SALESM	7698	9/8/1981	1700	0	30
7902	FORD	ANALYS	7566	12/3/1981	3100		20
7934	MILLER	CLERK	7782	1/23/1982	1400		10

Display the employees whose names are not having second letter as 'L' in EMP table

SQL>**SELECT** ***FROM** emp **WHERE** ename **NOT LIKE** '_L%';

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7369	SMITH	CLERK	7902	2/20/1981	2000		20
7521	WARD	SALESMAN	7698	2/22/1981	1350	500	30
7566	JONES	MANAGER	7839	4/2/1981	3075		20
7654	MARTIN	SALESMAN	7698	9/28/1981	1350	1400	30
7788	SCOTT	ANALYST	7566	12/9/1982	3100		20
7839	KING	PRESIDENT		11/17/1981	5100		10
7844	TURNER	SALESMAN	7698	9/8/1981	1700	0	30
7876	ADAMS	CLERK	7788	1/12/1983	1200		20
7900	JAMES	CLERK	7698	12/3/1981	1050		30
7902	FORD	ANALYST	7566	12/3/1981	3100		20
7934	MILLER	CLERK	7782	1/23/1982	1400		10

Display the employees whose names are not start with ‘S’ in EMP table.

SQL>**SELECT** ***FROM** emp **WHERE** ename **NOT LIKE** ‘S%’;

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7499	ALLEN	SALESMAN	7698	2/20/1981	1800	300	30
7521	WARD	SALESMAN	7698	2/22/1981	1350	500	30
7566	JONES	MANAGER	7839	4/2/1981	3075		20
7654	MARTIN	SALESMAN	7698	9/28/1981	1350	1400	30
7698	BLAKE	MANAGER	7839	5/1/1981	2950		30
7782	CLARK	MANAGER	7839	6/9/1981	2550		10
7839	KING	PRESIDENT		11/17/1981	5100		10
7844	TURNER	SALESMAN	7698	9/8/1981	1700	0	30
7876	ADAMS	CLERK	7788	1/12/1983	1200		20
7900	JAMES	CLERK	7698	12/3/1981	1050		30
7902	FORD	ANALYST	7566	12/3/1981	3100		20
7934	MILLER	CLERK	7782	1/23/1982	1400		10

Display the employees whose names are second letter start with ‘R’ from ending.

SQL>**SELECT** ***FROM** emp **WHERE** ename **LIKE** ‘%R_’;

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7521	WARD	SALESMAN	7698	2/22/1981	1350	500	30
7782	CLARK	MANAGER	7839	6/9/1981	2550		10
7902	FORD	ANALYST	7566	12/3/1981	3100		20

Display the names in EMP table whose names having ‘LL’.

SQL> **SELECT** ***FROM** emp **WHERE** ename **LIKE** ‘%LL%’;

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7499	ALLEN	SALESMAN	7698	2/20/1981	1800	300	30
7934	MILLER	CLERK	7782	1/23/1982	1400		10

Not Equals to (! = / ^= / <>) Operator:

Relational Operator Not Equals to

Not equals to (! = or ^= or <>)
This operator is used for inequality test.

Examples:

Display the details of employees whose salary is not equals to 2000.

```
SELECT * FROM emp WHERE sal != 2000;
```

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7499	ALLEN	SALESMAN	7698	2/20/1981	1600	300	30
7521	WARD	SALESMAN	7698	2/22/1981	1250	500	30
7566	JONES	MANAGER	7839	4/2/1981	2975		20
7654	MARTIN	SALESMAN	7698	9/28/1981	1250	1400	30
7698	BLAKE	MANAGER	7839	5/1/1981	2850		30
7782	CLARK	MANAGER	7839	6/9/1981	2450		10
7788	SCOTT	ANALYST	7566	12/9/1982	3000		20
7839	KING	PRESIDENT		11/17/1981	5000		10
7844	TURNER	SALESMAN	7698	9/8/1981	1500		30
7876	ADAMS	CLERK	7788	1/12/1983	1100		20
7900	JAMES	CLERK	7698	12/3/1981	950		30
7902	FORD	ANALYST	7566	12/3/1981	3000		20
7934	MILLER	CLERK	7782	1/23/1982	1300		10

```
SELECT * FROM emp WHERE sal ^= 2000;
```

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7499	ALLEN	SALESMAN	7698	2/20/1981	1600	300	30
7521	WARD	SALESMAN	7698	2/22/1981	1250	500	30
7566	JONES	MANAGER	7839	4/2/1981	2975		20
7654	MARTIN	SALESMAN	7698	9/28/1981	1250	1400	30
7698	BLAKE	MANAGER	7839	5/1/1981	2850		30
7782	CLARK	MANAGER	7839	6/9/1981	2450		10
7788	SCOTT	ANALYST	7566	12/9/1982	3000		20
7839	KING	PRESIDENT		11/17/1981	5000		10
7844	TURNER	SALESMAN	7698	9/8/1981	1500		30
7876	ADAMS	CLERK	7788	1/12/1983	1100		20
7900	JAMES	CLERK	7698	12/3/1981	950		30
7902	FORD	ANALYST	7566	12/3/1981	3000		20
7934	MILLER	CLERK	7782	1/23/1982	1300		10

SELECT * **FROM** emp **WHERE** sal <> 2000;

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7499	ALLEN	SALESMAN	7698	2/20/1981	1600	300	30
7521	WARD	SALESMAN	7698	2/22/1981	1250	500	30
7566	JONES	MANAGER	7839	4/2/1981	2975		20
7654	MARTIN	SALESMAN	7698	9/28/1981	1250	1400	30
7698	BLAKE	MANAGER	7839	5/1/1981	2850		30
7782	CLARK	MANAGER	7839	6/9/1981	2450		10
7788	SCOTT	ANALYST	7566	12/9/1982	3000		20
7839	KING	PRESIDENT		11/17/1981	5000		10
7844	TURNER	SALESMAN	7698	9/8/1981	1500		30
7876	ADAMS	CLERK	7788	1/12/1983	1100		20
7900	JAMES	CLERK	7698	12/3/1981	950		30
7902	FORD	ANALYST	7566	12/3/1981	3000		20
7934	MILLER	CLERK	7782	1/23/1982	1300		10

NOT IN Operator:

NOT IN:

'Not in' operator is quite opposite to 'IN' clause.

SQL> **SELECT** ***FROM** emp **WHERE** empno **NOT IN** (7125, 7369, 7782);

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7499	ALLEN	SALESMAN	7698	2/20/1981	1800	300	30
7521	WARD	SALESMAN	7698	2/22/1981	1350	500	30
7566	JONES	MANAGER	7839	4/2/1981	3075		20
7654	MARTIN	SALESMAN	7698	9/28/1981	1350	1400	30
7698	BLAKE	MANAGER	7839	5/1/1981	2950		30
7788	SCOTT	ANALYST	7566	12/9/1982	3100		20
7839	KING	PRESIDENT		11/17/198	5100		10
7844	TURNEF	SALESMAN	7698	9/8/1981	1700	0	30
7876	ADAMS	CLERK	7788	1/12/1983	1200		20
7900	JAMES	CLERK	7698	12/3/1981	1050		30
7902	FORD	ANALYST	7566	12/3/1981	3100		20
7934	MILLER	CLERK	7782	1/23/1982	1400		10

Inside DML statements:

SQL> **UPDATE** emp **SET** sal=sal+200 **WHERE** ename **NOT IN** ('SMITH','ALLEN','WARD');

SQL> **DELETE FROM** emp **WHERE** hiredate **NOT IN** ('22-DEC-82',' 17-NOV-81');

NOT BETWEEN Operator:

NOT BETWEEN

- Returns true if value specified is not within the specified range.
- Supports with numbers and date values.
- Not between is an exclusive operator which eliminates range limits from Output.

Example:-

SQL> **SELECT** ***FROM** emp **WHERE** Sal **NOT BETWEEN** 2000 **AND** 3000;

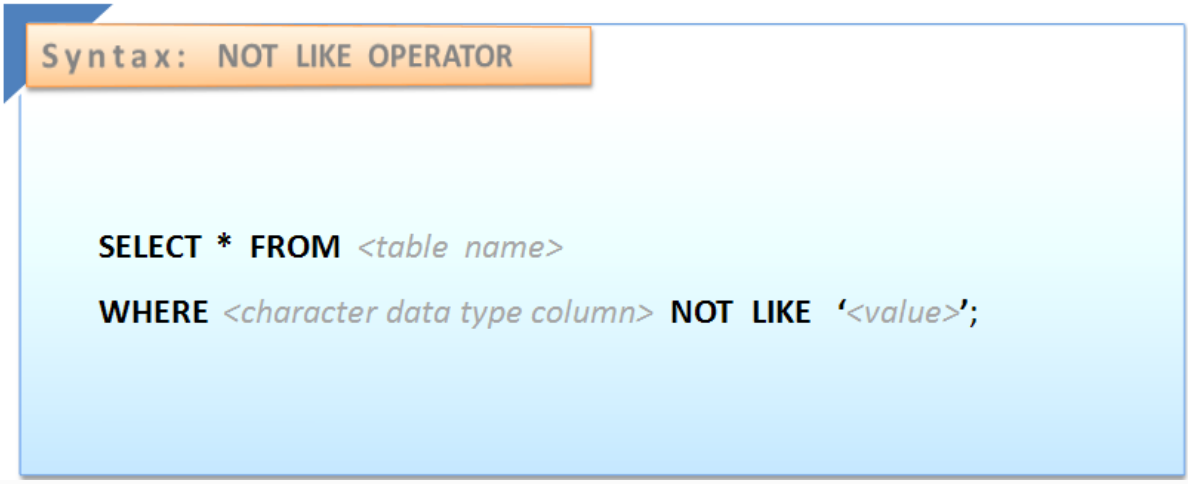
EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7499	ALLEN	SALESMAN	7698	2/20/1981	1800	300	30
7521	WARD	SALESMAN	7698	2/22/1981	1350	500	30
7566	JONES	MANAGER	7839	4/2/1981	3075		20
7654	MARTIN	SALESMAN	7698	9/28/1981	1350	1400	30
7788	SCOTT	ANALYST	7566	12/9/1982	3100		20
7839	KING	PRESIDENT		11/17/198:	5100		10
7844	TURNEF	SALESMAN	7698	9/8/1981	1700	0	30
7876	ADAMS	CLERK	7788	1/12/1983	1200		20
7900	JAMES	CLERK	7698	12/3/1981	1050		30
7902	FORD	ANALYST	7566	12/3/1981	3100		20
7934	MILLER	CLERK	7782	1/23/1982	1400		10

Lower bound – 'value 'must be lower when compare to 'upper bound 'value
Upper bound- 'value' must be higher when compare to 'lower bound 'value

IS NOT NULL Operator:

IS NOT NULL:-
-Use to search for NOT NULL values in the given input

-Supports with all types of data



Let's fetch all employees whose commission is NOT NULL.

NOT NULL Example:-

SQL> **SELECT** *__FROM__ emp **WHERE** comm **IS NOT NULL**;

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7499	ALLEN	SALESMAN	7698	2/20/1981	1800	300	30
7521	WARD	SALESMAN	7698	2/22/1981	1350	500	30
7654	MARTIN	SALESMAN	7698	9/28/1981	1350	1400	30
7844	TURNER	SALESMAN	7698	9/8/1981	1700	0	30

Original existing data in the ‘emp’ table is

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7369	SMITH	CLERK	7902	2/20/1981	2000		20
7499	ALLEN	SALESMAN	7698	2/20/1981	1800	300	30
7521	WARD	SALESMAN	7698	2/22/1981	1350	500	30
7566	JONES	MANAGER	7839	4/2/1981	3075		20
7654	MARTIN	SALESMAN	7698	9/28/1981	1350	1400	30
7698	BLAKE	MANAGER	7839	5/1/1981	2950		30
7782	CLARK	MANAGER	7839	6/9/1981	2550		10
7788	SCOTT	ANALYST	7566	12/9/1982	3100		20
7839	KING	PRESIDENT		11/17/1981	5100		10
7844	TURNER	SALESMAN	7698	9/8/1981	1700	0	30
7876	ADAMS	CLERK	7788	1/12/1983	1200		20
7900	JAMES	CLERK	7698	12/3/1981	1050		30
7902	FORD	ANALYST	7566	12/3/1981	3100		20
7934	MILLER	CLERK	7782	1/23/1982	1400		10

Chapter 12: Logical Operators

Logical AND Operator:

Logical Operator AND
Returns 'True' if both component conditions are true. Returns 'False' if any one component condition or both Component conditions are False.

Example:-

Display the details of Employees whose salary is Greater than 1000 AND also whose salary is less than 2000.

```
SQL> SELECT *FROM emp WHERE Sal > 1000 AND Sal <2000;
```

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7499	ALLEN	SALESMAN	7698	2/20/1981	1800	300	30
7521	WARD	SALESMAN	7698	2/22/1981	1350	500	30
7654	MARTIN	SALESMAN	7698	9/28/1981	1350	1400	30
7844	TURNER	SALESMAN	7698	9/8/1981	1700	0	30

Logical OR Operator:

Returns 'True' if either component conditions become TRUE. Returns 'False' if both the component conditions becomes 'False'.

Example:

Display the details of Employees whose salary is Greater than 1000 OR also whose salary is less than 2000.

```
SQL> SELECT *FROM emp WHERE Sal> 1000 OR Sal < 2000;
```

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7369	SMITH	CLERK	7902	2/20/1981	2000		20
7499	ALLEN	SALESMAN	7698	2/20/1981	1800	300	30
7521	WARD	SALESMAN	7698	2/22/1981	1350	500	30
7566	JONES	MANAGER	7839	4/2/1981	3075		20
7654	MARTIN	SALESMAN	7698	9/28/1981	1350	1400	30
7788	SCOTT	ANALYST	7566	12/9/1982	3100		20
7844	TURNER	SALESMAN	7698	9/8/1981	1700	0	30
7876	ADAMS	CLERK	7788	1/12/1983	1200		20
7900	JAMES	CLERK	7698	12/3/1981	1050		30
7902	FORD	ANALYST	7566	12/3/1981	3100		20
7934	MILLER	CLERK	7782	1/23/1982	1400		10

Logical NOT Operator:

The NOT operator returns 'True' if the condition is False and returns 'False' if the following condition is True.

Example:

Display the details of employees whose salary are Greater than or Equals to 3000.

SQL> **SELECT** * **FROM** emp **WHERE** Sal < 3000;

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7788	SCOTT	ANALYST	7566	12/9/1982	3000		20
7839	KING	PRESIDENT		11/17/1981	5000		10
7902	FORD	ANALYST	7566	12/3/1981	3000		20

Chapter 13: Arithmetic Operators

Multiplication (*) Operator:

Arithmetic Operator Multiplication (*): Used to perform multiplication.

Example:-

Display the details of the employees Incrementing their salary two times.

```
SQL> SELECT Sal * 2 FROM emp;
```

SAL*2
3200
2500
5950
2500
5700
4900
6000
10000
3000
2200
1900
6000
2600

Division (/) Operator:

Used to perform division test. Division will display only the Quotient value not the remainder value. Example 6/2 gives 3 because 2 divide 6 by 3 times.

Example:-

Display half of the salary of employees.

```
SQL> SELECT sal, sal/2 FROM emp;
```

SAL	SAL/2
1600	800
1250	625
2975	1487.5
1250	625
2850	1425
2450	1225
3000	1500
5000	2500
1500	750
1100	550
950	475
3000	1500
1300	650

Addition (+) Operator:

Arithmetic Operators – Used to perform any Arithmetic Operations like Addition, Subtraction, Multiplication and Divided by.
These operators have the highest precedence.

Arithmetic Operator Addition (+)

Example:-

Display salary of employees with 2000 increment in their salary.

```
SQL> SELECT e.*, e.sal + 2000 "Incremented salary" FROM emp e;
```

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO	Incremented salary
7369	SMITH	CLERK	7902	12/17/1980	800		20	2800
7499	ALLEN	SALESMAN	7698	2/20/1981	1600	300	30	3600
7521	WARD	SALESMAN	7698	2/22/1981	1250	500	30	3250
7566	JONES	MANAGER	7839	4/2/1981	2975		20	4975
7654	MARTIN	SALESMAN	7698	9/28/1981	1250	1400	30	3250
7698	BLAKE	MANAGER	7839	5/1/1981	2850		30	4850
7782	CLARK	MANAGER	7839	6/9/1981	2450		10	4450
7788	SCOTT	ANALYST	7566	12/9/1982	3000		20	5000
7839	KING	PRESIDENT		11/17/1981	5000		10	7000
7844	TURNER	SALESMAN	7698	9/8/1981	1500		30	3500
7876	ADAMS	CLERK	7788	1/12/1983	1100		20	3100
7900	JAMES	CLERK	7698	12/3/1981	950		30	2950
7902	FORD	ANALYST	7566	12/3/1981	3000		20	5000
7934	MILLER	CLERK	7782	1/23/1982	1300		10	3300

Subtraction (-) Operator:

Arithmetic Operator Subtraction (-), used to perform subtraction between two numbers and dates.

Example:

Display the details of employees decreasing their salary by 200.

```
SQL> SELECT e.*, e.sal - 200 FROM emp e;
```

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO	E.SAL-200
7499	ALLEN	SALESMAN	7698	2/20/1981	1600	300	30	1400
7521	WARD	SALESMAN	7698	2/22/1981	1250	500	30	1050
7566	JONES	MANAGER	7839	4/2/1981	2975		20	2775
7654	MARTIN	SALESMAN	7698	9/28/1981	1250	1400	30	1050
7698	BLAKE	MANAGER	7839	5/1/1981	2850		30	2650
7782	CLARK	MANAGER	7839	6/9/1981	2450		10	2250
7788	SCOTT	ANALYST	7566	12/9/1982	3000		20	2800
7839	KING	PRESIDENT		11/17/1981	5000		10	4800
7844	TURNER	SALESMAN	7698	9/8/1981	1500		30	1300
7876	ADAMS	CLERK	7788	1/12/1983	1100		20	900
7900	JAMES	CLERK	7698	12/3/1981	950		30	750
7902	FORD	ANALYST	7566	12/3/1981	3000		20	2800
7934	MILLER	CLERK	7782	1/23/1982	1300		10	1100

Chapter 14: Functions
String Functions:

Below is some of the string functions used to manipulate character information in strings.

- **UPPER**
- **LOWER**
- **INITCAP**
- **LENGTH**
- **RPAD**
- **LPAD**
- **LTRIM**
- **RTRIM**
- **TRIM**
- **TRANSLATE**
- **REPLACE**
- **SOUNDEX**
- **ASCII**
- **CHR**
- **SUBSTR**
- **INSTR**
- **DECODE**
- **GREATEST**
- **LEAST**
- **COALESCE**

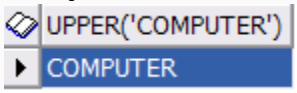
Upper

This function will convert the given string into uppercase.

Syntax: UPPER (*string*)

```
SQL> SELECT UPPER ('computer') FROM DUAL;
```

Output:-




```
SQL>SELECT UPPER (ename) FROM emp;
```

UPPER(ENAME)
SMITH
ALLEN
WARD
JONES
MARTIN
BLAKE
CLARK
SCOTT
KING
TURNER
ADAMS
JAMES
FORD
MILLER

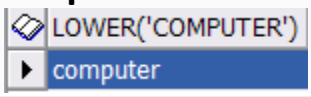
LOWER

This function will convert the given string into lowercase.

Syntax: LOWER (*string*)

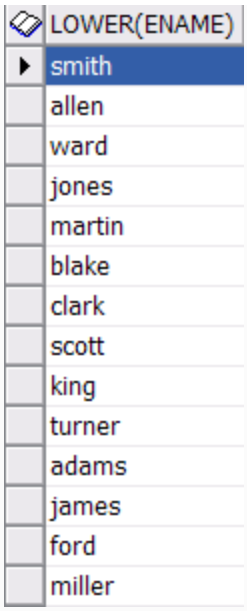
```
SQL> SELECT LOWER ('COMPUTER') FROM DUAL;
```

Output:-



LOWER('COMPUTER')
computer

```
SQL>SELECT LOWER (ename) FROM emp;
```



LOWER(ENAME)
smith
allen
ward
jones
martin
blake
clark
scott
king
turner
adams
james
ford
miller

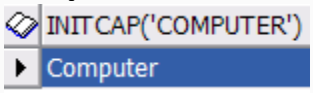
INITCAP

This function will capitalize the initial letter of the string.

Syntax: INITCAP (*string*)

```
SQL> SELECT INITCAP ('computer') FROM DUAL;
```

Output:-



INITCAP('COMPUTER')
Computer

```
SQL>SELECT INITCAP (ename) FROM emp;
```

INITCAP(ENAME)
Smith
Allen
Ward
Jones
Martin
Blake
Clark
Scott
King
Turner
Adams
James
Ford
Miller

LENGTH

This function will give length of the string.

Syntax:

LENGTH (*string*)

```
SQL> SELECT LENGTH ('computer') FROM DUAL;
```

Output:-

LENGTH('COMPUTER')
8

```
SQL>SELECT empno, ename, LENGTH (ename) FROM emp;
```

Output:-

EMPNO	ENAME	LENGTH(ENAME)
7369	SMITH	5
7499	ALLEN	5
7521	WARD	4
7566	JONES	5
7654	MARTIN	6
7698	BLAKE	5
7782	CLARK	5
7788	SCOTT	5
7839	KING	4
7844	TURNER	6
7876	ADAMS	5
7900	JAMES	5
7902	FORD	4
7934	MILLER	6

RPAD

This function will allows us to pad the right side of a column with any set of characters.

Syntax: RPAD (*string, length, [padding char]*)

```
SQL> SELECT RPAD ('computer', 15,'*'), RPAD ('computer',15,'*#') FROM DUAL;
```

Output:-

RPAD('COMPUTER',15,'*')	RPAD('COMPUTER',15,'*#')
computer*******	computer*#*#*#*

– Default padding character was blank space

LPAD

This function will allows you to pad the left side of a column with any set of characters.

Syntax: LPAD (*string, length, [padding char]*)

```
SQL> SELECT LPAD ('computer', 15,'*'), LPAD ('computer', 15,'*#') FROM DUAL;
```

Output:-

LPAD('COMPUTER',15,'*')	LPAD('COMPUTER',15,'*#')
*******computer	*#*#*#*computer

— Default padding character was blank space.

LTRIM

This function will trim off unwanted characters FROM the left end of string.

Syntax: LTRIM (*string, [unwanted chars]*)

SQL> **SELECT** LTRIM ('computer',' co'), LTRIM ('computer',' COM') **FROM** DUAL;

Output:-

LTRIM('COMPUTER','CO')	LTRIM('COMPUTER','COM')
mputer	computer

SQL> **SELECT** LTRIM ('computer','puter'), LTRIM ('computer','omputer') **FROM** DUAL;

Output:-

LTRIM('COMPUTER','PUTER')	LTRIM('COMPUTER','OMPUTER')
computer	computer

–If you haven't specified any unwanted characters it will display entire string.

RTRIM

This function will trim off unwanted characters FROM the right end of string.

Syntax: RTRIM (*string, [unwanted chars]*)

SQL> **SELECT** RTRIM ('computer','er'), RTRIM ('computer','er') **FROM** DUAL;

Output:-

RTRIM('COMPUTER','ER')	RTRIM('COMPUTER','ER')_1
comput	comput

SQL> **SELECT** RTRIM ('computer','omputer'), RTRIM ('computer',' compute') **FROM** DUAL;

Output :-

RTRIM('COMPUTER','OMPUTER')	RTRIM('COMPUTER','COMPUTE')
c	computer

–If you haven't specify any unwanted characters it will display entire string

TRIM

This function will trim off unwanted characters FROM the both sides of string.

Syntax: TRIM
(*unwanted chars* FROM *string*)

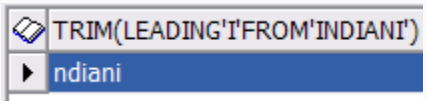
SQL> **SELECT TRIM** ('i' **FROM** 'Indiani') **FROM** DUAL;

Output:-

TRIM('I'FROM'INDIANI')
ndian

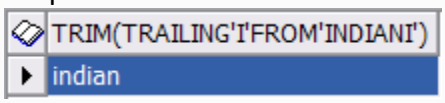
SQL> **SELECT TRIM** (**LEADING**'i' **FROM** 'indiani') **FROM** DUAL; -- *this will work as LTRIM*

Output:-



SQL> **SELECT TRIM** (**TRAILING**'i' **FROM** 'indiani') **FROM** DUAL; *-- this will work as RTRIM*

Output:-



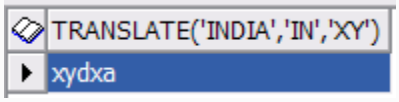
TRANSLATE

This function will replace the set of characters, character by character.

Syntax: TRANSLATE (*string, old chars, new chars*)

SQL> **SELECT** TRANSLATE ('india','in','xy') **FROM** DUAL;

Output:-



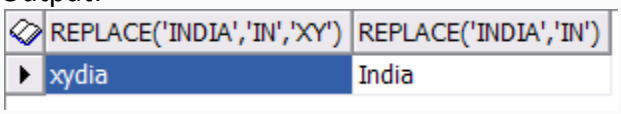
REPLACE

This function will replace the set of characters, string by string.

Syntax: REPLACE (*string, old chars, [new chars]*)

SQL> **SELECT REPLACE** ('india','in','xy'), **REPLACE** ('India',' in') **FROM** DUAL;

Output:-



SOUNDEX

This function returns a phonetic representation (the way it sounds) of a string.

Syntax: SOUNDEX (*string*)

The Soundex algorithm is as follows:

- The Soundex return value will always begin with the first letter of *string1*.
- The soundex function uses only the first 5 consonants to determine the NUMERIC portion of the return value, except if the first letter of *string1* is a vowel.
- The soundex function is **not** case-sensitive. What this means is that both uppercase and lowercase characters will generate the same soundex return value.

SQL> **SELECT** * **FROM** emp **WHERE** SOUNDEX (ename) = SOUNDEX ('SMIT');

Output:-

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7369	SMITH	CLERK	7902	2/20/1981	2400		20

```
SQL> SELECT 'how' || ' are' || ' you' FROM DUAL;
```

Output:-

'HOW'||'ARE'||'YOU'
how are you

ASCII

This function will return the decimal representation in the database character set of the first

character of the string.

Syntax: ASCII (*string*)

```
SQL> SELECT ASCII ('a'), ASCII ('apple') FROM DUAL;
```

Output:-

ASCII('A')	ASCII('APPLE')
97	97

CHR

This function will return the character having the binary equivalent of the input in either the

database character set or the national character set.

Syntax: CHR (*number*)

```
SQL> SELECT CHR (97) FROM DUAL;
```

Output:-

CHR(97)
a

SUBSTRING

This function is used to extract the particular portion of the string

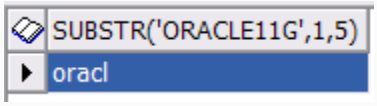
Syntax:-

```
SQL>SELECT SUBSTR (<'input string'>, <<span STYLE="font-size: 11pt;"> starting position</span> >, <<span STY
;
```

Example:-

```
SQL>SELECT SUBSTR ('oracle11g', 1, 5) FROM DUAL;
```

Output:-



SUBSTR('ORACLE11G',1,5)
oracl

Display the employee's names whose name start with same letter and end with same letter

<span **STYLE**="font-size: 11pt;">SQL>**SELECT** *__FROM__ emp **WHERE** SUBSTR <span **STYLE**="font-size: 11p

--no rows selected

INSTRING

This function is used to extract the position of the character in given string.

Syntax:-

SQL>**SELECT** INSTR(<**INPUT TYPE**="text" />, < **CHARACTER TO** search>, < Staring **POSITION** >, < occurrence **NUM**
FROM

;

Example:-

SQL> **SELECT** INSTR ('ANAND','A',1,2) **FROM** DUAL;

Output:-

INSTR ('ANAND','A',1,2)

3

DECODE

- Decode is used to check for multiple conditions while manipulating or retrieving the data.
- It implements "IF" construct logic
- If the number of parameters are odd and different then decode will display nothing.
- If the number of parameters are even and different then decode will display last value.
- If all the parameters are null then decode will display nothing.
- If all the parameters are zeros then decode will display zero.

Syntax: DECODE (**VALUE**, if1, then1, if2, then2… else);

Example:-

SQL> **SELECT** sal, DECODE (sal, 500,'Low', 5000,'High','Medium') **FROM** emp;

Output:-

SAL	RANGE
2400	Medium
2160	Medium
1620	Medium
3075	Medium
1620	Medium
2950	Medium
2550	Medium
3100	Medium
5100	Medium
2040	Medium
1440	Medium
1260	Medium
3100	Medium
1680	Medium

SQL> **SELECT** ename, sal, deptno, DECODE (deptno, 10, sal*0.15, 20, sal*.25, 30, sal*.35, Sal*.45) bonus **FROM**

Output:-

ENAME	SAL	DEPTNO	BONUS
SMITH	2400	20	600
ALLEN	2160	30	756
WARD	1620	30	567
JONES	3075	20	768.75
MARTIN	1620	30	567
BLAKE	2950	30	1032.5
CLARK	2550	10	382.5
SCOTT	3100	20	775
KING	5100	10	765
TURNER	2040	30	714
ADAMS	1440	20	360
JAMES	1260	30	441
FORD	3100	20	775
MILLER	1680	10	252

GREATEST

This will give the greatest string.

Syntax: GREATEST (*strng1, string2, string3 … stringn*)

SQL> **SELECT** GREATEST ('a', 'b', 'c'), GREATEST ('satish','srinu','saketh') **FROM** DUAL;

Output:-

GREATEST('A','B','C')	GREATEST('SATISH','SRINU','SAKETH')
c	srinu

- If all the parameters are nulls then it will display nothing.
- If any of the parameters is null it will display nothing.

LEAST
This function will give the least string.

Syntax: LEAST (*strng1, string2, string3 … stringn*)

```
SQL> SELECT LEAST ('a', 'b', 'c'), LEAST ('satish','srinu','saketh') FROM DUAL;
```

Output:-

LEAST('A','B','C')	LEAST('SATISH','SRINU','SAKETH')
a	saketh

- If all the parameters are nulls then it will display nothing.
- If any of the parameters is null it will display nothing.

COALESCE
This function will give the first not-null string.

Syntax: COALESCE (*strng1, string2, string3 … stringn*)

```
SQL> SELECT COALESCE ('a','b','c'), COALESCE (NULL,'a',NULL,'b') FROM DUAL;
```

Output:-

COALESCE('A','B','C')	COALESCE(NULL,'A',NULL,'B')
a	a

Numeric Functions:

Numeric functions accept numeric input and return numeric values.

NUMERIC FUNCTIONS:-

- **ABS**
- **SIGN**
- **SQRT**
- **MOD**
- **NVL**
- **POWER**
- **EXP**
- **LN**
- **LOG**

- **CEIL**
- **FLOOR**
- **ROUND**
- **TRUNC**
- **BITAND**
- **GREATEST**
- **LEAST**
- **COALESCE**

ABS

- Absolute value is the measure of the magnitude of value.
- Absolute value is always a positive number.

Syntax: – ABS (value)

Example:-

SQL> **SELECT** ABS (5), ABS (-5), ABS (0), ABS (**NULL**) **FROM** DUAL;

ABS(5)	ABS(-5)	ABS(0)	ABS(NULL)
5	5	0	

SIGN

- Sign gives the sign of a value.

Syntax: – SIGN (value) **Example:**-

SQL>**SELECT** SIGN (5), SIGN (-5), SIGN (0), SIGN (**NULL**), SIGN (0.5) **FROM** DUAL;

SIGN(5)	SIGN(-5)	SIGN(0)	SIGN(NULL)	SIGN(0.5)
1	-1	0		1

SQRT

- This will give the square root of the given value.
- Input value must be positive.

Syntax: – SQRT (value)

Example:-

SQL>**SELECT SQRT** (2), **SQRT** (4), **SQRT** (9), **SQRT** (0), **SQRT** (**NULL**) **FROM** DUAL;

SQRT(2)	SQRT(4)	SQRT(9)	SQRT(0)	SQRT(NULL)
1.4142135623731	2	3	0	

In case, if we enter a negative number then system will raise an error.

SQL>**SELECT SQRT** (-2) **FROM** DUAL;

Output :

ORA-01428: argument '-2' **IS OUT OF** range

MOD

- This will give the remainder.

Syntax: – MOD (value, divisor)

Example:-

SQL>**SELECT MOD** (1,5),**MOD** (5,1),**MOD** (0,0),**MOD** (**NULL**, **NULL**),**MOD** (7,4),**MOD** (4,7),**MOD** (-4,7) **FROM** DUAL;

MOD(1,5)	MOD(5,1)	MOD(0,0)	MOD(NULL,NULL)	MOD(7,4)	MOD(4,7)	MOD(-4,7)
1	0	0		3	4	-4

NVL

- NVL replaces the NULL value with the specified value.
- Nvl function takes two input parameters.

I.e. If first parameter is null then it will return the second parameter otherwise it will return the first parameter.

Syntax: – NVL (,)

Example:-

SQL>**SELECT** NVL (**NULL**, 5), NVL (100, 10), NVL (**NULL**, **NULL**) **FROM** DUAL;

NVL(NULL,5)	NVL(100,10)	NVL(NULL,NULL)
5	100	

In emp table for Emplyee 'SMITH', "comm" value is null…

SQL>**SELECT** empno, ename, comm, deptno **FROM** emp **WHERE** empno=7369;

EMPNO	ENAME	COMM	DEPTNO
7369	SMITH		20

Let's use NVL function to convert the null value to known value for "comm" column in EMP table.

SQL>**SELECT** empno, ename, nvl (comm, 100) emp_comm, deptno **FROM** emp **WHERE** empno=7369;

EMPNO	ENAME	EMP_COMM	DEPTNO
7369	SMITH	100	20

POWER

- POWER function is used to raise a value to a given exponent.

Syntax: – POWER (value, exponent)

Example:-

SQL>**SELECT POWER** (2,3),**POWER** (3,2),**POWER** (1,1),**POWER** (0,0),**POWER** (**NULL**, **NULL**),**POWER** (-2,-3) **FROM** DUAL;

POWER(2,3)	POWER(3,2)	POWER(1,1)	POWER(0,0)	POWER(NULL,NULL)	POWER(-2,-3)
8	9	1	1		-0.125

EXP

- This function will raise the value to the given POWER.

Syntax: – EXP (value)

Example:-

SQL> **SELECT EXP** (0), **EXP** (1), **EXP** (2), **EXP** (**NULL**), **EXP** (-2) **FROM** DUAL;

EXP(0)	EXP(1)	EXP(2)	EXP(NULL)	EXP(-2)
1	2.71828182845905	7.38905609893065		0.135335283236613

LN

- This function is based on natural or base e logarithm.
- Input must be greater than zero.

Syntax: – LN (value)

Example:-

SQL>**SELECT** LN (1), LN (2), LN (**NULL**) **FROM** DUAL;

LN(1)	LN(2)	LN(NULL)
0	0.693147180559945	

Suppose we try **TO** enter a **VALUE** that **IS** zero **OR** negative **NUMBER THEN** system will raise an error.

SELECT LN (1), LN (2), LN (0), LN (**NULL**) **FROM** DUAL;

Output :

ORA-01428: argument '0' **IS OUT OF** range.

LN and EXP are reciprocal to each other.

EXP (3) = 20.0855369

LN (20.0855369) = 3

LOG

- This function is based on 10 based logarithms.
- Input value must be greater than zero which is positive only.

Syntax: – LOG (10, value)

Example:-

SQL>**SELECT** LOG (10, 1), LOG (10, 10), LOG (10, **NULL**) **FROM** DUAL;

LOG(10,1)	LOG(10,10)	LOG(10,NULL)
0	1	

SQL>**SELECT** LN (3), LOG (**EXP** (1), 3) **FROM** DUAL;

LN(3)	LOG(EXP(1),3)
1.09861228866811	1.09861228866811

CEIL

- This function will produce a whole number that is greater than or equal to the specified value.

Syntax: – CEIL (value)

Example:-

SQL>**SELECT CEIL**(5.1),**CEIL**(5),**CEIL**(-5),**CEIL**(-5.1),**CEIL**(0),**CEIL**(**NULL**) **FROM** DUAL;

CEIL(5.1)	CEIL(5)	CEIL(-5)	CEIL(-5.1)	CEIL(0)	CEIL(NULL)
6	5	-5	-5	0	

FLOOR

- This function will produce a whole number that is less than or equal to the specified value.

Syntax: – FLOOR (value)

Example:-

SQL> **SELECT FLOOR**(5.1),**FLOOR**(5),**FLOOR**(-5),**FLOOR**(-5.1),**FLOOR**(0),**FLOOR**(**NULL**) **FROM** DUAL;

FLOOR(5.1)	FLOOR(5)	FLOOR(-5)	FLOOR(-5.1)	FLOOR(0)	FLOOR(NULL)
5	5	-5	-6	0	

ROUND

- This function will ROUND numbers to a given number of digits of precision.

Syntax: – ROUND (value, precision)

Example:-

SQL>**SELECT** ROUND (123.2345), ROUND (123.2345, 2), ROUND (123.2354, 2) **FROM** DUAL;

ROUND(123.2345)	ROUND(123.2345,2)	ROUND(123.2354,2)
123	123.23	123.24

SQL>**SELECT** ROUND (123.2345, 0), ROUND (123.2345,-2), ROUND (123.2354,-4) **FROM** DUAL;

ROUND(123.2345,0)	ROUND(123.2345,-2)	ROUND(123.2354,-4)
123	100	0

TRUNC

- This function will truncates or chops off digits of precision from a number.

Syntax: – TRUNC (value, precision)

Example:-

SQL>**SELECT** TRUNC (123.2345), TRUNC (123.2345, 2), TRUNC (123.2354, 3) **FROM** DUAL;

TRUNC(123.2345)	TRUNC(123.2345,2)	TRUNC(123.2354,3)
123	123.23	123.235

BITAND

- This function will perform bitwise and operation.

Syntax: – BITAND (value1, value2)

Example:-

SQL>**SELECT** BITAND (2,3),BITAND (1,1),BITAND (0,0),BITAND (0,1),BITAND (**NULL**, **NULL**),BITAND (-1,-2) **FROM**

BITAND(2,3)	BITAND(1,1)	BITAND(0,0)	BITAND(0,1)	BITAND(NULL,NULL)	BITAND(-1,-2)
2	1	0	0		-2

GREATEST

- This function will give the greatest number in list of values.
- It will take 'n' number of arguments but data type should be same.

Syntax: – GREATEST (value1, value2….valuen)

Example:-

```
SQL>SELECT GREATEST (1, 4, NULL, 0,-5) FROM DUAL;
```

GREATEST(1,4,NULL,0,-5)

```
SQL>SELECT GREATEST (1, 4, 1000, 1.005,-5) FROM DUAL;
```

GREATEST(1,4,1000,1.005,-5)
1000

- If all the values are zeros then it will display zero.
- If all the parameters are nulls then it will display nothing.
- If any of the parameters is null it will display nothing.

LEAST

- This function will give the least number in list of values.
- It will take 'n' number of arguments but data type should be same.

Syntax: – LEAST (value1, value2….valuen)

Example:-

```
SQL>SELECT LEAST (1, 2, 3), LEAST (-1, -2, -3) FROM DUAL;
```

LEAST(1,2,3)	LEAST(-1,-2,-3)
1	-3

- If all the values are zeros then it will display zero.
- If all the parameters are nulls then it will display nothing.
- If any of the parameters is null it will display nothing.

COALESCE

- This function will return first non-null value in the specified list.

Syntax: – COALESCE (value1, value2….valuen)

Example:-

```
SQL>SELECT COALESCE (NULL, 2, NULL, 1) FROM DUAL;
```

COALESCE(NULL,2,NULL,1)
2

Date Functions:

- Oracle default date format is DD-MON-YYYY.

- We can change the default format to our desired format by using the following command. But this will expire once the session was closed.

```
SQL> ALTER SESSION SET NLS_DATE_FORMAT = 'DD-MONTH-YYYY';
```

Below are the Oracle DATE FUNCTIONS used to manipulate the date values:-

- **SYSDATE**
- **CURRENT_DATE**
- **CURRENT_TIMESTAMP**
- **SYSTIMESTAMP**
- **LOCALTIMESTAMP**
- **DBTIMEZONE**
- **SESSIONTIMEZONE**
- **TO_CHAR**
- **TO_DATE**
- **ADD_MONTHS**
- **MONTHS_BETWEEN**
- **NEXT_DAY**
- **LAST_DAY**
- **EXTRACT**
- **GREATEST**
- **LEAST**
- **ROUND**
- **TRUNC**
- **NEW_TIME**
- **COALESCE**

SYSDATE

- This will give the current date and time

Example:-

```
SQL>SELECT SYSDATE FROM DUAL;
```

SYSDATE
9/4/2012 9:01:49 AM

CURRENT_DATE

- This will returns the current date in the session's time zone.

Example:-

```
SQL>SELECT CURRENT_DATE FROM DUAL;
```

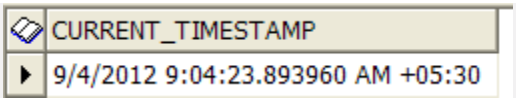
CURRENT_DATE
9/4/2012 9:03:16 AM

CURRENT_TIMESTAMP

- This will returns the current timestamp with the active time zone information.

Example:-

```
SQL>SELECT CURRENT_TIMESTAMP FROM DUAL;
```



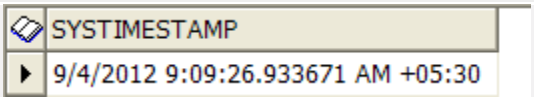
CURRENT_TIMESTAMP
9/4/2012 9:04:23.893960 AM +05:30

SYSTIMESTAMP

- This will returns the system date, including fractional seconds and time zone of the Database.

Example:-

```
SQL> SELECT SYSTIMESTAMP FROM DUAL;
```



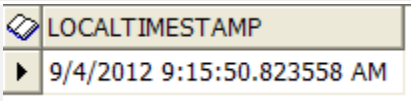
SYSTIMESTAMP
9/4/2012 9:09:26.933671 AM +05:30

LOCALTIMESTAMP

- This will returns local timestamp in the active time zone information, with no time zone information shown.

Example:-

```
SQL>SELECT LOCALTIMESTAMP FROM DUAL;
```



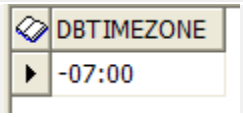
LOCALTIMESTAMP
9/4/2012 9:15:50.823558 AM

DBTIMEZONE

- This will returns the current database time zone in UTC format. (Coordinated Universal Time).

Example:-

```
SQL>SELECT DBTIMEZONE FROM DUAL;
```



DBTIMEZONE
-07:00

SESSIONTIMEZONE

- This will return the value of the current session's time zone**.**

  Example:-

```
SQL>SELECT SESSIONTIMEZONE FROM DUAL;
```

SESSIONTIMEZONE
+05:30

  TO_CHAR

- This will be used to EXTRACT various date formats.
- The available date formats as follows.

  Syntax: – TO_CHAR (date, format)

  DATE FORMATS:

Date Format	Description
DD	No of days in month
DDD	No of days in year
MM	No of month
MON	Three letter abbreviation of month
MONTH	Fully spelled out month
RM	Roman numeral month
DY	Three letters abbreviated day

DAY	Fully spelled out day
Y	Last one digit of the year
YY	Last two digits of the year
YYY	Last three digits of the year
YYYY	Full four digit year
SYYYY	Signed year
I	One digit year FROM ISO standard
IY	Two digit year from ISO standard
IYY	Three digit year from ISO standard
IYYY	Four digit year from ISO standard
Y, YYY	Year with comma
YEAR	Fully spelled out year
CC	Century
Q	No of quarters
W	No of weeks in month

WW	No of weeks in year
IW	No of weeks in year from ISO standard
HH	Hours
MI	Minutes
SS	Seconds
FF	Fractional seconds
AM or PM	Displays AM or PM depending upon time of day
A.M or P.M	Displays A.M or P.M depending upon time of day
AD or BC	Displays AD or BC depending upon the date
A.D or B.C	Displays AD or BC depending upon the date
FM	Prefix to month or day, suppresses padding of month or Day.
TH	Suffix to a number
SP	suffix to a number to be spelled out
SPTH	Suffix combination of TH and SP to be both spelled out
THSP	same as SPTH

Examples:-

```
SQL>SELECT TO_CHAR (SYSDATE,'dd-mm-yyyy hh24: mm:ss') FROM DUAL;
```

TO_CHAR(SYSDATE,'DD-MM-YYYYHH24:MM:SS')
04-09-2012 09:09:50

```
SQL>SELECT TO_CHAR (SYSDATE,'dd month year') FROM DUAL;
```

TO_CHAR(SYSDATE,'DDMONTHYEAR')
04 september twenty twelve

TO_DATE

- This will be used to convert the string into date format.

Syntax: - TO_DATE (date)

Example:-

```
SQL>SELECT TO_CHAR (TO_DATE ('24/dec/2006','dd/mon/yyyy'), 'dd * month * day') date_format FROM
DUAL;
```

DATE_FORMAT
24 * december * sunday

-- If you are not using TO_CHAR oracle will display output in default date format.

ADD_MONTHS

- This will add the specified months to the given date.

Syntax: - ADD_MONTHS (date, no_of_months)

Example:-

```
SQL> SELECT</span><span STYLE="font-family: Calibri;">ADD_MONTHS (SYSDATE, 5) FROM DUAL;
```

ADD_MONTHS(SYSDATE,5)
2/4/2013 10:03:11 AM

```
SQL>SELECT ADD_MONTHS (SYSDATE,-5) FROM DUAL;
```

ADD_MONTHS(SYSDATE,-5)
4/4/2012 10:04:46 AM

- If no_of_months is zero then it will display the same date.

```
SQL>SELECT ADD_MONTHS (SYSDATE, 0) FROM DUAL;
```

ADD_MONTHS(SYSDATE,0)
9/4/2012 10:08:48 AM

- If no_of_months is null then it will display nothing.

```
SQL>SELECT ADD_MONTHS (SYSDATE, NULL) FROM DUAL;
```

ADD_MONTHS(SYSDATE,NULL)

MONTHS_BETWEEN

- This will give difference of months between two dates.

Syntax: - MONTHS_BETWEEN (date1, date2)

Example:-

```
SQL>SELECT MONTHS_BETWEEN (ADD_MONTHS (SYSDATE, 5), SYSDATE) FROM DUAL;
```

MONTHS_BETWEEN(ADD_MONTHS(SYSDATE,5),SYSDATE)
5

```
SQL>SELECT MONTHS_BETWEEN (TO_DATE ('11-aug-1990','dd-mon-yyyy'), TO_DATE ('11- jan-1990','dd-mon-y
DUAL;
```

MON_BET
7

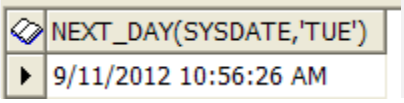
NEXT_DAY

- This will produce next day of the given day FROM the specified date.

Syntax: - NEXT_DAY (date, day)

Example:-

```
SQL>SELECT NEXT_DAY (SYSDATE,'TUE') FROM DUAL;
```



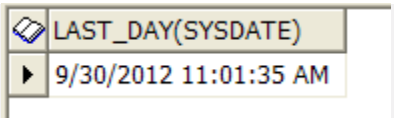
LAST_DAY

- This will produce last day of the given date.

Syntax: - LAST_DAY (date)

Example:-

```
SQL>SELECT LAST_DAY (SYSDATE) FROM DUAL;
```



EXTRACT

- This is used to EXTRACT a portion of the date value**.**

Syntax: - EXTRACT ((year | month | day | hour | minute | second) FROM date)

Example:-

```
SQL>SELECT EXTRACT (YEAR FROM SYSDATE) FROM DUAL;
```

EXTRACT(YEARFROMSYSDATE)
2012

```
SQL>SELECT EXTRACT (MONTH FROM SYSDATE) FROM DUAL;
```

EXTRACT(MONTHFROMSYSDATE)
9

```
SQL>SELECT EXTRACT (DAY FROM SYSDATE) FROM DUAL;
```

EXTRACT(DAYFROMSYSDATE)
4

GREATEST

- This will give the GREATEST date.
- It will take 'n' number of arguments but data type should be same.

Syntax: - GREATEST (date1, date2….daten)

Example:-

```
SQL>SELECT GREATEST (TO_DATE ('11-jan-1990','dd-mon-yyyy'), TO_DATE ('11-mar-1990','dd-mon-yyyy'), TO_
1990','dd-mon-yyyy')) FROM DUAL;
```

GREATEST_DATE
4/11/1990

- If all the values are zeros then it will display zero.
- If all the parameters are nulls then it will display nothing.
- If any of the parameters is null it will display nothing.

LEAST

- This will give the LEAST date.

- It will take 'n' number of arguments but data type should be same.

Syntax: - LEAST (date1, date2….daten)

Example:-

```
SQL>SELECT LEAST(TO_DATE('11-jan-1990','dd-mon-yyyy'),TO_DATE('11-mar-1990','dd-mon-yyyy'),TO_DATE('1
DUAL;
```

LEAST_DATE
1/11/1990

- If all the values are zeros then it will display zero.
- If all the parameters are nulls then it will display nothing.
- If any of the parameters is null it will display nothing.

ROUND

- ROUND will ROUNDs the date to which it was equal to or greater than the given date.

Syntax: - ROUND (date, (day | month | year))

- If the second parameter was year then ROUND will checks the month of the given date in

The following ranges**.**

JAN -- JUN

JUL -- DEC

- If the month falls between JAN and JUN then it returns the first day of the current year.

- If the month falls between JUL and DEC then it returns the first day of the next year.
- If the day falls between 1 and 15 then it returns the first day of the current month.

- If the day falls between 16 and 31 then it returns the first day of the next month.
- If the second parameter was day then ROUND will checks the week day of the given date

  In the following ranges.

  SUN -- WED

  THU -- SUN

- If the week day falls between SUN and WED then it returns the previous Sunday.
- If the weekday falls between THU and SUN then it returns the next Sunday.
- If the second parameter was null then it returns nothing.
- If we are not specifying the second parameter then ROUND will resets the time to the beginning of the current day in case of user specified date.
- If we are not specifying the second parameter then ROUND will resets the time to the beginning of the next day in case of SYSDATE**.**

Example:-

```
SQL>SELECT ROUND (TO_DATE ('24-dec-04','dd-mon-yy'),'year'), ROUND (TO_DATE ('11-mar-06','dd-mon-yy'),'
DUAL;
```

ROUND(TO_DATE('24-DEC-04','DD-MON-YY'),'YEAR')	ROUND(TO_DATE('11-MAR-06','DD-MON-YY'),'YEAR')
1/1/2005	1/1/2006

```
SQL>SELECT ROUND (TO_DATE ('11-jan-04','dd-mon-yy'),'month'), ROUND (TO_DATE ('18-jan-04','dd-mon-yy'),
DUAL
```

ROUND(TO_DATE('11-JAN-04','DD-MON-YY'),'MONTH')	ROUND(TO_DATE('18-JAN-04','DD-MON-YY'),'MONTH')
1/1/2004	2/1/2004

```
SQL>SELECT ROUND (TO_DATE ('26-dec-06','dd-mon-yy'),'day'), ROUND (TO_DATE ('29-dec-06','dd-mon-yy'),'d
DUAL;
```

ROUND(TO_DATE('26-DEC-06','DD-MON-YY'),'DAY')	ROUND(TO_DATE('29-DEC-06','DD-MON-YY'),'DAY')
12/24/2006	12/31/2006

TRUNC

- TRUNC will chops off the date to which it was equal to or less than the given date.

Syntax: - TRUNC (date, (day | month | year))

- If the second parameter was year then it always returns the first day of the current year.
- If the second parameter was month then it always returns the first day of the current month.
- If the second parameter was day then it always returns the previous Sunday.
- If the second parameter was null then it returns nothing.
- If you are not specifying the second parameter then trunk will resets the time to the beginning of the current day.

SQL>**SELECT** TRUNC (TO_DATE ('24-dec-04','dd-mon-yy'),'year'), TRUNC (TO_DATE ('11-mar-06','dd-mon-yy'),'y

TRUNC(TO_DATE('24-DEC-04','DD-MON-YY'),'YEAR')	TRUNC(TO_DATE('11-MAR-06','DD-MON-YY'),'YEAR')
1/1/2004	1/1/2006

SQL>**SELECT** TRUNC (TO_DATE ('11-jan-04','dd-mon-yy'),'month'), TRUNC (TO_DATE ('18-jul-04','dd-mon-yy'),'m

TRUNC(TO_DATE('11-JAN-04','DD-MON-YY'),'MONTH')	TRUNC(TO_DATE('18-JUL-04','DD-MON-YY'),'MONTH')
1/1/2004	7/1/2004

SQL>**SELECT** TRUNC (TO_DATE ('26-dec-06','dd-mon-yy'),'day'), TRUNC (TO_DATE ('29-dec-06','dd-mon-yy'),'da

TRUNC(TO_DATE('26-DEC-06','DD-MON-YY'),'DAY')	TRUNC(TO_DATE('29-DEC-06','DD-MON-YY'),'DAY')
12/24/2006	12/24/2006

NEW_TIME

- This will give the desired time zone's date and time.

Syntax: NEW_TIME (date, current_timezone, desired_timezone)

TIMEZONES

AST/ADT -- Atlantic standard/day light time.

BST/BDT -- Bering standard/day light time.

CST/CDT -- Central standard/day light time.

EST/EDT -- Eastern standard/day light time.

GMT -- Greenwich mean time.

HST/HDT -- Alaska-Hawaii standard/day light time.

MST/MDT -- Mountain standard/day light time.

NST -- Newfoundland standard time.

PST/PDT -- Pacific standard/day light time.

YST/YDT -- Yukon standard/day light time.

Example:-

```
SQL>SELECT TO_CHAR (NEW_TIME (SYSDATE,'gmt','yst'),'dd mon yyyy hh: mi: ss am') FROM DUAL;
```

TO_CHAR(NEW_TIME(SYSDATE,'GMT','YST'),'DDMONYYYYHH:MI:SSAM')
04 sep 2012 06:32:49 am

```
SQL>SELECT TO_CHAR (NEW_TIME (SYSDATE,'gmt','est'),'dd mon yyyy hh: mi: ss am') FROM DUAL;
```

TO_CHAR(NEW_TIME(SYSDATE,'GMT','EST'),'DDMONYYYYHH:MI:SSAM')
04 sep 2012 10:33:46 am

COALESCE

- This will give the first non-null date.

  Syntax: - COALESCE (date1, date2, date3….daten)

  Example:-

```
SQL>SELECT COALESCE ('12-jan-90','13-jan-99'), COALESCE (NULL,'12-jan-90','23-mar-98', NULL) FROM DUAL;
```

COALESCE('12-JAN-90','13-JAN-99')	COALESCE(NULL,'12-JAN-90','23-MAR-98',NULL)
12-jan-90	12-jan-90

Conversion Functions:

These functions are used to convert the values from one type to another type

Types:-

- **TO_NUMBER**
- **TO_CHAR**
- **TO_DATE**
- **Other Conversion Functions**
- **BIN_TO_NUM**
- **CHARTOROWID**
- **ROWIDTOCHAR**

Conversion Rules between data types:-

Data Type	Char	Num	Date
Char	–	**Yes**	**Yes**
Num	**Yes**	**–**	**Not Applicable**
Date	**Yes**	**Not Applicable**	**–**

TO_NUMBER

- To_Number will convert a char or varchar into number.

SQL>TO_NUMBER (**CHAR**, [format]);

Example:

SQL>**SELECT** TO_NUMBER ('1,23,456.56','9,99,999.99') **FROM** DUAL;

TO_NUMBER('1,23,456.56','99,99,999.99')
123456.56

TO_CHAR:-

- TO Char function will convert a number or date to character string.

Syntax: For number to char.
For date to char: – TO_CHAR (date, [format]).

Example:-

SQL>**SELECT** TO_CHAR (SYSDATE,'day') **FROM** DUAL;

TO_CHAR(SYSDATE,'DAY')
wednesday

SQL>**SELECT** TO_CHAR (123456,'99, 99,999.99') **FROM** DUAL;

TO_CHAR(123456,'99,99,999.99')
1,23,456.00

TO_DATE

- To date function will convert a char or varchar value into date value.

Syntax: – TO_DATE (char, [format])

Example:-

SQL>**SELECT** TO_DATE (12232012,'mmddyyyy') **FROM** DUAL;

TO_DATE('12232012','MMDDYYYY')
12/23/2012

SQL>**SELECT** TO_DATE ('02112012','ddmmyyyy') **FROM** DUAL;

TO_DATE('02112012','DDMMYYYY')
11/2/2012

SQL>**SELECT** TO_DATE ('20121102','yyyymmdd') **FROM** DUAL;

TO_DATE('20121102','YYYYMMDD')
11/2/2012

BIN_TO_NUM

BIN_TO_NUM will convert the binary value to its numerical equivalent.

Syntax: -

```
SQL>BIN_TO_NUM (binary_bits)
```

Example:- Binay bits passed are used to fetch the equivalent numeric value.

```
SQL>SELECT Bin_to_num (1, 0, 1) FROM DUAL;
```

BIN_TO_NUM(1,0,1)
5

- If all the bits are zero then it produces zero.
- If all the bits are null then it produces an error.

CHARTOROWID

- This will convert a character string to act like an internal oracle row identifier or rowid**.**

ROWIDTOCHAR

- This will convert an internal oracle row identifier/rowid to character string.

Group Functions:

- Group Functions are the functions which process a set of rows and return a single value.
- Group Functions are also called as aggregate functions (or) summary functions.
- Group functions will be applied on all the rows and produces single output.
- Group functions are mainly used to fetch summary information.

Types:-

- **SUM**

- **AVG**
- **MAX**
- **MIN**
- **COUNT**

- Max, Min and Count functions work with any data type.

- Sum and Avg functions work only with Number data types.

SUM

- This will give the sum of the values to the specified column.

Syntax: – **SUM** (
)

Example:-

SQL>**SELECT SUM** (sal) **FROM** EMP;

SUM(SAL)
31725

SQL>**SELECT** deptno, **SUM** (sal) **FROM** EMP **GROUP BY** deptno;

DEPTNO	SUM(SAL)
10	9050
20	12475
30	10200

AVG:-

- This will give the Average value

Syntax: – AVG (
)

SQL>**SELECT** AVG(sal) **FROM** emp **WHERE** deptno=20;

AVG(SAL)
66676.666

TRUNC

- This will give the average of the values of the specified column.

Syntax: – TRUNC (column)

Example:-

SQL>**SELECT** TRUNC (Avg (sal), 2) **FROM** emp;

TRUNC(AVG(SAL),2)
2266.07

SQL>**SELECT** deptno, TRUNC (Avg (sal), 2) **FROM** emp **GROUP BY** deptno;

DEPTNO	TRUNC(AVG(SAL),2)
10	3016.66
20	2495
30	1700

MAX

- This will give the maximum of the values of the specified column.

Syntax: – MAX (column)

Example:-

SQL> **SELECT MAX** (sal) **FROM** emp;

MAX(SAL)
5100

SQL>**SELECT** deptno, **MAX** (sal) **FROM** emp **GROUP BY** deptno;

DEPTNO	MAX(SAL)
10	5100
20	3100
30	2950

MIN

- This will give the minimum of the values of the specified column.

Syntax: – MIN (column)

Example:-

SQL>**SELECT MIN** (sal) **FROM** emp;

MIN(SAL)
1050

SQL>**SELECT** deptno, **MIN** (sal) **FROM** emp **GROUP BY** deptno;

DEPTNO	MIN(SAL)
10	1400
20	1200
30	1050

COUNT

- This will give the count of the values of the specified column.

Syntax: – COUNT (column)

Example:-

SQL>**SELECT COUNT** (*) **FROM** emp;

COUNT(*)
14

SQL>**SELECT** deptno, **COUNT** (*) **FROM** emp **GROUP BY** deptno;

DEPTNO	COUNT(*)
10	3
20	5
30	6

Chapter 15: Joins

What Is a Join?

- A Join is nothing but a temporary relation between the common columns of the table (or) the column sharing the common data.
- Join is a mechanism to retrieve data from more than one data base tables.
- The purpose of a join is to combine the data across tables.
- A join is actually performed by the where clause which combines the specified rows of tables**.**
- JOINS are always defined only between two tables.
- When we read the data from more than one table join is mandatory.
- The default join working in the background is called CROSS JOIN.
- If 'N' tables are there then at least (N-1) joins are needed.
- A join is a set operation applicable only to RDBMS.
- REFERENCES are a constraint used to define the relationship between the columns of different tables but it is a constraint validates the data only during the DML operation.
- Joins are defined at the application level, because of which they are temporary.
- Retrieving the data from more than one table without a join leads to Cartesian product i.e.,. All possible combination of rows.
- Join is a condition to get only the relevant matching rows of different tables.

Joins are categorized based on their usage, below are different types of Joins:-

- **Inner Join.**
- **Outer Join.**

Inner Join

Inner join is to retrieve the matching rows of joining tables (or) to retrieve the rows of different tables those are having matching values.

Types of inner Joins:

- Equi Joins
- Non-Equi Joins

Outer join again has classification based on its usage

- Outer Join is used to retrieve the rows of tables that have non-matching values along with matching values.
- '(+)' is the symbol used to represent outer join.

Types of Outer Joins:-

- Left outer join.

- Right Outer Join.
- Full Outer Join.

Inner Join (or) Simple Join (or) Equi Join:

- Inner Join is to retrieve the rows of different tables that having matching values.
- When two tables are joined together using equality of values in one or more columns, they make an Equi Join.
- Inner Join is also called as Simple Joins or Equi Joins.
- Relationship between the columns of different tables is established by using an operator '='.

We will be using EMP and DEPT tables for the demonstration of joins. Please get acquainted with the EMP and DEPT table data.

```
SQL>SELECT * FROM dept;
```

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

```
SQL> SELECT * FROM emp;
```

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7369	SMITH	CLERK	7902	12/17/1980	800		20
7499	ALLEN	SALESMAN	7698	2/20/1981	1600	300	30
7521	WARD	SALESMAN	7698	2/22/1981	1250	500	30
7566	JONES	MANAGER	7839	4/2/1981	2975		20
7654	MARTIN	SALESMAN	7698	9/28/1981	1250	1400	30
7698	BLAKE	MANAGER	7839	5/1/1981	2850		30
7782	CLARK	MANAGER	7839	6/9/1981	2450		10
7788	SCOTT	ANALYST	7566	12/9/1982	3000		20
7839	KING	PRESIDENT		11/17/1981	5000		10
7844	TURNER	SALESMAN	7698	9/8/1981	1500		30
7876	ADAMS	CLERK	7788	1/12/1983	1100		20
7900	JAMES	CLERK	7698	12/3/1981	950		30
7902	FORD	ANALYST	7566	12/3/1981	3000		20
7934	MILLER	CLERK	7782	1/23/1982	1300		10

EMP and DEPT table data are joined using '=' operator with DEPT column on both tables.

```
SQL>SELECT emp. empno,
   emp. ename,
   emp. deptno,
   dept. dname,
   dept.loc
   FROM EMP, DEPT
   WHERE emp.deptno=dept.deptno;
```

EMPNO	ENAME	DEPTNO	DNAME	LOC
7782	CLARK	10	ACCOUNTING	NEW YORK
7839	KING	10	ACCOUNTING	NEW YORK
7934	MILLER	10	ACCOUNTING	NEW YORK
7369	SMITH	20	RESEARCH	DALLAS
7876	ADAMS	20	RESEARCH	DALLAS
7902	FORD	20	RESEARCH	DALLAS
7788	SCOTT	20	RESEARCH	DALLAS
7566	JONES	20	RESEARCH	DALLAS
7499	ALLEN	30	SALES	CHICAGO
7698	BLAKE	30	SALES	CHICAGO
7654	MARTIN	30	SALES	CHICAGO
7900	JAMES	30	SALES	CHICAGO
7844	TURNER	30	SALES	CHICAGO
7521	WARD	30	SALES	CHICAGO

Using Table Aliases:-

SQL>**SELECT** E.empno, E.ename, E.deptno, D.dname, D.loc **FROM** emp E, dept D **WHERE** E.deptno=D.deptno D.deptno=10;

EMPNO	ENAME	DEPTNO	DNAME	LOC
7782	CLARK	10	ACCOUNTING	NEW YORK
7839	KING	10	ACCOUNTING	NEW YORK
7934	MILLER	10	ACCOUNTING	NEW YORK

Here E,D are alias names for EMP and DEPT Tables and these are to used to avoid the ambiguity problems.

We can write the same query using 'USING CLAUSE' instead of '='.

A different representation of above query, however data returned will be the same.

SQL>**SELECT** empno, ename, deptno, dname, loc **FROM** emp **JOIN** dept **USING** (deptno) **WHERE** deptno=10;

EMPNO	ENAME	DEPTNO	DNAME	LOC
7782	CLARK	10	ACCOUNTING	NEW YORK
7839	KING	10	ACCOUNTING	NEW YORK
7934	MILLER	10	ACCOUNTING	NEW YORK

Left Outer Join:

- Left Outer join will fetch all records from left hand side table in a join, irrespective of the matching.
- In other words Left outer join will display the all the records which are in left hand side table but not in right hand side table, along with the matching records.
- '(+)' is the symbol used to represent outer join.
- And in Left Outer join we place this symbol '(+)' on left side of join.

Examples:-

For Left outer join we will be using EMP and DEPT tables with deptno as join between the tables.

SQL>**SELECT** * **FROM** dept;

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

– For record with deptno = 40 there are no child records in EMP table, we will be using this record for demonstration.

SQL>**SELECT** * **FROM** emp;

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7369	SMITH	CLERK	7902	12/17/1980	800		20
7499	ALLEN	SALESMAN	7698	2/20/1981	1600	300	30
7521	WARD	SALESMAN	7698	2/22/1981	1250	500	30
7566	JONES	MANAGER	7839	4/2/1981	2975		20
7654	MARTIN	SALESMAN	7698	9/28/1981	1250	1400	30
7698	BLAKE	MANAGER	7839	5/1/1981	2850		30
7782	CLARK	MANAGER	7839	6/9/1981	2450		10
7788	SCOTT	ANALYST	7566	12/9/1982	3000		20
7839	KING	PRESIDENT		11/17/1981	5000		10
7844	TURNER	SALESMAN	7698	9/8/1981	1500		30
7876	ADAMS	CLERK	7788	1/12/1983	1100		20
7900	JAMES	CLERK	7698	12/3/1981	950		30
7902	FORD	ANALYST	7566	12/3/1981	3000		20
7934	MILLER	CLERK	7782	1/23/1982	1300		10

SQL>**SELECT** e.empno, e.ename, d.deptno, d.dname, d.loc **FROM** emp e, dept d **WHERE** e.deptno (+) =d.deptno;

EMPNO	ENAME	DEPTNO	DNAME	LOC
7782	CLARK	10	ACCOUNTING	NEW YORK
7839	KING	10	ACCOUNTING	NEW YORK
7934	MILLER	10	ACCOUNTING	NEW YORK
7369	SMITH	20	RESEARCH	DALLAS
7876	ADAMS	20	RESEARCH	DALLAS
7902	FORD	20	RESEARCH	DALLAS
7788	SCOTT	20	RESEARCH	DALLAS
7566	JONES	20	RESEARCH	DALLAS
7499	ALLEN	30	SALES	CHICAGO
7698	BLAKE	30	SALES	CHICAGO
7654	MARTIN	30	SALES	CHICAGO
7900	JAMES	30	SALES	CHICAGO
7844	TURNER	30	SALES	CHICAGO
7521	WARD	30	SALES	CHICAGO
		40	OPERATIONS	BOSTON

From result we can observe that there is a additional record from DEPT table because of Left Outer Join.

The corresponding columns on EMP side will have NULL values.

Right Outer Join:

- Right Outer join will fetch all records from Right hand side table in a join, irrespective of the matching.
- In other words Right outer join will display the all the records which are in Right hand side table but not in left hand side table, along with the matching records.
- '(+)' is the symbol used to represent outer join.
- And in Right Outer join we place this symbol '(+)' on Right side of join.

Examples:

For Right outer join we will be using EMP and DEPT tables with deptno as join between the tables.

```
SQL>SELECT * FROM dept;
```

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

```
SQL>SELECT * FROM emp;
```

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7369	SMITH	CLERK	7902	12/17/1980	800		20
7499	ALLEN	SALESMAN	7698	2/20/1981	1600	300	30
7521	WARD	SALESMAN	7698	2/22/1981	1250	500	30
7566	JONES	MANAGER	7839	4/2/1981	2975		20
7654	MARTIN	SALESMAN	7698	9/28/1981	1250	1400	30
7698	BLAKE	MANAGER	7839	5/1/1981	2850		30
7782	CLARK	MANAGER	7839	6/9/1981	2450		10
7788	SCOTT	ANALYST	7566	12/9/1982	3000		20
7839	KING	PRESIDENT		11/17/1981	5000		10
7844	TURNER	SALESMAN	7698	9/8/1981	1500		30
7876	ADAMS	CLERK	7788	1/12/1983	1100		20
7900	JAMES	CLERK	7698	12/3/1981	950		30
7902	FORD	ANALYST	7566	12/3/1981	3000		20
7934	MILLER	CLERK	7782	1/23/1982	1300		10

– we can see that there is a new record in EMP table with empno = 3567 and ename = ‘SQL’ . the same will be used for demonstration.

```
SQL>SELECT e.empno, e.ename, d.deptno, d.dname, d.loc FROM emp e, dept d WHERE e.deptno =d.deptno(+);
```

EMPNO	ENAME	DEPTNO	DEPTNO_1	DNAME	LOC
7782	CLARK	10	10	ACCOUNTING	NEW YORK
7839	KING	10	10	ACCOUNTING	NEW YORK
7934	MILLER	10	10	ACCOUNTING	NEW YORK
7369	SMITH	20	20	RESEARCH	DALLAS
7876	ADAMS	20	20	RESEARCH	DALLAS
7902	FORD	20	20	RESEARCH	DALLAS
7788	SCOTT	20	20	RESEARCH	DALLAS
7566	JONES	20	20	RESEARCH	DALLAS
7499	ALLEN	30	30	SALES	CHICAGO
7698	BLAKE	30	30	SALES	CHICAGO
7654	MARTIN	30	30	SALES	CHICAGO
7900	JAMES	30	30	SALES	CHICAGO
7844	TURNER	30	30	SALES	CHICAGO
7521	WARD	30	30	SALES	CHICAGO
3567	SQL	50			

From result we can observe that there is a additional record from EMP table because of Right Outer Join.

The corresponding columns on DEPT side will have NULL values.

Full Outer Join:

- Full Outer Join is to retrieve all the matching data and all additional non-matching data from both left and right side of the tables.
- In Other words Full Outer join will display the all matching records and the non-matching records from both tables.
- Full outer join is a combination of both Left outer and Right outer join

For full outer join we will be using the key word 'FULL OUTER JOIN' instead of symbol '(+)'.

Examples:-
For Full outer joins we will be using EMP and DEPT tables with deptno as join between the tables.

SQL>**SELECT** * **FROM** dept;

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

SQL>**SELECT** * **FROM** emp;

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7369	SMITH	CLERK	7902	12/17/1980	800		20
7499	ALLEN	SALESMAN	7698	2/20/1981	1600	300	30
7521	WARD	SALESMAN	7698	2/22/1981	1250	500	30
7566	JONES	MANAGER	7839	4/2/1981	2975		20
7654	MARTIN	SALESMAN	7698	9/28/1981	1250	1400	30
7698	BLAKE	MANAGER	7839	5/1/1981	2850		30
7782	CLARK	MANAGER	7839	6/9/1981	2450		10
7788	SCOTT	ANALYST	7566	12/9/1982	3000		20
7839	KING	PRESIDENT		11/17/1981	5000		10
7844	TURNER	SALESMAN	7698	9/8/1981	1500		30
7876	ADAMS	CLERK	7788	1/12/1983	1100		20
7900	JAMES	CLERK	7698	12/3/1981	950		30
7902	FORD	ANALYST	7566	12/3/1981	3000		20
7934	MILLER	CLERK	7782	1/23/1982	1300		10

– we can see that there is a new record in EMP table with empno = 3567 and ename = 'SQL' . the same will be used for demonstration.

Here we are writing full outer join by using **"full outer"…**

```
SQL>SELECT e.empno, e.ename, e.job, e.sal, e.deptno "EMP_DEPTNO", d.dname, d.loc
FROM emp e FULL OUTER JOIN DEPT d
ON (e.deptno=d.deptno);
```

From result we can observe that there is a additional record from both EMP and DEPT table as this is full outer join.

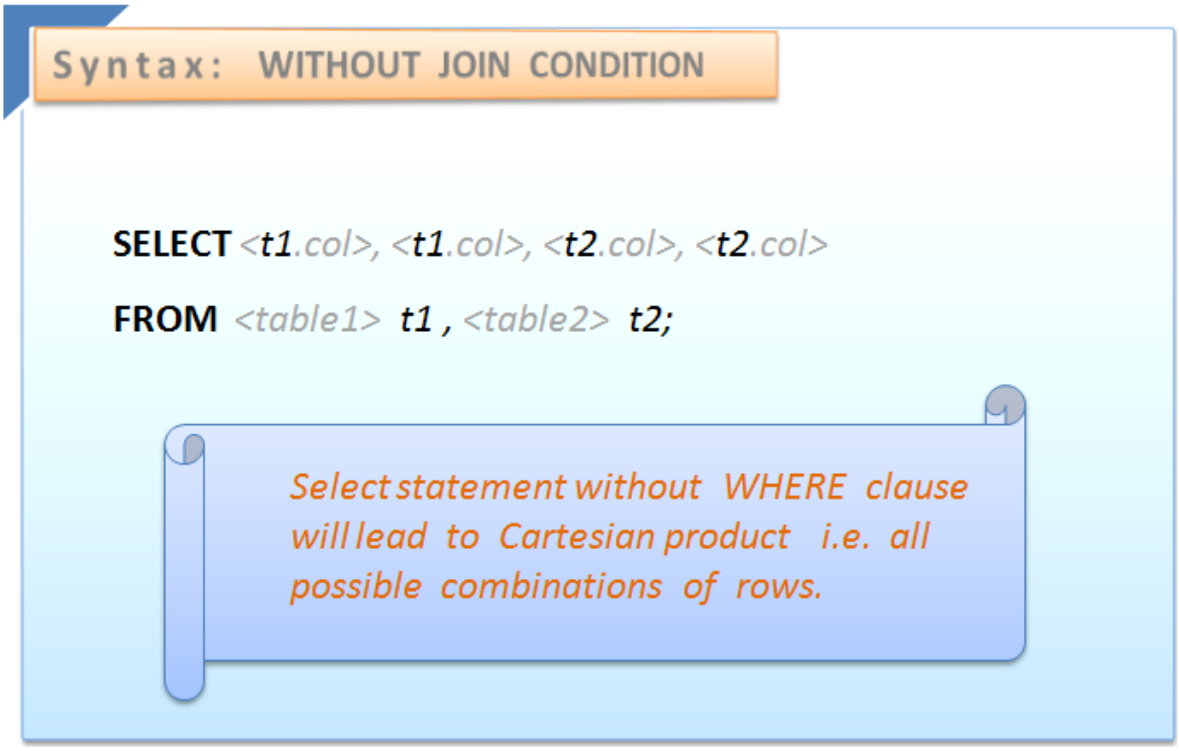
The corresponding columns that don't have data will have NULL values.

Cross Join:

- Cross join will fetch all possible combinations between the tables.
- Cross Join is same like not specifying any condition between the tables.
- Retrieving the data from more than one data base table without join condition leads to Cartesian product
  (or) cross product i.e. all possible combination of rows.

When it will be occurring:-

- When the join condition omitted.
- Condition not specified in the where clause.



By using "**cross join**" key word

Examples:-

SQL> **SELECT** * **FROM** dept;

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

SQL>**SELECT** * **FROM** emp;

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7369	SMITH	CLERK	7902	12/17/1980	800		20
7499	ALLEN	SALESMAN	7698	2/20/1981	1600	300	30
7521	WARD	SALESMAN	7698	2/22/1981	1250	500	30
7566	JONES	MANAGER	7839	4/2/1981	2975		20
7654	MARTIN	SALESMAN	7698	9/28/1981	1250	1400	30
7698	BLAKE	MANAGER	7839	5/1/1981	2850		30
7782	CLARK	MANAGER	7839	6/9/1981	2450		10
7788	SCOTT	ANALYST	7566	12/9/1982	3000		20
7839	KING	PRESIDENT		11/17/1981	5000		10
7844	TURNER	SALESMAN	7698	9/8/1981	1500		30
7876	ADAMS	CLERK	7788	1/12/1983	1100		20
7900	JAMES	CLERK	7698	12/3/1981	950		30
7902	FORD	ANALYST	7566	12/3/1981	3000		20
7934	MILLER	CLERK	7782	1/23/1982	1300		10
3567	SQL						50

SQL>**SELECT** empno, ename, sal, dname, loc **FROM** emp **CROSS JOIN** dept;

Or

```
SQL>SELECT empno,
   ename,
   sal,
   dname,
   loc
   FROM emp, dept;
```

These queries will fetch 56 rows in total as EMP table has 14 rows and DEPT table has 5 rows .i.e. 14*4=56 rows will be displayed

By using cross join we can retrieve all possible combinations

Non-Equi Join:

- If the relation between the columns of different tables is established by using an operator other than '=' than that join is called NON- Equi join.

  i.e., A join which contains an operator other than '=' in the joins condition but not != (not equls to).
  In a normal case we donot use 'Non Equi joins' as result data will have ambiguous results.

  Examples:-
  Here we are using two tables EMP and SALGRADE and these tables having no primary key and foreign key relationship

SQL>**SELECT** * **FROM** emp;

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7369	SMITH	CLERK	7902	12/17/1980	800		20
7499	ALLEN	SALESMAN	7698	2/20/1981	1600	300	30
7521	WARD	SALESMAN	7698	2/22/1981	1250	500	30
7566	JONES	MANAGER	7839	4/2/1981	2975		20
7654	MARTIN	SALESMAN	7698	9/28/1981	1250	1400	30
7698	BLAKE	MANAGER	7839	5/1/1981	2850		30
7782	CLARK	MANAGER	7839	6/9/1981	2450		10
7788	SCOTT	ANALYST	7566	12/9/1982	3000		20
7839	KING	PRESIDENT		11/17/1981	5000		10
7844	TURNER	SALESMAN	7698	9/8/1981	1500		30
7876	ADAMS	CLERK	7788	1/12/1983	1100		20
7900	JAMES	CLERK	7698	12/3/1981	950		30
7902	FORD	ANALYST	7566	12/3/1981	3000		20
7934	MILLER	CLERK	7782	1/23/1982	1300		10

SQL> **SELECT** * **FROM** salgrade;

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

```
SQL>SELECT e.empno,
      e.ename,
      e.deptno,
      s.grade
      FROM emp e,
      salgrade s
      WHERE e.sal BETWEEN s.losal AND s.hisal;
```

EMPNO	ENAME	DEPTNO	GRADE
7369	SMITH	20	1
7876	ADAMS	20	1
7900	JAMES	30	1
7521	WARD	30	2
7654	MARTIN	30	2
7934	MILLER	10	2
7499	ALLEN	30	3
7844	TURNER	30	3
7566	JONES	20	4
7698	BLAKE	30	4
7782	CLARK	10	4
7788	SCOTT	20	4
7902	FORD	20	4
7839	KING	10	5

Chapter 16: Sub queries

Sub Query:

- A sub query is a SELECT statement which is embedded in another SELECT statement .
- Sub queries are very useful to achieve complex requirements.
- We can write a sub query in any part of the select statement.

- - The SELECT Clause.
- - The FROM Clause.
- - The WHERE Clause.

- Sub queries are always enclosed with in parenthesis.
- We can write any number of sub queries in a select statement.
- Execution of the select statement will always start from its sub queries.

Examples:-

Simple sub query

```
SQL> SELECT * FROM emp
                WHERE job  = (SELECT job FROM emp WHERE ename='KING');
```

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7839	KING	PRESIDENT		11/17/1981	5100		10

Sub queries were given different names based on its usage.

- Single row sub queries.
- Multi row sub queries.
- Correlated sub queries.

SINGLE ROW SUBQUERIES:

- Single row sub query is a sub query which will return only one value.

```
SQL>SELECT * FROM emp
                WHERE sal > (SELECT sal FROM emp WHERE empno = 7566);
```

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7788	SCOTT	ANALYST	7566	12/9/1982	3100		20
7839	KING	PRESIDENT		11/17/1981	5100		10
7902	FORD	ANALYST	7566	12/3/1981	3100		20

MULTI ROW SUBQUERIES:

Multi row sub query is a sub query which will return more than one value. In such cases we should include
operators like 'ANY, ALL, IN or NOT IN' along with the sub query.

```
SQL>SELECT * FROM emp
            WHERE sal > ANY (SELECT sal FROM emp WHERE sal BETWEEN 2500 AND 4000);
```

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7566	JONES	MANAGER	7839	4/2/1981	3075		20
7698	BLAKE	MANAGER	7839	5/1/1981	2950		30
7788	SCOTT	ANALYST	7566	12/9/1982	3100		20
7839	KING	PRESIDENT		11/17/1981	5100		10
7902	FORD	ANALYST	7566	12/3/1981	3100		20

```
SQL>SELECT * FROM emp
            WHERE sal  > ALL (SELECT sal FROM emp WHERE sal BETWEEN 2500 AND 4000);
```

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7839	KING	PRESIDENT		11/17/1981	5100		10

```
SQL>SELECT * FROM emp
            WHERE sal  IN (SELECT sal FROM emp WHERE deptno=10);
```

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7934	MILLER	CLERK	7782	1/23/1982	1680		10
7782	CLARK	MANAGER	7839	6/9/1981	2550		10
7839	KING	PRESIDENT		11/17/1981	5100		10

CORELATED SUBQUERIES

A sub query is called correlated sub query if the sub query has a relation with the parent query.

A normal sub query is evaluated once for the entire parent statement where as a correlated sub query is
evaluated for every row processed by the parent statement.

```
SQL> SELECT DISTINCT * FROM emp e
            WHERE &n = (SELECT COUNT (DISTINCT (b.sal))
                                    FROM emp b WHERE  sal>= e.sal);
```

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7698	BLAKE	MANAGER	7839	5/1/1981	2950		30

Sub Query In From Clause:

- In the FROM clause of the SELECT statement we specify tables or views as data sources. However along with tables and views we can as well specify sub queries as data sources which are called as inline views
- Inline view is a named sub query in the FROM clause of the main query.
- Inline view (sub query) should have an alias and we can use it just like a view – inside a SQL statement.

Example1:-
In the below example FROM clause contains sub query as a data source.

```
SQL>SELECT ename,
    sal,
    rownum rank
    FROM (SELECT * FROM emp ORDER BY sal);
```

ENAME	SAL	RANK
ADAMS	1440	2
WARD	1620	3
MARTIN	1620	4
MILLER	1680	5
TURNER	2040	6
ALLEN	2160	7
SMITH	2400	8
CLARK	2550	9
BLAKE	2950	10
JONES	3075	11
SCOTT	3100	12
FORD	3100	13
KING	5100	14

Example2:-

```
SQL>SELECT * FROM (SELECT rownum rno, emp.* FROM emp)
   WHERE rno=&1;
```

RNO	EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
2	7499	ALLEN	SALESMAN	7698	2/20/1981	2160	300	30

Sub Query in Select Clause:

- Sub-Query in SELECT clause is nothing but providing SELECT statement in the place of column name.
- Sub query in SELECT should always return one value.
- It can be Independent query or it can be dependent by joining with one of table/view in from clause.

Example1:-

```
SQL>SELECT empno, ename, sal, (SELECT MAX (sal) FROM emp) maxsal FROM emp;
```

EMPNO	ENAME	SAL	MAXSAL
7369	SMITH	2000	5100
7499	ALLEN	1800	5100
7521	WARD	1350	5100
7566	JONES	3075	5100
7654	MARTIN	1350	5100
7698	BLAKE	2950	5100
7782	CLARK	2550	5100
7788	SCOTT	3100	5100
7839	KING	5100	5100
7844	TURNER	1700	5100
7876	ADAMS	1200	5100
7900	JAMES	1050	5100
7902	FORD	3100	5100
7934	MILLER	1400	5100

- In the above example we have provided 'SELECT STATEMENT' in the place of column name which will return only single value.

If we try to return multiple values, system will raise an error which is shown below.

```
SQL>SELECT empno, ename, sal, (SELECT MAX (sal) FROM emp GROUP BY deptno) maxsal FROM emp;
```

Output:-

```
ORA-01427: single-row sub query returns more than one row.
```

In the above 'SELECT STATEMENT' use of 'GROUP BY' returns multiple values.

Example2:-

In this example we are trying to fetch highest and lowest salary for each department.

```
SQL>SELECT empno, ename, sal, deptno,
```

```
(SELECT MIN (sal) FROM emp WHERE deptno=e.deptno)losal,
(SELECT MAX (sal) FROM emp WHERE deptno=e.deptno) hisal
 FROM EMP e ORDER BY deptno;
```

EMPNO	ENAME	SAL	DEPTNO	LOSAL	HISAL
7782	CLARK	2550	10	1400	5100
7839	KING	5100	10	1400	5100
7934	MILLER	1400	10	1400	5100
7369	SMITH	2000	20	1200	3100
7876	ADAMS	1200	20	1200	3100
7902	FORD	3100	20	1200	3100
7788	SCOTT	3100	20	1200	3100
7566	JONES	3075	20	1200	3100
7499	ALLEN	1800	30	1050	2950
7698	BLAKE	2950	30	1050	2950
7654	MARTIN	1350	30	1050	2950
7900	JAMES	1050	30	1050	2950
7844	TURNER	1700	30	1050	2950
7521	WARD	1350	30	1050	2950

Sub Query In Where Clause:

- Sub-Query in WHERE clause is basically used for conditions. i.e. if a condition has to be driven based on a value retrieved by query then we use sub query in where clause.

Example1:-

Display the employees who are working in location 'CHICAGO'. (this query can be written in different ways and below is one of the way)

```
SQL>SELECT empno, ename, sal, deptno FROM emp WHERE deptno IN(SELECT deptno FROM dept WHERE loc='CHICAGO');
```

EMPNO	ENAME	SAL	DEPTNO
7499	ALLEN	1800	30
7521	WARD	1350	30
7654	MARTIN	1350	30
7698	BLAKE	2950	30
7844	TURNER	1700	30
7900	JAMES	1050	30

Example 2:-

Display the employees whose salary is less than average pay

SQL>SELECT empno, ename, deptno FROM emp WHERE sal <(SELECT AVG (sal) FROM emp) ORDER BY sal);

EMPNO	ENAME	DEPTNO	SAL
7900	JAMES	30	1050
7876	ADAMS	20	1200
7521	WARD	30	1350
7654	MARTIN	30	1350
7934	MILLER	10	1400
7844	TURNER	30	1700
7499	ALLEN	30	1800
7369	SMITH	20	2000

Example 3:-

Raise the salary by 20% for the employees whose salary is less than Average salary

```
SQL>UPDATE emp SET sal=sal+sal*.25 WHERE sal < (SELECT AVG(sal) FROM emp);
```

Nested Sub Queries:

- A query with in a Sub-Query is nothing but Nested Sub Query.
- To achieve complex requirements we will keep adding sub queries, resulting in nested structure.

Example1:-

In the below example we are trying to fetch Grand parent record. (the 'MANAGER' of 'TURNER' manager)

```
SQL> SELECT *FROM emp WHERE empno
   IN (SELECT mgr FROM emp WHERE
   empno IN (SELECT mgr FROM emp WHERE ename ='TURNER'));
```

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7839	KING	PRESIDENT		11/17/1981	5100		10

Example2:-

Display the second highest salary in emp table.

SQL>SELECT * FROM emp
WHERE sal = (SELECT MAX (sal) FROM emp WHERE sal < (SELECT MAX (sal) FROM EMP));

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7788	SCOTT	ANALYST	7566	12/9/1982	3100		20
7902	FORD	ANALYST	7566	12/3/1981	3100		20

Chapter 17: Set Operators

UNION:

- UNION Operator combines the results of two select statements into one result set, and then eliminates
any duplicates rows from the final result set.

For Set operators, to demonstrate and make you understand better we will be creating two new tables ‘Dept2’ and ‘Dept3’ with department information. Observe the script and data carefully:

SQL>SELECT * FROM dept2;

DEPTNO	DNAME	LOC
20	RESEARCH	DALLAS
30	SALES	CHICAGO
40	OPERATIONS	BOSTON
10	ACCOUNTING	NEW YORK
50	HR	SAN DIEGO
60	FINANCE	COLUMBUS

SQL>SELECT * FROM dept3;

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

By using union operator,

SQL>SELECT * FROM dept2 UNION SELECT * FROM dept3;

DEPTNO	DNAME	LOC
10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS
30	SALES	CHICAGO
40	OPERATIONS	BOSTON
50	HR	SAN DIEGO
60	FINANCE	COLUMBUS
70	FACILITIES	BOSTON

From result we can see that records from dept 2 and dept 3 are combined without having any duplicates.

Union All:

- UNION ALL Operator combines the results of two select statements into one result set including Duplicates.

For Set operators, to demonstrate and make you understand better we will be creating two new tables ‘Dept2’ and ‘Dept3’ with department information. Observe the script and data carefully:

SQL>SELECT * FROM dept2;

DEPTNO	DNAME	LOC
20	RESEARCH	DALLAS
30	SALES	CHICAGO
40	OPERATIONS	BOSTON
10	ACCOUNTING	NEW YORK
50	HR	SAN DIEGO
60	FINANCE	COLUMBUS

SQL>SELECT * FROM dept3;

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

By using union all operator,

```
SQL>SELECT * FROM dept2

 UNION ALL

 SELECT * FROM dept3;
```

DEPTNO	DNAME	LOC
10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS
30	SALES	CHICAGO
40	OPERATIONS	BOSTON
50	HR	SAN DIEGO
60	FINANCE	COLUMBUS
10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS
30	SALES	CHICAGO
40	OPERATIONS	BOSTON
70	FACILITIES	BOSTON

From result we can see that records from dept2 and dept3 are combined along with duplicate records.

INTERSECT:

- INTERSECT Operator returns only those rows that are common in both tables.

For Set operators, to demonstrate and make you understand better we will be creating two new tables 'Dept2' and 'Dept3' with department information. Observe the script and data carefully:

SQL>SELECT * FROM dept2;

DEPTNO	DNAME	LOC
20	RESEARCH	DALLAS
30	SALES	CHICAGO
40	OPERATIONS	BOSTON
10	ACCOUNTING	NEW YORK
50	HR	SAN DIEGO
60	FINANCE	COLUMBUS

SQL>SELECT * FROM dept3;

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

By using intersect operator,

```
SQL>SELECT * FROM dept2
    INTERSECT
    SELECT * FROM dept3;
```

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

From the result we can see that common records are fetched from both the tables.

MINUS:

- MINUS Operator takes the result set of first select statement and removes those rows that are returned by a second select statement.

For Set operators, to demonstrate and make you understand better we will be creating two new tables 'Dept2' and 'Dept3' with department information. Observe the script and data carefully:

```
SQL>SELECT * FROM dept2;
```

DEPTNO	DNAME	LOC
20	RESEARCH	DALLAS
30	SALES	CHICAGO
40	OPERATIONS	BOSTON
10	ACCOUNTING	NEW YORK
50	HR	SAN DIEGO
60	FINANCE	COLUMBUS

SQL>SELECT * FROM dept3;

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

By using minus operator:

```
SQL>SELECT * FROM dept2
   MINUS
   SELECT * FROM dept3;
```

DEPTNO	DNAME	LOC
50	HR	SAN DIEGO
60	FINANCE	COLUMBUS

From result we can see that records from dept 3 are removed from dept2.

Chapter 18: Indexes

- INDEX is a pointer locates the physical address of data.
- We will be creating indexes explicitly to speed up SQL statement execution on a table.
- It is automatically activated when index column is used in “where” clause.
- Indexes can be created on a single column or a group of columns.
- When an index is created, it first sorts the data and then it assigns a ROWID for each row.
- When there are thousands of records in a table, retrieving information will have performance issue. Therefore indexes are created on columns which are accessed frequently, so that the information can be retrieved quickly.

TYPES OF INDEXES:-

- Unique index
- Non-unique index
- Composite index
- Function-based index

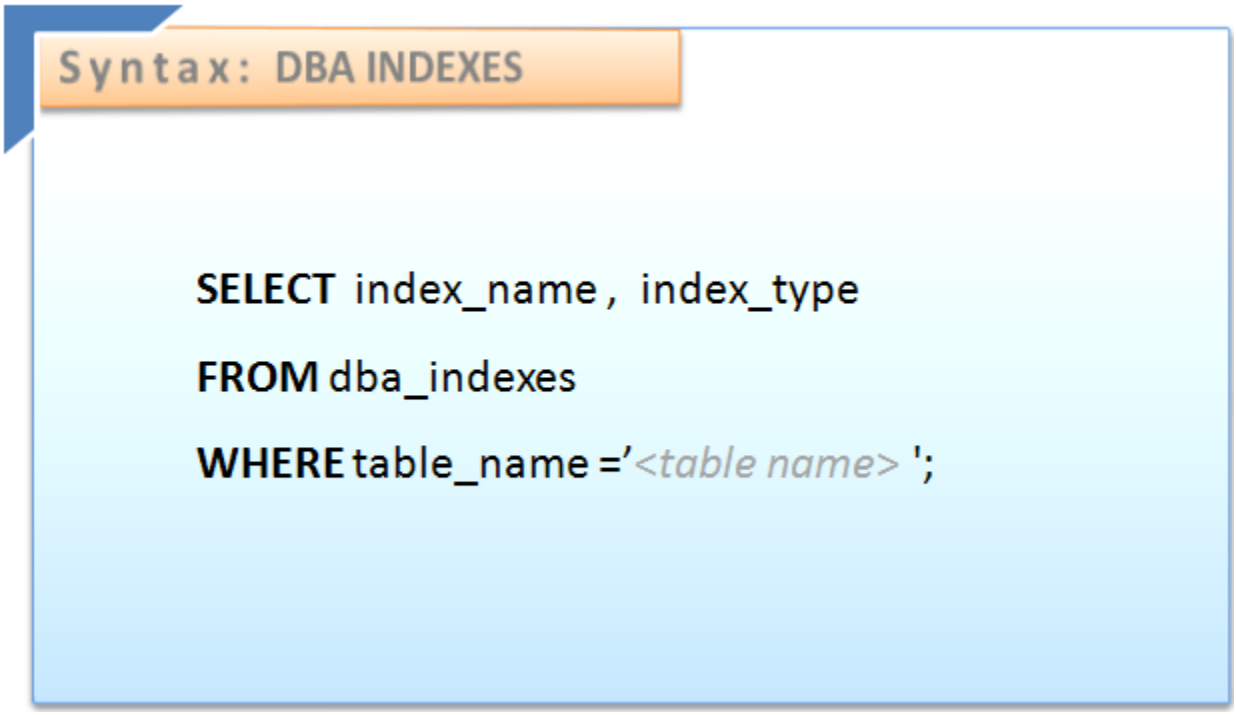
System tables for created index information:

User_indexes:-

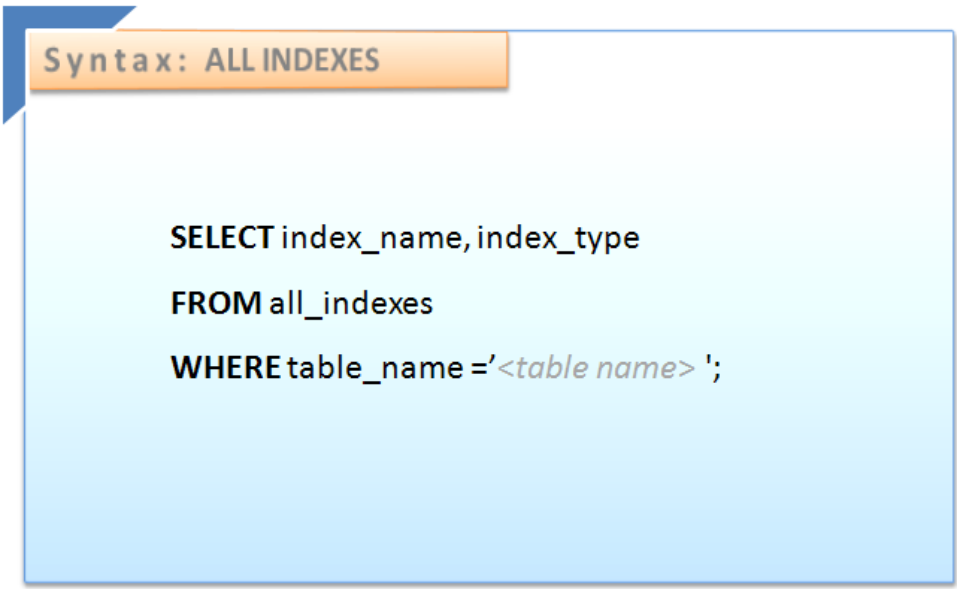
Syntax: USER INDEXES

```
SELECT index_name, index_type
FROM user_indexes
WHERE table_name ='<table name> ';
```

Dba_indexes:-

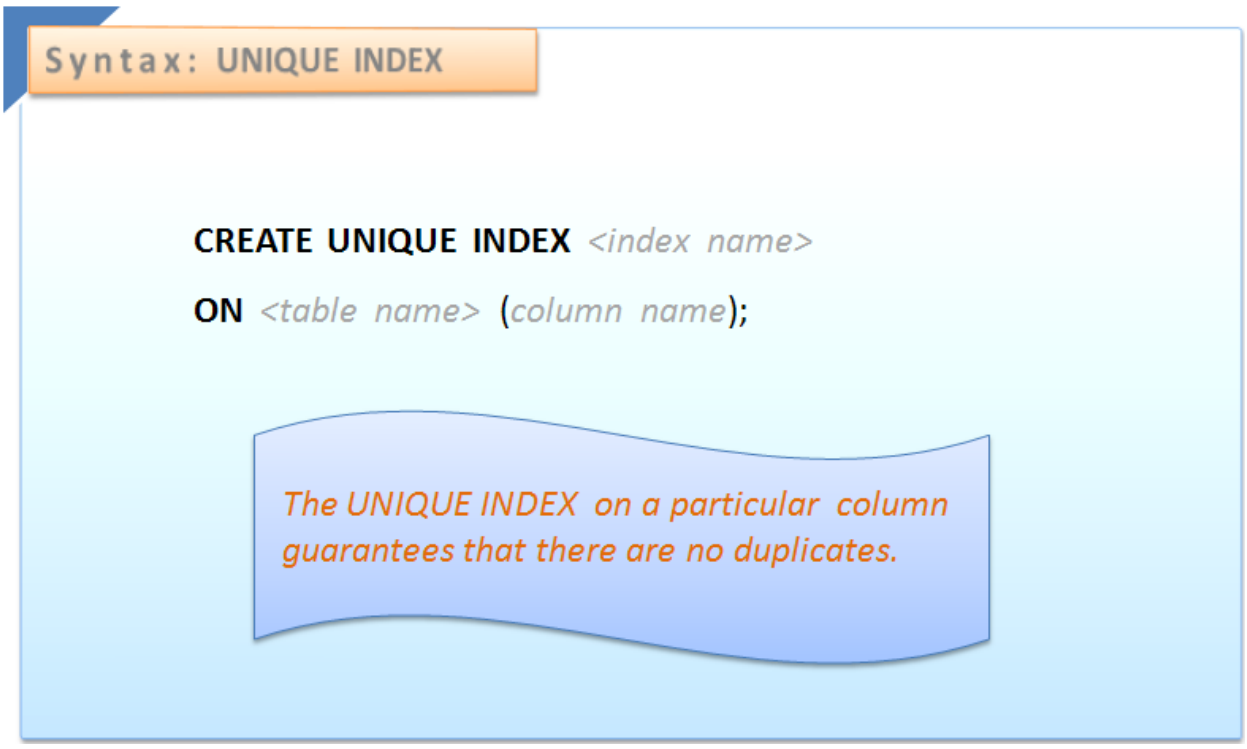


All_indexes:-



Unique Index:

- Unique index is also a part of indexes which guarantee that no two rows of a table have duplicate values in the columns that define the index.
- It will not allow duplicate values.
- Whenever we create unique index, internally unique constraint will get created and similarly when unique constraint is created unique index is created.



Example:-

```
SQL> CREATE UNIQUE INDEX na_uniq_deptno ON dept (deptno);

-- Index created.
```

Creating UNIQUE INDEXon multiple columns.

```
SQL> CREATE UNIQUE INDEX na_uniq_deptno ON dept (dname, loc);

--index created
```

Note: Usage of this index is based on optimizer. Internally during the execution, when ever required
Index gets applied automatically. We will not be able to observe the application of index as it gets
applied internally. Please refer to the performance tuning article.

Example:-

Whenever we create unique index, system will internally create unique constraint on the table column.

```
SQL> CREATE UNIQUE INDEX na_loc_idx ON dept(loc);
```

```
SQL>SELECT *FROM dept;
```

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

From above records we can see that there is record already available with LOC as 'CHICAGO'. Now if we try to insert same value (CHICAGO) into dept table in LOC column.

```
SQL> INSERT INTO dept VALUES (50,'TESTING','CHICAGO');

 Output-

* ERROR at line 1:
ORA-00001: UNIQUE CONSTRAINT (APPS.NA_IDX) violated
```

Note: System will raise an error because we had created unique index in that particular column (LOC)

For dropping index

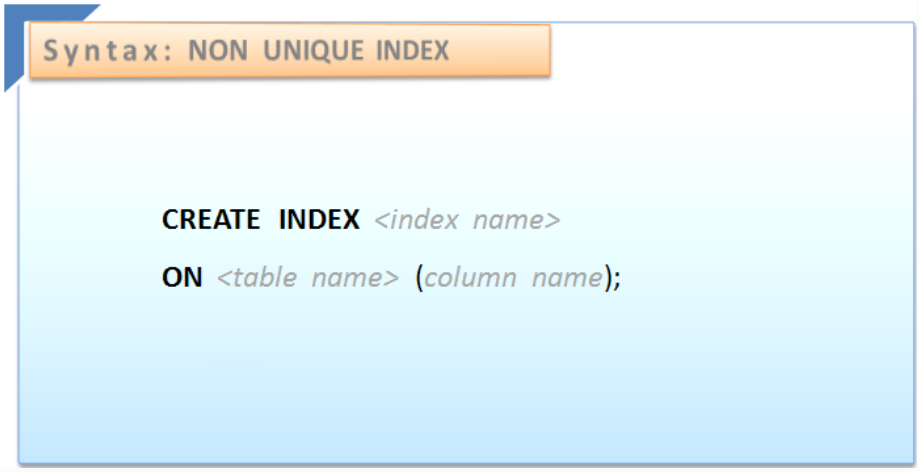
Example:-

```
SQL> DROP INDEX idx7;

-- Index dropped
```

Non Unique Indexes:

- Non unique index does not impose any restrictions on column values.
- NON Unique index- indexes duplicate values.



Example:-

```
SQL>CREATE INDEX na_non_job ON emp (job);

 --index created
```

So when ever we use this index column (JOB) in “where” clause, index (na_non_job) will automatically
activated.

Example:-

```
SQL>SELECT *FROM emp WHERE job='CLERK';
```

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7369	SMITH	CLERK	7902	2/20/1981	2400		20
7876	ADAMS	CLERK	7788	1/12/1983	1440		20
7900	JAMES	CLERK	7698	12/3/1981	1260		30
7934	MILLER	CLERK	7782	1/23/1982	1680		10

Note: Usage of this index is based on optimizer. Internally during the execution, when ever required
Index gets applied automatically. We will not be able to observe the application of index as it gets
applied internally. Please refer to the performance tuning article.

For dropping index

Example:-

```
SQL> DROP INDEX idx1;

-- Index dropped
```

Composite Index:

- Composite index is an index created on multiple columns (It allows maximum 32 columns).
- Columns in a composite index can appear in any order and need not be adjacent columns of the table.
- It will not create any constraints like unique.

Example:-

```
SQL> CREATE INDEX na_comp_idx ON emp(hiredate, sal)

-- Index created.
```

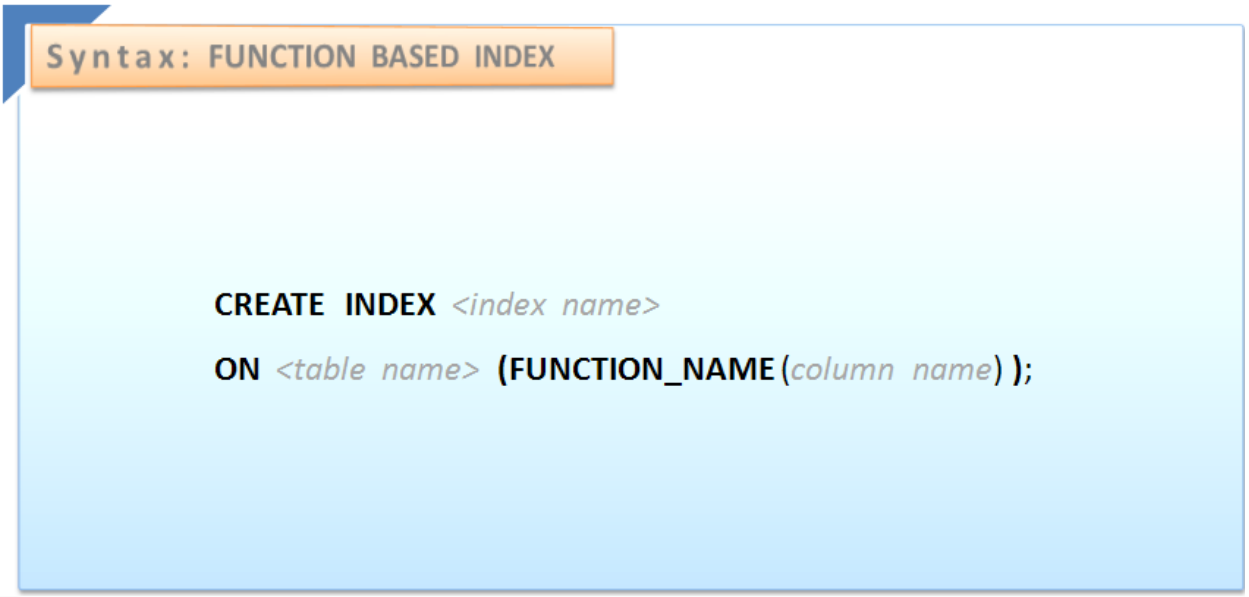
Example:-

SQL> **DROP INDEX** idx1; -- *Index dropped*

Note: Usage of this index is based on optimizer. Internally during the execution, when ever required
Index gets applied automatically. We will not be able to observe the application of index as it gets
applied internally. Please refer to the performance tuning article.

Function Based Index:

- Function based index is a index which is based on function instead of a column.
- We use Function based index to improve performance whenever we have functions used in the SELECT and WHERE clause.



Example:-

SQL> **CREATE INDEX** na_fbi_job **ON** emp (<span **STYLE**="background-color: yellow;">**UPPER** (job));

Now data retrieved from the below query will be much faster as we have function index available.

SQL> **SELECT** ***FROM** emp **WHERE UPPER** (job) =’MANAGER’;

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7566	JONES	MANAGER	7839	4/2/1981	3075		20
7698	BLAKE	MANAGER	7839	5/1/1981	2950		30
7782	CLARK	MANAGER	7839	6/9/1981	2550		10

We can not create index by using more than one function.

Note:

Usage of this index is based on optimizer. Internally during the execution, when ever required Index gets applied automatically. We will not be able to observe the application of index as it gets applied internally. Please refer to the performance tuning article.

For dropping index:

Example:-

```
SQL> DROP INDEX idx4;

-- Index dropped</code>
```

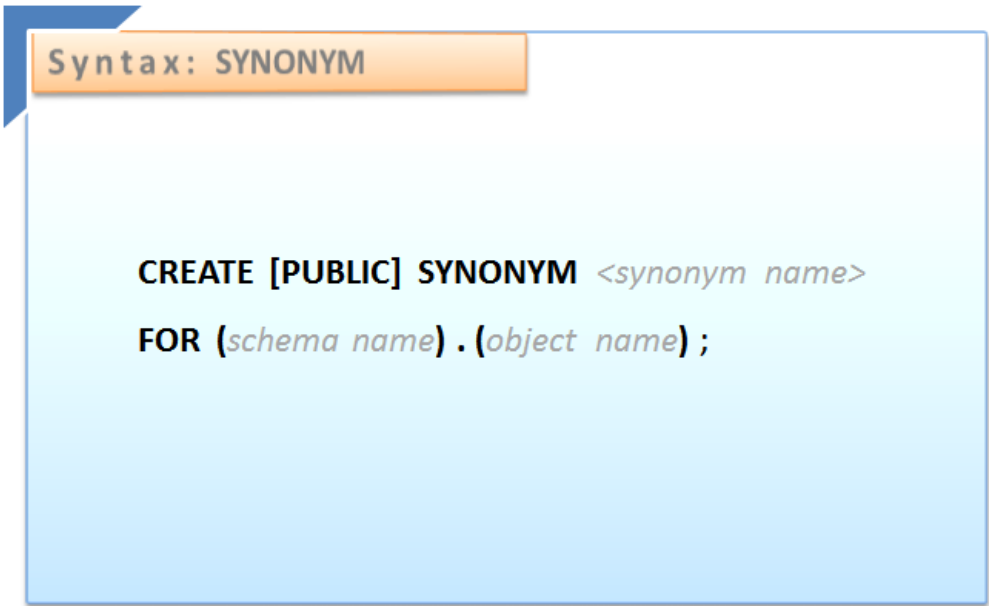
Chapter 19: Synonym

Synonym:

A synonym is a database object, which is used as an alias for a table, view or sequence.

- A Synonym acts as an alternative name for an existing schema object.
- Synonym basically allows us to create a pointer to an object that exists in different schema.
- By using a Synonym, we can avoid the entry of the schema name, when referencing objects that belong together schema.

PUBLIC keyword is optional, we specify PUBLIC keyword if we want to create PUBLIC synonym.



Types:-

1) Private
2) Public

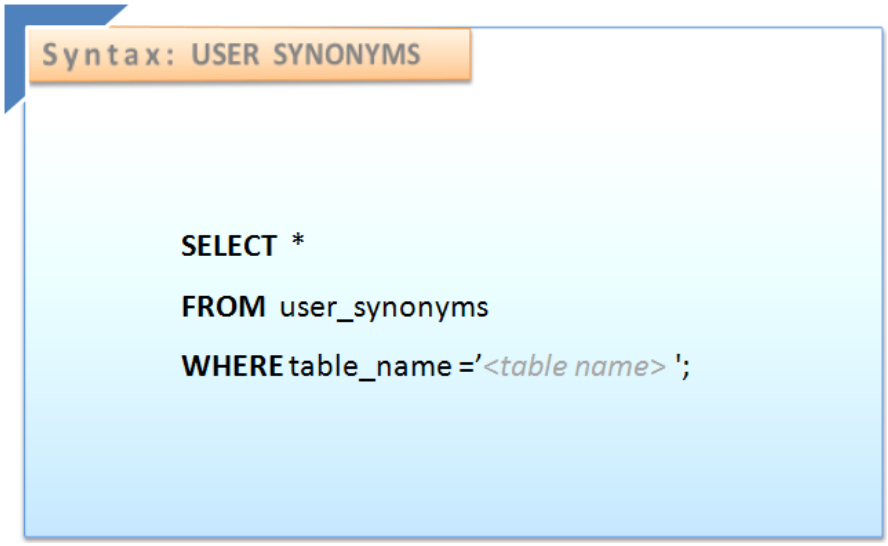
- Private synonym is available to the particular user who creates.
- Public synonym is created by DBA which is available to all the users.

Advantages:-

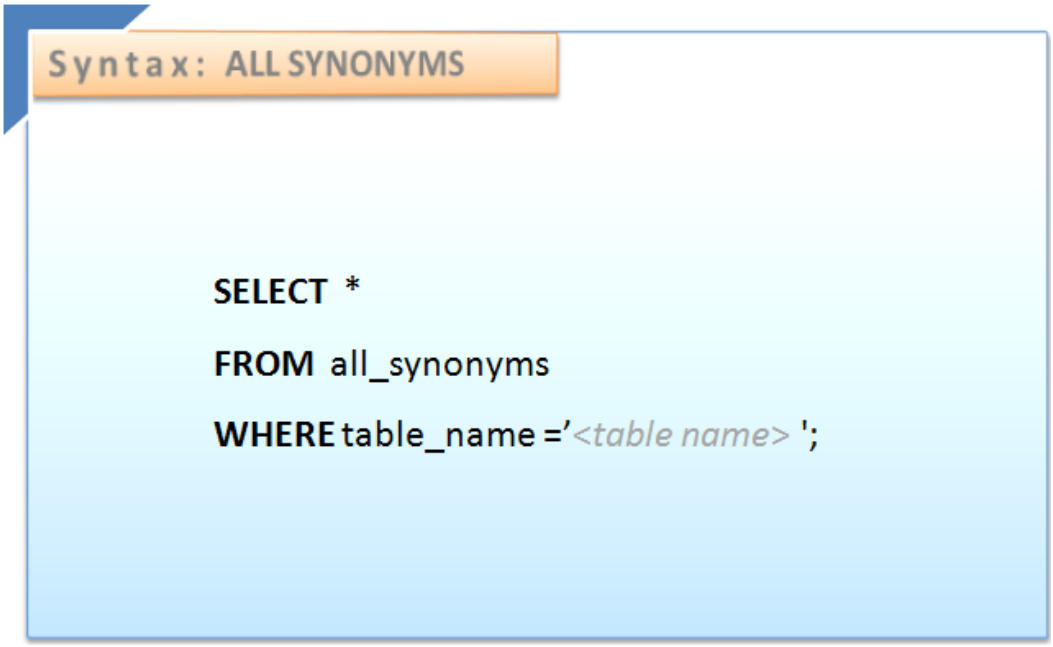
- Hide the name and owner of the object.
- Provides location transparency for remote objects of a distributed database.

System tables for viewing Synonyms:-

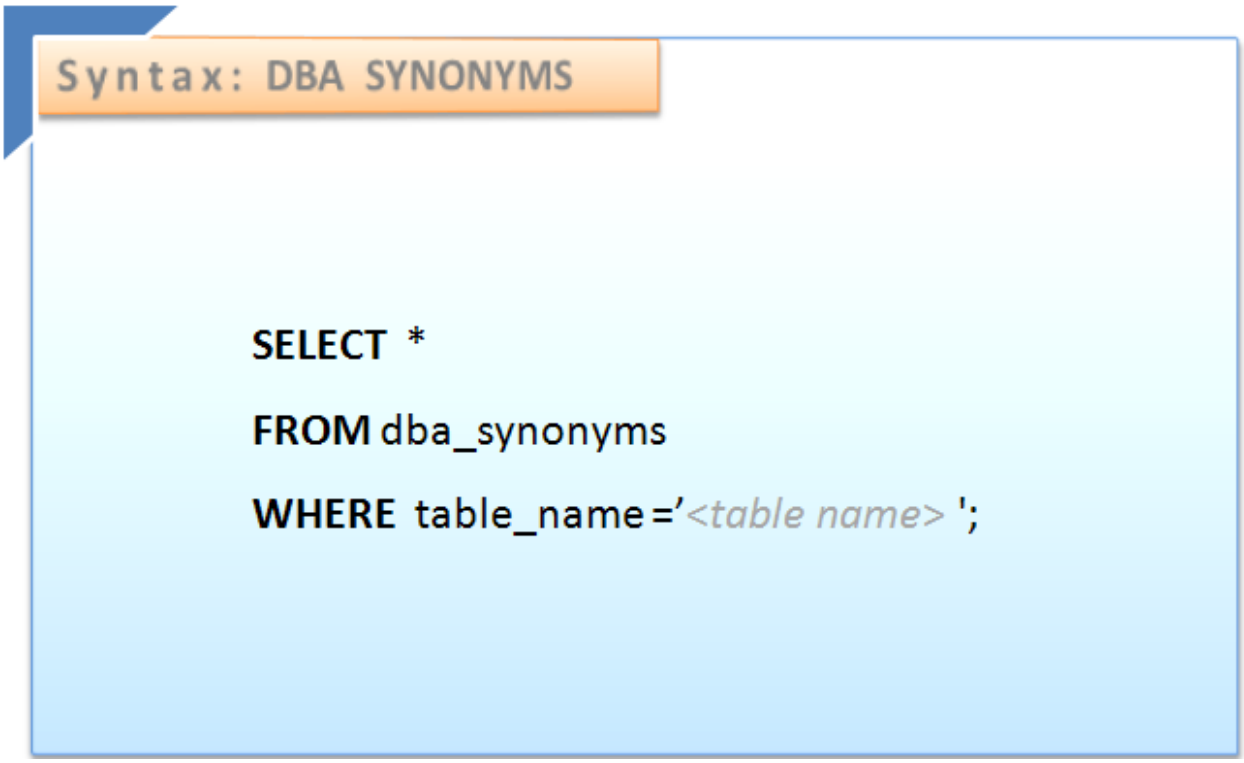
user_synonyms:-



all_synonyms:-



dba_synonyms:-



Private Synonym:

- **Private synonym** is a synonym that is available to the particular user who creates it.
- To create a private synonym in our own schema, we must have the CREATE SYNONYM system privilege.
- Private synonym is specific to the user.

To demonstrate private synonym lets create a database user first in the database.

```
SQL> CREATE USER user1 IDENTIFIED BY user1;</code>
-- User created.
SQL> GRANT CONNECT TO user1;
-- GRANT succeeded.
```

Once the 'user1' is created then login database using user1.

```
SQL> conn user1/user1@vis;
```

And now from 'user1' if we wanted to access EMP table that belongs to SCOTT schema then we should
refer to EMP using 'schemaname.tablename' (SCOTT.EMP)

```
SQL> SELECT * FROM scott.emp;
```

Output:

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7369	SMITH	CLERK	7902	12/17/1980	800		20
7499	ALLEN	SALESMAN	7698	2/20/1981	1600	300	30
7521	WARD	SALESMAN	7698	2/22/1981	1250	500	30
7566	JONES	MANAGER	7839	4/2/1981	2975		20
7654	MARTIN	SALESMAN	7698	9/28/1981	1250	1400	30
7698	BLAKE	MANAGER	7839	5/1/1981	2850		30
7782	CLARK	MANAGER	7839	6/9/1981	2450		10
7788	SCOTT	ANALYST	7566	12/9/1982	3000		20
7839	KING	PRESIDENT		11/17/1981	5000		10
7844	TURNER	SALESMAN	7698	9/8/1981	1500		30
7876	ADAMS	CLERK	7788	1/12/1983	1100		20
7900	JAMES	CLERK	7698	12/3/1981	950		30
7902	FORD	ANALYST	7566	12/3/1981	3000		20
7934	MILLER	CLERK	7782	1/23/1982	1300		10

– To access EMP table, every time we should refer using SCOTT schema as SCOTT is the owner of the table.

From ‘user1’ if we wanted to hide the EMP table then we should create synonym for EMP table in ‘user1’.

/ Inside User1*/*

SQL> **CREATE** SYNONYM na_emp **FOR** scott.emp;

-- Synonym created.
*/*Once the synonym is created, we can use this synonym just like a table in select statement*/*

SQL> **SELECT** * **FROM** na_emp;

Output:

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7369	SMITH	CLERK	7902	12/17/1980	800		20
7499	ALLEN	SALESMAN	7698	2/20/1981	1600	300	30
7521	WARD	SALESMAN	7698	2/22/1981	1250	500	30
7566	JONES	MANAGER	7839	4/2/1981	2975		20
7654	MARTIN	SALESMAN	7698	9/28/1981	1250	1400	30
7698	BLAKE	MANAGER	7839	5/1/1981	2850		30
7782	CLARK	MANAGER	7839	6/9/1981	2450		10
7788	SCOTT	ANALYST	7566	12/9/1982	3000		20
7839	KING	PRESIDENT		11/17/1981	5000		10
7844	TURNER	SALESMAN	7698	9/8/1981	1500		30
7876	ADAMS	CLERK	7788	1/12/1983	1100		20
7900	JAMES	CLERK	7698	12/3/1981	950		30
7902	FORD	ANALYST	7566	12/3/1981	3000		20
7934	MILLER	CLERK	7782	1/23/1982	1300		10

– NA_EMP is the synonym that hides actual EMP table. From now on ‘user1’ can directly execute ‘SELECT * FROM na_emp;’ without referring to SCOTT.EMP table.

Drop synonym:-

SQL> **DROP** SYNONYM na_emp;

-- Synonym dropped.

Public Synonym:

- **Public Synonym** is a synonym which can be accessed by all the all the users in the database. These synonyms are generally created by Database administrators.
- To create public synonym, we should specify PUBLIC keyword in the syntax.
- Public synonyms are owned by special schema in the oracle database called PUBLIC. As mentioned earlier, public
synonyms can be referenced by all users in the database.

To create public synonym first we try to connect to ‘system’ or ‘sysdba’.

*/*connect to ‘system’ or ‘sysdba’ user who has the full admin access to CREATE public synonym.*/*

SQL>conn system/manager

Then create a public synonym for EMP table which is in ‘scott’ user.

SQL> **CREATE** PUBLIC SYNONYM na_pub_emp **FOR** scott.emp;

-- Synonym created.

Provide necessary privileges to ‘user1’ to access synonym ‘na_pub_emp’.SQL> GRANT ALL on na_pub_emp to user1;

SQL> **GRANT ALL ON** na_pub_emp **TO** user1;
-- grant succeeded.

Now let’s try to connect to user1 and access the public synonym.

SQL> conn user1/user1@vis; SQL> **SELECT** * **FROM** na_pub_emp;

Note: NA_PUB_EMP is the public synonym that hides actual EMP table.

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7369	SMITH	CLERK	7902	12/17/1980	800		20
7499	ALLEN	SALESMAN	7698	2/20/1981	1600	300	30
7521	WARD	SALESMAN	7698	2/22/1981	1250	500	30
7566	JONES	MANAGER	7839	4/2/1981	2975		20
7654	MARTIN	SALESMAN	7698	9/28/1981	1250	1400	30
7698	BLAKE	MANAGER	7839	5/1/1981	2850		30
7782	CLARK	MANAGER	7839	6/9/1981	2450		10
7788	SCOTT	ANALYST	7566	12/9/1982	3000		20
7839	KING	PRESIDENT		11/17/1981	5000		10
7844	TURNER	SALESMAN	7698	9/8/1981	1500		30
7876	ADAMS	CLERK	7788	1/12/1983	1100		20
7900	JAMES	CLERK	7698	12/3/1981	950		30
7902	FORD	ANALYST	7566	12/3/1981	3000		20
7934	MILLER	CLERK	7782	1/23/1982	1300		10

Just like 'user1', similarly all other users in the database can access the 'NA_PUB_EMP' synonym as it is a public synonym.

Drop the public synonym:-

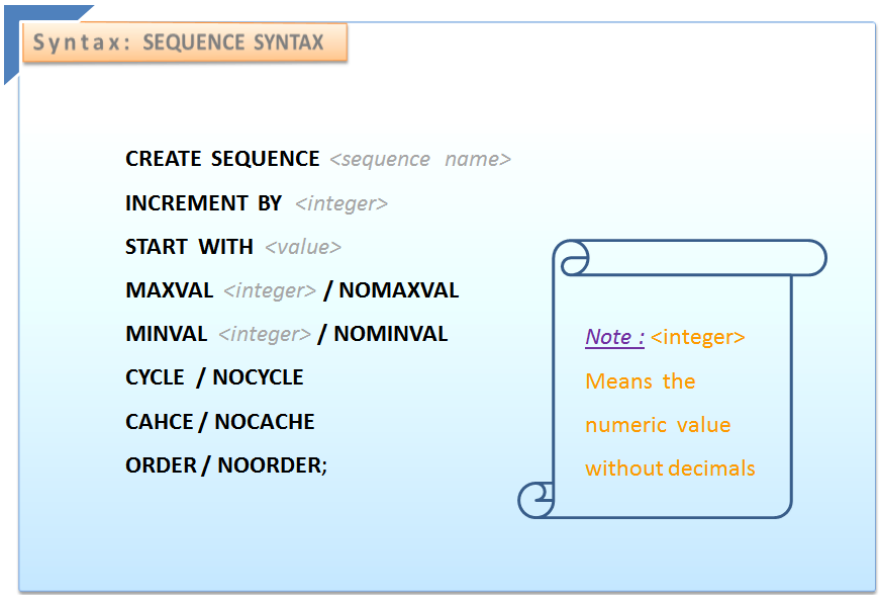
```
SQL> DROP PUBLIC SYNONYM emp_syn;

-- Synonym dropped
```

Chapter 20: Sequence

What Is Sequence?

- Sequence is a Schema or Database Object that can generate UNIQUE Sequential Values.
- Can be used to generate PRIMARY KEY values automatically.



By default the sequence starts with 1, increments by 1 with minvalue of 1 and with nocycle, no cache.

SEQUENCE can be either incremented or decremented Sequence.
INCREMENT BY: -

- Specifies the interval between the sequence numbers.
- Value can be positive or negative. But cannot be zero
- If the value is positive it is Incremented sequence else it is decremented sequence
- If omitted defaults to increment1 by 1.

MINVAL clause:-

- Specifies the sequences minimum value.

NOMINVAL clause

- Specify to indicate a minimum value of 1 for an ascending sequence or -10^{26} for a descending sequence. This is the default.

MAXVAL clause:-

- Specifies the maximum value that can be generated.

NOMAXVAL clause:-

- Specify to indicate a maximum value of 10^{27} for an ascending sequence or -1 for a descending sequence. This is the default.

CYCLE clause:-

- Specifies the sequence, will continue to generate values after reaching either maximum 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 sequence are not pre-allocated.

ORDER clause:-

- Guarantees the sequence number to be generated in the order of request.

NOORDER clause:-

- Does not guarantee the sequence number with order.

Example:-

```
SQL> CREATE SEQUENCE sampleseq
INCREMENT BY 1
START WITH 0
MINVALUE 0
MAXVALUE 10
```

```
NOCACHE
NOCYCLE;

Ouput:
--sequence created
```

Using A Sequence:

How to use a sequence ?

After the creation of sequence, it generates sequential numbers for use in our tables. Reference the sequence values by using the NEXTVAL and CURRVAL Pseudo columns.

Pseudo Column-

It behaves like table column, but it is not actually stored in a table.

The available pseudo columns for sequence are:-

- CURRVAL
- NEXTVAL

NEXTVAL-

- When we call 'sequencename.NEXTVAL' , a new sequence number is generated and displayed.
- If we run it again then next value will be generated.

CURRVAL-

It is used to refer to a sequence number that the current session has just generated Or the current value that sequence holds.

- 'sequencename.CURRVAL' should be used to fetch last value generated.
- Next value must be run first to generate a sequence number in the current user's session before CURRVAL can be called.

To demonstrate the sequence 'sampleseq' created in the previous lesson, lets first create a table and run some insert statements.

```
SQL>CREATE TABLE seq_tab (empno NUMBER(10),ename varchar2(10));

-- Table created
```

Let's INSERT a record INTO 'seq_tab' table.

Sampleseq – it is the name of the sequence which we have created in the previous lesson.
Nextval – it is pseudo column.

```
SQL>INSERT INTO seq_tab

VALUES (sampleseq.NEXTVAL,'CAMERON');

-- 1 row inserted
```

```
SQL>SELECT *FROM seq_tab;
```

EMPNO	ENAME	DEPT
2	CAMERON	2

– Now lets check the CURRVAL

```
SQL>SELECT <span STYLE="background-color: yellow;"> sampleseq.CURRVAL </span> FROM dual;
```

CURVAL
2

Now, if we again INSERT the record INTO same table then -

```
SQL> INSERT INTO seq_tab VALUES (sampleseq.NEXTVAL,'JOHN');

-- 1 row inserted
```

```
SQL>SELECT *FROM seq_tab;
```

EMPNO	ENAME	DEPT
2	CAMERON	2
3	JOHN	3

– Now lets check the CURRVAL

```
SQL> SELECT sampleseq.CURRVAL FROM dual;
```

CURVAL
3

In the above example we can see that sequence number is automatically generated for empno column.

Altering And Dropping The Sequence:

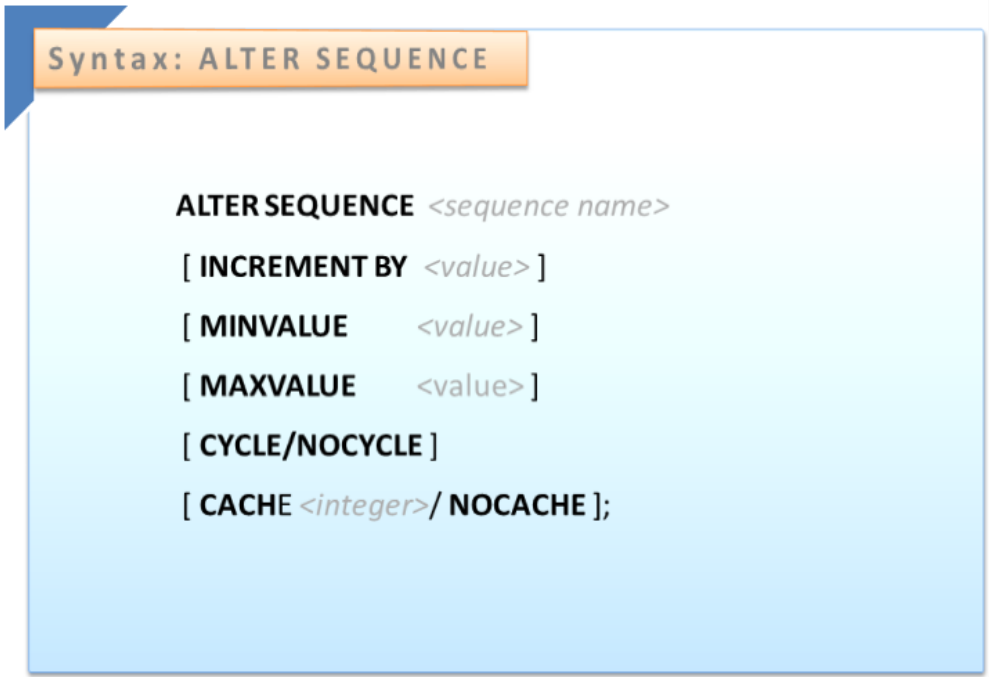
We can alter the sequence only for specific options.

- Set or eliminate minvalue or maxvalue.
- Change the increment value.
- Change the number of cached sequence numbers

Guidelines for altering the sequence or modifying the sequence

- We must be the owner of the sequence or should have privileges to ALTER the sequence.
- Only future sequence number is affected by ALTER SEQUENCE statement.

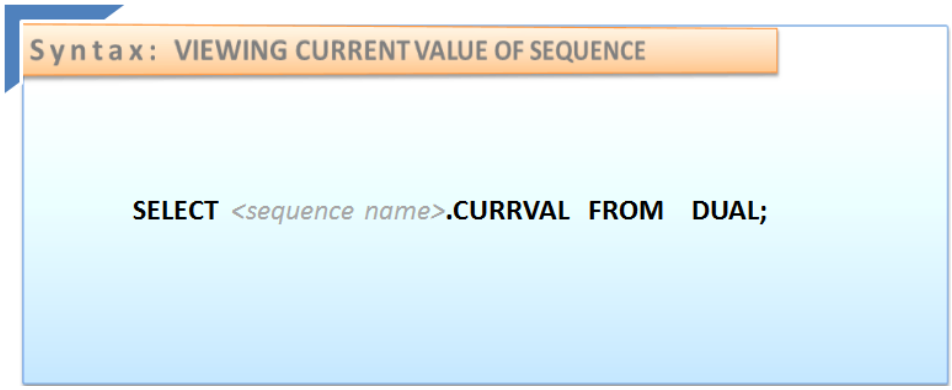
The START WITH option cannot be changed using ALTER SEQUENCE. The sequence must be dropped and re-created in order to restart the sequence at different number



Example:-

```
SQL>ALTER SEQUENCE sampleseq</code>
  MAXVAL 10
  CACHE
  NOCYCLE;
```

Viewing the Current value of sequence



Example:-

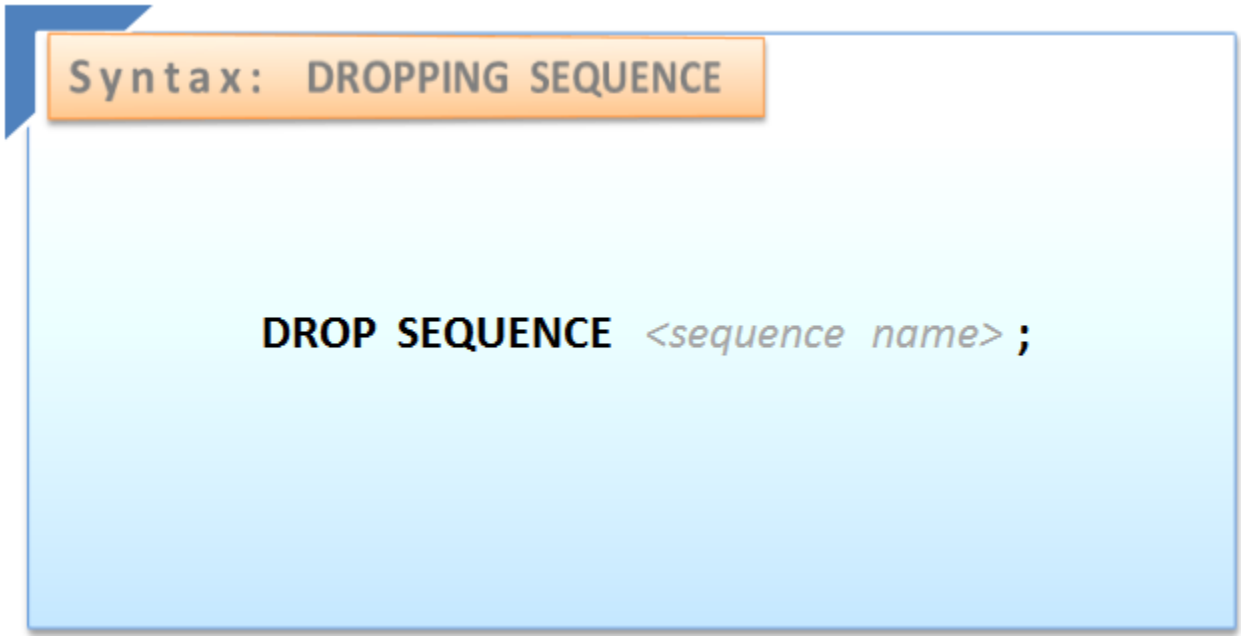
```
SQL>SELECT sampleseq.CURRVAL FROM DUAL;

-- Output
```

CURRVAL
7

DROP SEQUENCE:-

To remove the sequence from data dictionary, use the DROP SEQUENCE statement. We must be the owner of the sequence or should have DROP ANY SEQUENCE privilege to remove it.



Example:-

```
SQL>DROP SEQUENCE sampleseq ;

--sequence dropped
```

VIEWING THE SEQUENCES CREATED:-

The following tables lists all the Sequences which are created by the users.

- DBA_SEQUENCES
- ALL_SEQUENCES
- USER_SEQUENCES

Example:-

```
SELECT * FROM USER_SEQUENCES;
```

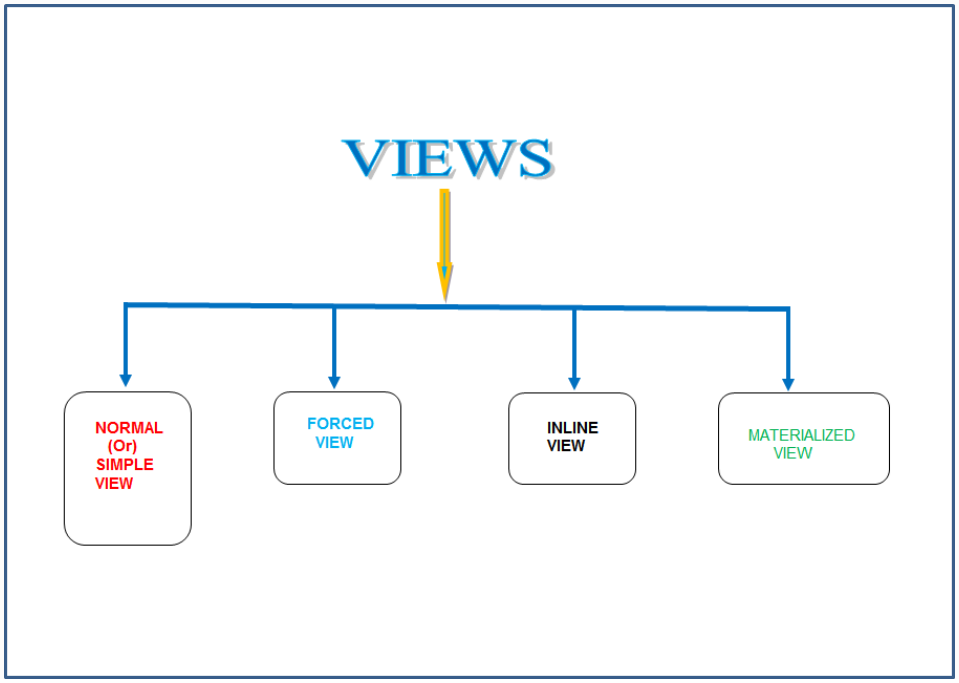
Chapter 21: Views

View

- A view is a logical representation of the data.
- View does not have storage of its own. (i.e. View does not hold any data)
- View can be used just like a table in SELECT statement.
- A view can be considered as a virtual table.
- A view is stored as a SELECT statement in the database. DML operations on a view like INSERT, UPDATE,
  DELETE affects the data in the original table upon which the view is based.

For example, if we wanted a sub ordinate to access EMP table without the details of salaries then we can
create a view for the sub ordinate with only required columns.

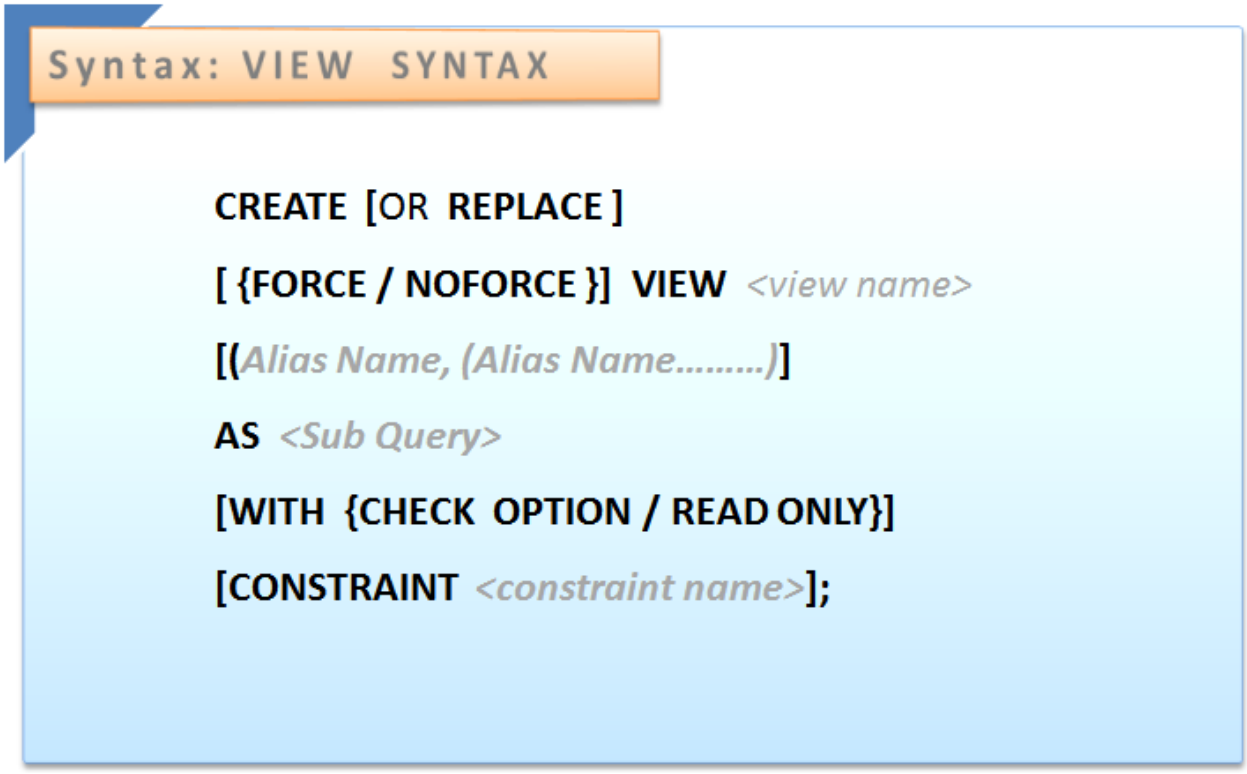
VIEW TYPES :- We have four types of views:
1) Normal / Simple View
2) Forced View
3) Inline View
4) Materialized View.



WHY VIEWS?

- Provides additional level of security by restricting access to a predetermined set of rows and/or columns of a table.
- Hide the data complexity.
- Simplify commands for the user.

The following image shows the syntax of View



The Data Dictionary Table's for viewing the Views created are:

1) USER_VIEWS

Example:-

SQL> **SELECT** * **FROM** USER_VIEWS;

2) ALL_VIEWS

SQL> **SELECT** view_type, text **FROM** all_views **WHERE VIEW** _name =< **VIEW** _name>;

3) DBA_VIEWS

SQL> **SELECT** view_type, text **FROM** dba_views **WHERE VIEW** _name = <**VIEW** _name>

Simple View:

Simple view is the view that is created based on a single table.

A Simple View is the one that:

- Derives data from only one table.
- Contains no functions or groups of data.
- Can perform DML operations through the view.

Examples:-

```
SQL> SELECT * FROM emp;
```

Now let's try to create a view based on EMP table as follows.

```
SQL> CREATE OR REPLACE VIEW na_emp_v AS SELECT * FROM emp;

VIEW created
SQL> SELECT * FROM na_emp_v;
```

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7369	SMITH	CLERK	7902	2/20/1981	2000		20
7499	ALLEN	SALESMAN	7698	2/20/1981	1800	300	30
7521	WARD	SALESMAN	7698	2/22/1981	1350	500	30
7566	JONES	MANAGER	7839	4/2/1981	3075		20
7654	MARTIN	SALESMAN	7698	9/28/1981	1350	1400	30
7698	BLAKE	MANAGER	7839	5/1/1981	2950		30
7782	CLARK	MANAGER	7839	6/9/1981	2550		10
7788	SCOTT	ANALYST	7566	12/9/1982	3100		20
7839	KING	PRESIDENT		11/17/1981	5100		10
7844	TURNER	SALESMAN	7698	9/8/1981	1700	0	30
7876	ADAMS	CLERK	7788	1/12/1983	1200		20
7900	JAMES	CLERK	7698	12/3/1981	1050		30
7902	FORD	ANALYST	7566	12/3/1981	3100		20
7934	MILLER	CLERK	7782	1/23/1982	1400		10

Create a view with specified columns:-

SQL> **CREATE OR REPLACE VIEW** <span **STYLE**="background-color: yellow;"> na_emp_v (empno, enan
emp;
--View created.

SQL> **SELECT** * **FROM** na_emp_v;

EMPNO	ENAME	SAL	DEPTNO
7369	SMITH	2000	20
7499	ALLEN	1800	30
7521	WARD	1350	30
7566	JONES	3075	20
7654	MARTIN	1350	30
7698	BLAKE	2950	30
7782	CLARK	2550	10
7788	SCOTT	3100	20
7839	KING	5100	10
7844	TURNER	1700	30
7876	ADAMS	1200	20
7900	JAMES	1050	30
7902	FORD	3100	20
7934	MILLER	1400	10

Dropping a view:-

Syntax:-

```
SQL> DROP VIEW view_name;
```

Example:-

```
SQL> DROP VIEW na_emp_v;
VIEW dropped.
```

Complex Views:

Complex view is the view that is created on multiple tables.

- We cannot perform DML operations directly on complex view. But by using **instead of triggers** We can do DML operations on complex views.
- A Complex view is one that:
- Derives data from many tables.
- Contains functions or groups of data.
- Does not allow DML operations directly.

Examples:-

We can use DEPT and EMP tables for creating complex views.

```
SQL> SELECT * FROM dept;
SQL> SELECT * FROM emp;
```

```
SQL> CREATE OR REPALCE VIEW na_emp_v AS
  SELECT e.empno,
              e.ename,
                    e.sal,
                    e.deptno,
                    d.dname,
                    d.loc FROM emp e,dept d
  WHERE e.deptno = d.deptno;

VIEW created.
```

Once the view is created then we can query the data from view.

```
SQL> SELECT * FROM na_emp_v;
```

Creating a complex view by using group functions:-

```
SQL> CREATE OR REPLACE VIEW na_emp_v AS
SELECT deptno, COUNT (*) total
FROM emp
GROUP BY deptno;

VIEW created.
```

Query data from view

```
SQL> SELECT * FROM na_emp_v;
```

Dropping a view:-

Syntax:-

```
SQL> DROP VIEW <view_name>
```

Example to drop a View:-

```
SQL> DROP VIEW na_emp_v;

VIEW dropped.
```

DML Operations on Views:

Rules for DML's on views:-

For delete: -

- It is not possible to delete when the view is derived from more than one database table.

For update: -

- It is not possible to update when the view is derived from more than one database table.
- It is not possible to update the columns which are derived from arithmetic expression andfunction from pseudo columns.

For insertion:-

- It is not possible to insert when the view is derived from more than one database table.
- It is not possible to insert into the columns which are derived from arithmetic expression andfunction from pseudo columns.
- It is not possible to insert the data into the view if the mandatory/primary key columns of thebase tables are missing in the view definition.

DML operations on simple views:-

For better understanding lets create a sample table first.

```
SQL> CREATE TABLE na_dept (deptno NUMBER (10), dname VARCHAR2(10), loc VARCHAR2(10)); --table create
```

Once the table is created, let's create a simple view on top of it.

```
SQL> CREATE VIEW na_dept_view AS SELECT *FROM na_dept;

-- View created.
```

/*NA_DEPT_VIEW don't have any records as this is created on top of a empty table*/

Insert record into NA_DEPT_VIEW to know whether DML operations will reflect the base table or not.

```
SQL>INSERT INTO na_dept_view VALUES (10,'ACCOUNTING','ENGLAND');

--1 row inserted
```

Now we can see that records are inserted into the table.

/*Data in Base Table*/

```
SQL>SELECT *FROM na_dept;
```

DEPTNO	DNAME	LOC
10	ACCOUNTING	ENGLAND

/*Data in View*/

```
SQL>SELECT * FROM na_dept_view;
```

DEPTNO	DNAME	LOC
10	ACCOUNTING	ENGLAND

*Similarly we can perform other DML operations like update and delete on the view which internally affects the base table.

DML operations on complex views:-

Complex view is a view based on multiple tables.

Below is the complex view that is created on two tables EMP and DEPT

```
SQL>CREATE VIEW na_complex_v AS
SELECT e.empno, e.ename, e.sal ,d.deptno,d.dname, d.loc
FROM emp e, dept d
WHERE e.deptno = d.deptno;

-- View created.
```

When we try to insert record into the complex view system throws an error, as there are multiple tables involved.

```
SQL>INSERT INTO na_complex_v VALUES (7625,'BLAKE', 12000,20,'TESTING','CHICAGO');
```

ERROR at line 1:

ORA-01776: cannot modify more than one base table through a join view

It is clear that we cannot perform any DML operations on COMPLEX VIEWS, but we can overcome this

by using INSTEAD OF TRIGGERS.

INSTEAD OF TRIGGERS:-

- Instead of triggers can be used only with views.
- Effective for views in which there are multiple tables involved.

Let's create INSERTED OF TRIGGER on 'na_complex_v'. Inside trigger, we will write a plsql code to insert data into EMP and DEPT separately as our view is based on these two tables.

```
SQL>CREATE OR REPLACE TRIGGER na_instd_of_trig INSTEAD OF INSERT ON na_complex_v FOR EACH ROW B
DEPT (deptno,dname,loc) VALUES (:NEW.deptno, :NEW.dname, :NEW.LOC); END;
```

/* :new key word is used to refer to the values in INSERT statements. */

Once the trigger is created lets insert the data into complex view i.e. 'na_complex_v'

```
SQL>INSERT INTO na_complex_v VALUES (7625,'BLAKE', 12000,20,'TESTING','CHICAGO');
```

Output:

```
--1 row inserted
```

Insert statement will trigger the 'INSTEAD OF TRIGGER' which will insert the data into EMP

and DEPT tables separately.

/*Data in DEPT Table*/

```
SQL>SELECT *FROM dept;
```

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

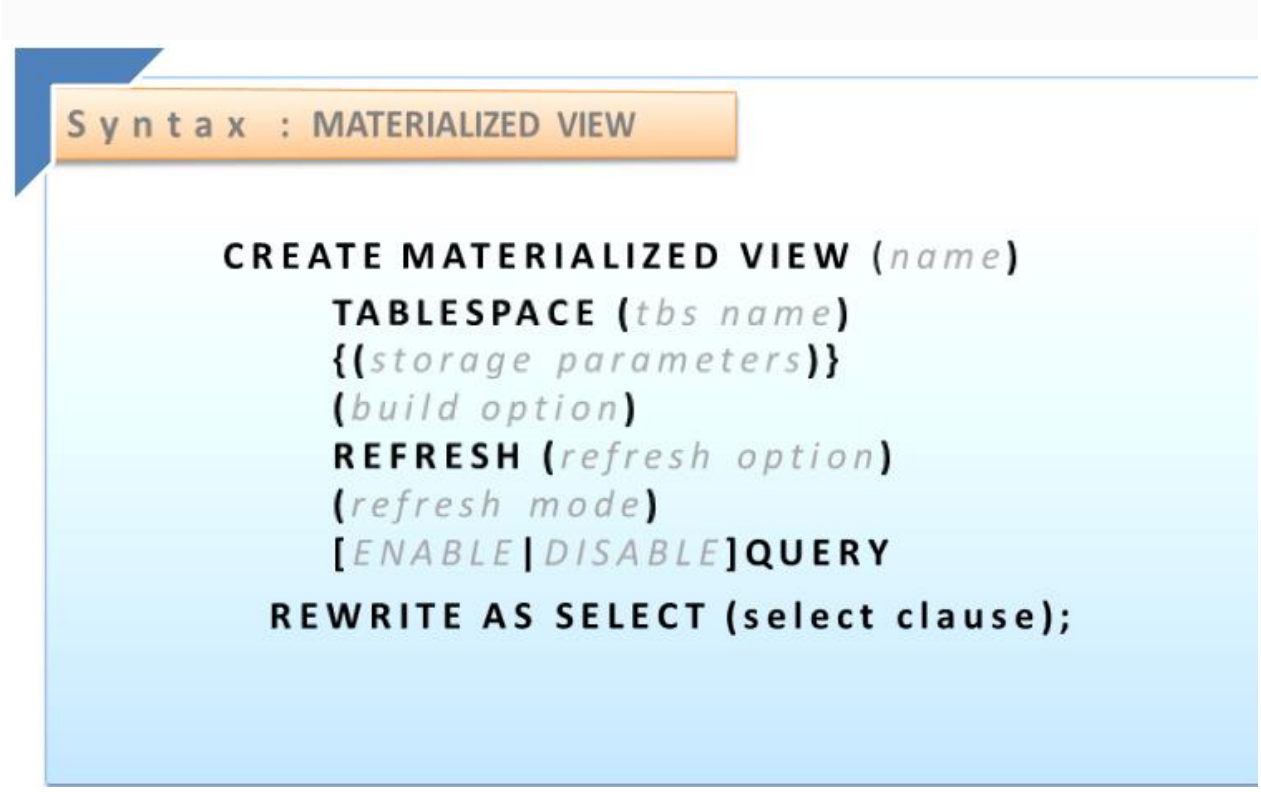
/*Data in EMP Table*/

```
SQL>SELECT * FROM EMP;
```

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7369	SMITH	CLERK	7902	12/17/1980	800		20
7499	ALLEN	SALESMAN	7698	2/20/1981	1600	300	30
7521	WARD	SALESMAN	7698	2/22/1981	1250	500	30
7566	JONES	MANAGER	7839	4/2/1981	2975		20
7654	MARTIN	SALESMAN	7698	9/28/1981	1250	1400	30
7698	BLAKE	MANAGER	7839	5/1/1981	2850		30
7782	CLARK	MANAGER	7839	6/9/1981	2450		10
7788	SCOTT	ANALYST	7566	12/9/1982	3000		20
7839	KING	PRESIDENT		11/17/1981	5000		10
7844	TURNER	SALESMAN	7698	9/8/1981	1500		30
7876	ADAMS	CLERK	7788	1/12/1983	1100		20
7900	JAMES	CLERK	7698	12/3/1981	950		30
7902	FORD	ANALYST	7566	12/3/1981	3000		20
7934	MILLER	CLERK	7782	1/23/1982	1300		10
7625	BLAKE				12000		

MATERIALIZED VIEW:

- A materialized view is a database object that contains the results of a query.
- Materialized View is a static view that holds data in it.
- Materialized view takes the snapshot of the data whenever it is created or refreshed.
- Materialized view does not support DML operations on it.
- To create a “materialized view “user should have permission in schema.
- It is used to maintain historic data.
- It is used for data analysis and reporting purpose.
- Materialized are mainly created to overcome the performance issues and to store historic data.



The determines when Materialized View is built:

- **BUILD IMMEDIATE:** view is built at creation time
- **BUILD DEFFERED:** view is built at a later time
- **ON PREBUILT TABLE:** use an existing table as view source
- Must set **QUERY_REWRITE_INTEGRITY** to **TRUSTED**

Refresh Options

-COMPLETE – totally refreshes the view

- Can be done at any time; can be time consuming

-FAST- incrementally applies data changes

- A materialized view log is required on each detail table
- Data changes are recorded in MV logs or direct loader logs
- Many other requirements must be met for fast refreshes

-FORCE –does a FAST refresh in favor of a COMPLETE

- The default refresh option

Refresh Modes-ON COMMIT –refreshes occur whenever a commit is performed on one of the view's underlying detail table(s).

- Available only with single table aggregate or join based views
- Keeps view data transactionally accurate
- Need to check alert log for view creation errors

ON DEMAND –Refreshes are initiated manually using one of the procedures in the DBMS_MVIEW package.

- Can be used with all types of materialized views
- Manual Refresh Procedures
- **DBMS_MVIEW.REFRESH**(,)
- **DBMS_MVIEW.REFRESH_ALL_MVIEWS()**

Example 1:

```
CREATE OR REPLACE MATERIALIZED VIEW na_mv_view
REFRESH FAST START WITH SYSDATE NEXT SYSDATE + 1
AS SELECT * FROM emp;
```

– In the above example all the options that are not specified are defaulted.

Example 2:

```
SQL> CREATE MATERIALIZED VIEW items_summary_mv ON PREBUILT TABLE
REFRESH FORCE AS
SELECT a.PRD_ID,
    a.SITE_ID,
    a.TYPE_CODE,
    a.CATEG_ID,
    SUM(a.GMS) GMS,
    SUM(a.NET_REV) NET_REV,
    SUM(a.BOLD_FEE) BOLD_FEE,
    SUM(a.BIN_PRICE) BIN_PRICE,
    SUM(a.GLRY_FEE) GLRY_FEE,
    SUM(a.QTY_SOLD) QTY_SOLD,
    COUNT(a.ITEM_ID) UNITS
FROM na_items a GROUP BY a.PRD_ID, a.SITE_ID, a.TYPE_CODE, a.CATEG_ID;
```

Difference Between Normal and Materialized View:

Normal View:-

- It is a stored select statement
- It is a virtual component
- It allows DESC, DML, SELECT on it
- It is stored permanently in “user_views” system table.
- It can be shared with other users
- DML on view are reflected in table and DML on table are reflected in view
- It is used to share “selected rows and columns” with other rows
- It is used for reporting purpose
- It will improve the performance while manipulating or retrieving data through views

Materialized view:-

- It is a static view
- It holds data in it
- It will not support DML on it
- DML on table will not be reflected in view
- To create it “create materialized view “permission is required.
- It is used to maintain historic data.
- It is used for data analysis and reporting purpose.
- It is same as SNAP SHOT (defined by DBA only)

Force View:

A view can be created even if the defining query of the view cannot be executed. We call such a view
as view with errors.
For example, if a view refers to a non-existent table or an invalid column of an existing table or if the
owner of the view does not have the required privileges, then the view can still be created and entered
into the data dictionary.
We can create such views (i.e. view with errors) by using the FORCE option in
the CREATE VIEW command:

Example: – In this example we are trying to create a view using a table that does not exist in the database.
/* dummy_table is not available in the database*/

```
CREATE OR REPLACE FORCE VIEW na_view
AS SELECT * FROM dummy_table;
```

Output:-
Warning: View created with compilation errors.

Specify FORCE if we want to create the view regardless of whether the base tables of the view or the
referenced object types exist or the owner of the schema containing the view has privileges on them.
The advantage of force view is that in future if view script becomes valid, we can start using the view.