**Database Concepts**

**Text :**

Meaningless information. It will be maintained by the text file. Notepad is used to create the text file.

Note pad will not maintain any dictionary.

Eg: Text files.

**Data :** It is a meaningful information. It will be maintained by word document file. Word pad is used to create the create the word document file. Word pad will maintain the dictionary.

Eg: MS-Word document File.

**Database :** It is a structured data. It will be maintained in the form of records. Database will be maintained in the form of files.

Eg: Excel files,

**DBMS ( Data Base Management System ) :**

Data will be maintained in the files. This data is to be updated to meet the client ( eg : company, bank, railways ) requirements, Some Mechanism is required to update the data. Such mechanism is called as database management system. Some commands will be maintained by the DBMS to update the data.

It is collection of Data and Commands to update the data.

Eg :

INSERT, DELETE, UPDATE, SELECT

In the DBMS data will be maintained in the form of files.

Eg:

MS FOX-PRO, MS-ACCESS

MS FOX-PRO is standalone database. It can not be shared by multiple clients and it fails to maintain the huge amount of data base.

**Disadvantages of using files:**

1. No security for the files. There may be a chance of deleting files accidentally. Once file has been deleted, loss of entire data.

2) A file cannot be shared by the multiple users at a time through Network.

3) Data redundancy: It is duplication of data.

**Data Normalization cannot be achieved with files:**

4) Relationship cannot be established between Master Record and Detailed record.

5) There is no order of storing data such as ascending / descending order

6) Constraints cannot be established on the data.

7) Automatic generations of Id not possible using files

**RDBMS ( Relational Data Base Management System ) :**

With RDBMS we can overcome the disadvantages of DBMS packages.

Mathematician E.F CODD has formed **TWELVE** rules to overcome the all disadvantages with the DBMS. Any data base which satisfies at least **SIX** rules out of **TWELVE** rules formed by the mathematician CODD is called as RDBMS.

Eg: All Database servers are RDBMS

ORACLE , MS-SQL , MY-SQL, INFORMIX, DB2, SYBASE, postgres

### Dr Edgar F Codd’s Twelve Commandments

**Rule 0: Foundation rule**

Any relational database management system that is propounded to be RDBMS or advocated to be a RDBMS should be able to manage the stored data in its entirety through its relational capabilities.

**Rule 1: Rule of Information**

Relational Databases should store the data in the form of relations. Tables are relations in Relational Database Management Systems. Be it any user defined data or meta-data, it is important to store the value as an entity in the table cells.

**Rule 2: Rule of Guaranteed Access**

The use of pointers to access data logically is strictly forbidden. Every data entity which is atomic in nature should be accessed logically by using a right combination of the name of table, primary key represented by a specific row value and column name represented by attribute value.

**Rule 3: Rule of Systematic Null Value Support**

Null values are completely supported in relational databases. They should be uniformly considered as ‘missing information’. Null values are independent of any data type. They should not be mistaken for blanks or zeroes or empty strings. Null values can also be interpreted as ‘inapplicable data’ or ‘unknown information.’

**Rule 4: Rule of Active and online relational Catalog**

In the Database Management Systems lexicon, ‘metadata’ is the data about the database or the data about the data. The active online catalog that stores the metadata is called ‘Data dictionary’. The so called data dictionary is accessible only by authored users who have the required privileges and the query languages used for accessing the database should be used for accessing the data of data dictionary.

**Rule 5: Rule of Comprehensive Data Sub-language**

A single robust language should be able to define integrity constraints, views, data manipulations, transactions and authorizations. If the database allows access to the aforementioned ones, it is violating this rule.

**Rule 6: Rule of Updating Views**

Views should reflect the updates of their respective base tables and vice versa. A view is a logical table which shows restricted data. Views generally make the data readable but not modifiable. Views help in data abstraction.

**Rule 7: Rule of Set level insertion, update and deletion**

A single operation should be sufficient to retrieve, insert, update and delete the data.

**Rule 8: Rule of Physical Data Independence**

Batch and end user operations are logically separated from physical storage and respective access methods.

**Rule 9: Rule of Logical Data Independence**

Batch and end users can change the database schema without having to recreate it or recreate the applications built upon it.

**Rule 10: Rule of Integrity Independence**

Integrity constraints should be available and stored as metadata in data dictionary and not in the application programs.

**Rule 11: Rule of Distribution Independence**

The Data Manipulation Language of the relational system should not be concerned about the physical data storage and no alterations should be required if the physical data is centralized or distributed.

**Rule 12: Rule of Non Subversion**

Any row should obey the security and integrity constraints imposed. No special privileges are applicable.

**Comparison of different Database servers**

**ORACLE:**

It was developed by the Oracle Corporation.

It is a mostly used data base in the real time applications.

It was developed for all Operating Systems and Machines in the market

We can achieve 95 % of RDBMS with this data base.

O/S : WINDOWS, LINUX, UNIX, SOLARIS, IBM-AIX, HP-UX,

Machines : Intel Machines and Spark Machines.

**MS-SQL Server :**

It was developed by the Micro Soft.

It is a mostly used with Micro Soft front end products.

It was developed only for Window Operating Systems and Intel Machines.

We can not achieve complete RDBMS with this data base.

Dis-Adv 🡪 There is no security for data base server as it is running on Windows OS.

**MYSQL :**

It was developed my Apache software foundation.

Later it was adopted by the Sun Soft.

It was developed for all Operating Systems and Machines in the market

It is in-built with SOLARIS Operating system.

We can not achieve complete RDBMS with this data base.

It fails to support too many users to share the data.

Recently it was taken over by Oracle Corporation.

**DB2 :**

It was developed by the IBM

It is a mostly used data base with IBM mainframe applications.

We can achieve 99 % of RDBMS with this data base.

This is called universal database.

O/S : WINDOWS, LINUX, UNIX, SOLARIS, IBM-AIX, HP-UX,

Machines : Intel Machines

**SYBASE :**

It was developed by the Sybase Corporation.

It was developed for all Operating Systems and Machines in the market

We cannot achieve complete RDBMS with this data base.

**INFORMICS :**

It was developed by Informics Software company.

It was developed only for windows Operating Systems and Intel Machines.

It is mostly used with C++ Applications.

We cannot achieve complete RDBMS with this data base.

Postgres:

It was developed by the Postgres Inc.

It was developed for all Operating Systems and Machines in the market.

It can be used with any front applications.

**Advantages of RDBMS**

**Primary Advantages :**

1. Data Security can be achieved
2. We can overcome Data redundancy
3. Data can be sharable
4. Data Normalization can be achieved

**Other Advantages:**

1) We can achieve the CODD’s Rules

2) Data representation is in the form of rows and columns.

3) Clients Operating system can be any one ( mostly clients are Windows O/S)

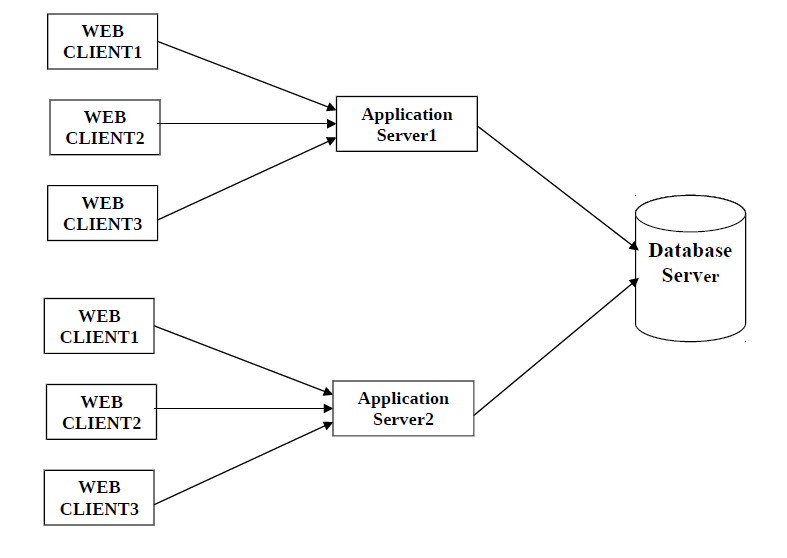
4) Data can stored in any format such Char, Number, Binary, Text

5) It provides support for ACID properties

6) Huge amount of data can be maintained

7) Data integrity can be achieved

## Databases on the Web



**Applications On the server :**

Every application will have its own drivers to connect to data base.

The common Applications are Java, **.**Net, ETL Tools, ERP Applications.

**Working with database**

Data will be stored in the form of tables.

Table is logical one. Internally (i.e on Hard Disk ), data will be maintained in form of files with security.

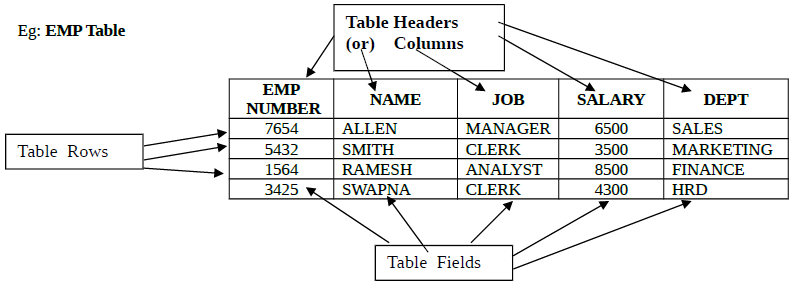
Oracle engine will look after the security of data files.

When user created a table, internally files generates to store and maintain data.

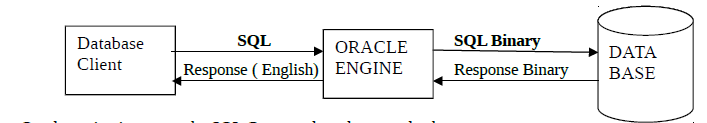
**Table :** It is a database object. It maintains the data in the from of records.

**Record :** It is an entity in a table. It is a collection of fields

**Field :** It is a member of entity. It is a intersection of Row and column.



**Structured Query Language ( SQL )**

It is a language, which can be used by the Oracle Engine to perform the database transactions.

Oracle engine interprets the SQL Commands and sent to database.

Oracle engine has it’s own interpreter to interpret the commands. Since it is an interpreter, it interprets the command each and every time of request.

**Compiler :** Generates Binary code only once and it will be used for every time. Hence compiler based one gives the more performance than the interpreter based.

**Role Of SQL Developer :**

1) SQL developer develop and test the SQL Queries as per the requirement of Front end Application Developer.

1. SQL developer will receive document form the front end application developer and some times form DBA

3) Should go through the documentation and analyze the requirements given by the front end application developers and DBA

4) SQL developer should prepare technical documents, which shows logic implemented in the development of queries and also functionality of the queries.

5) SQL Queries will be saved in the form of text file called SQL scripts. extension ( . SQL)

6) These Technical documents and SQL scripts will be submitted to client as the client required these documents for the future in-order to maintain project.

7) SQL scripts are of two types 1) Front End Application developer scripts 2) Data base set up scripts

8) DBA will set up the data base at the client location called on site by using Data base set up scripts

**SQL and NOSQL Database**

**(NOSQL --> N**ot **O**nly **S**tructured **Q**uery **L**anguage**)**

|  |  |  |
| --- | --- | --- |
| **S.No** | **SQL** | **NOSQL** |
| **1.** | Row Oriented data | Column Oriented data |
| **2.** | Normalized data | De-normalized data |
| **3.** | Small amount of structured, semi-structured and un-structured, but good for structured data. | Large amount of data, structured, semi-structured not un-structured data |
| **4.** | Supports both OLTP and OLAP, mostly used for OLTP | Supports both OLTP and OATP, mostly used for OLAP |
| **5.** | Supports CRUD Operations | Supports CRUD Operations |
| **6.** | Supports ACID Properties | Non ACID |
| **7.** | Joins, views, indexes,triggers | No joins |
| **8.** | Platform Independent | Platform Independent |
| **9.** | An RDBMS is governed by its schema, which describes the whole structure of tables. | is schema-less, it doesn't have the concept of fixed columns schema; defines only column families. i.e schema free |
| **10.** | Vertical Scaling | Horizontal scaling |
| **11.** | Primary Key and foreign key | Only Primary key, no foreign key concept |
| **12** | SQL supports | Only query language |

**Oracle database**

**Different editions of Oracle database.**

Following are the four editions of the Oracle database.

* **Enterprise Edition:** It is the most robust and secure edition. It offers all features, including superior performance and security.

**It is mostly used database in the industry**

* **Standard Edition:** It provides the base functionality for users that do not require Enterprise Edition's robust package.

**Used for practice.**

* **Express Edition (XE):** It is the lightweight, free and limited Windows and Linux edition.

**Used for Learning data base**

* **Oracle Lite:** It is designed for mobile devices.

**The Oracle Corporation**

Oracle Corporation is the largest software company in the field of database business. Its relational database was the first to support SQL which has since become the industry standard.

Oracle database is one of the most trusted and widely used relational database engines.

**Brief History of Oracle Database**

The current version of Oracle Database is the result of over 35 years of innovative development.

evolution of Oracle Database include the following:

**Founding of Oracle**

In 1977, Larry Ellison, Bob Miner, and Ed Oates started the consultancy Software Development Laboratories, which became Relational Software, Inc. (RSI). In 1983, RSI became Oracle Systems Corporation and then later Oracle Corporation.

**First commercially available RDBMS**

In 1979, RSI introduced Oracle V2 (Version 2) as the first commercially available [SQL](https://docs.oracle.com/database/121/CNCPT/glossary.htm" \l "GUID-30430C74-B2C1-43A0-B7FF-64111B786BC5)-based RDBMS, a landmark event in the history of relational databases.

**Portable version of Oracle Database**

Oracle Version 3, released in 1983, was the first relational database to run on mainframes, minicomputers, and PCs. The database was written in C, enabling the database to be ported to multiple platforms.

Enhancements to concurrency control, data distribution, and scalability

Version 4 introduced multiversion [read consistency](https://docs.oracle.com/database/121/CNCPT/glossary.htm" \l "GUID-A2D92FBD-9C36-432E-A44F-0462DB2E5527).

Version 5, released in 1985, supported client/server computing and [distributed database](https://docs.oracle.com/database/121/CNCPT/glossary.htm" \l "GUID-DB7296DF-74E0-45E1-9BC2-5052DD543214) systems.

Version 6 brought enhancements to disk I/O, row locking, scalability, and backup and recovery. Also, Version 6 introduced the first version of the **[PL/SQL](https://docs.oracle.com/database/121/CNCPT/glossary.htm" \l "GUID-68532EDC-3324-48DE-9AD5-52CD2EC671EA)** language, a proprietary procedural extension to SQL.

**PL/SQL stored program units**

Oracle7, released in 1992, introduced PL/SQL stored procedures and triggers.

**Objects and partitioning**

Oracle8 was released in 1997 as the object-relational database, supporting many new data types. Additionally, Oracle8 supported partitioning of large tables.

**Internet computing**

Oracle8i Database, released in 1999, provided native support for internet protocols and server-side support for Java. Oracle8i was designed for internet computing, enabling the database to be deployed in a multitier environment.

**Oracle Real Application Clusters (Oracle RAC)**

Oracle9i Database introduced Oracle RAC in 2001, enabling multiple instances to access a single database simultaneously. Additionally, Oracle XML Database **([Oracle XML DB](https://docs.oracle.com/database/121/CNCPT/glossary.htm" \l "GUID-FFC80925-7152-41DA-8D78-47D8201FA2CA))** introduced the ability to store and query XML.

**Grid computing**

Oracle Database 10g introduced [grid computing](https://docs.oracle.com/database/121/CNCPT/glossary.htm" \l "GUID-EF507F26-F59A-415E-8C3E-3B95BFBA2F7C) in 2003. This release enabled organizations to virtualize computing resources by building a [grid infrastructure](https://docs.oracle.com/database/121/CNCPT/glossary.htm" \l "GUID-B9983AA6-0B7F-4FC3-B5FC-36DD4F4E1997) based on low-cost commodity servers. A key goal was to make the database self-managing and self-tuning. [Oracle Automatic Storage Management (Oracle ASM)](https://docs.oracle.com/database/121/CNCPT/glossary.htm" \l "GUID-8E45BEB5-F8AF-4896-B04A-7016812BD96D) helped achieve this goal by virtualizing and simplifying database storage management.

**Manageability, diagnosability, and availability**

Oracle Database 11g, released in 2007, introduced a host of new features that enabled administrators and developers to adapt quickly to changing business requirements. The key to adaptability is simplifying the information infrastructure by consolidating information and using automation wherever possible.

**Plugging In to the Cloud**

Oracle Database 12c, released in 2013, was designed for the Cloud, featuring a new Multitenant architecture, In-Memory column store, and support for JSON documents. Oracle Database 12c helps customers make more efficient use of their IT resources, while continuing to reduce costs and improve service levels for users.

**Oracle versions and its release year.**

|  |  |  |
| --- | --- | --- |
| **S.No** | **Version** | **Year** |
| 1. | Version 1 | 1977 |
| 2. | Version 2 | 1979 |
| 3. | Version 3 | 1983 |
| 4. | Version 4 | Not released |
| 5. | Version 5 | 1985 |
| 6. | Version 6 | Not released |
| 7. | Oracle7 | 1992 |
| 8. | Oracle8i | 1999 |
| 10. | Oracle9i | 2001 |
| 11. | Oracle 10g | 2003 |
| 12. | Oracle 11g | 2007 |
| 13. | Oracle 12c | 2013 |

**Oracle Client Tools :**

1) SQL \* PLUS

2) Forms 6i/ 10g

3) Reports 6i/10g

4) Oracle Apps 11i/ R-12

5) Oracle Apps Frame Work

6) Oracle ADF

7) Oracle Fusion

8) OBIEE

9) TOAD

10) Oracle Developer

**SQL \* PLUS:**

It is a Command Line Interface provided by the Oracle Corporation to work with server.

It is used by the SQL developer to develop and test SQL queries.

After development and testing these queries will be given to front end application developer

and also to DBA.

**Oracle Database Concepts**

**Schema Objects**

One characteristic of an RDBMS is the independence of physical data storage from logical data structures.

In Oracle Database, a database **[schema](https://docs.oracle.com/database/121/CNCPT/glossary.htm" \l "GUID-D55ED7A6-3BC4-4A16-981F-92E7E905A64D)** is a collection of logical data structures, or schema objects. A database user owns a database schema, which has the same name as the [user name](https://docs.oracle.com/database/121/CNCPT/glossary.htm" \l "GUID-E141F64B-73ED-43DC-B22A-DA0210B8EBEF).

Schema objects are user-created structures that directly refer to the data in the database. The database supports many types of schema objects, the most important of which are tables and indexes.

A schema object is one type of [database object](https://docs.oracle.com/database/121/CNCPT/glossary.htm" \l "GUID-785767E0-DB47-4579-AA4D-F46C8680ACCA). Some database objects, such as profiles and roles, do not reside in schemas.

**Tables**

A table describes an entity such as employees. You define a table with a table name, such as employees, and set of columns. In general, you give each **[column](https://docs.oracle.com/database/121/CNCPT/glossary.htm" \l "GUID-F1D9AA5F-C66A-4D5F-A84C-8D7360DE3BE5)** a name, **a [data type](https://docs.oracle.com/database/121/CNCPT/glossary.htm" \l "GUID-3873B26C-657D-4508-B13D-9155F1D5D8F4)**, and a width when you create the table.

A table is a set of rows. A column identifies an attribute of the entity described by the table, whereas a **[row](https://docs.oracle.com/database/121/CNCPT/glossary.htm" \l "GUID-14228D8A-2E66-49A7-B041-DFB6CA759D8C)** identifies an instance of the entity. For example, attributes of the employees entity correspond to columns for employee ID and last name. A row identifies a specific employee.

You can optionally specify a rule, called an **[integrity constraint](https://docs.oracle.com/database/121/CNCPT/glossary.htm" \l "GUID-67F8FE8C-EBA5-4796-820A-8919982A1411)**, for a column. One example is a NOT NULL integrity constraint. This constraint forces the column to contain a value in every row.

### Indexes:

An index is an optional data structure that you can create on one or more columns of a table. Indexes can increase the performance of data retrieval.

When processing a request, the database can use available indexes to locate the requested rows efficiently. Indexes are useful when applications often query a specific row or range of rows.

Indexes are logically and physically independent of the data. Thus, you can drop and create indexes with no effect on the tables or other indexes. All applications continue to function after you drop an index.

**Data Access**

A general requirement for a DBMS is to adhere to accepted industry standards for a data access language.

**Structured Query Language (SQL)**

SQL is a set-based declarative language that provides an interface to an RDBMS such as Oracle Database. In contrast to procedural languages such as C, which describe how things should be done, SQL is nonprocedural and describes what should be done.

SQL is the ANSI standard language for relational databases. All operations on the data in an Oracle database are performed using SQL statements. For example, you use SQL to create tables and query and modify data in tables.

A SQL statement can be thought of as a very simple, but powerful, computer program or instruction. Users specify the result that they want (for example, the names of employees), not how to derive it. A SQL statement is a string of SQL text such as the following:

SQL statements enable you to perform the following tasks:

* Query data
* Insert, update, and delete rows in a table
* Create, replace, alter, and drop objects
* Control access to the database and its objects
* Guarantee database consistency and integrity

**PL/SQL and Java**

PL/SQL is a procedural extension to Oracle SQL. PL/SQL is integrated with Oracle Database, enabling you to use all of the Oracle Database SQL statements, functions, and data types. You can use PL/SQL to control the flow of a SQL program, use variables, and write error-handling procedures.

A primary benefit of PL/SQL is the ability to store application logic in the database itself. A **[PL/SQL procedure](https://docs.oracle.com/database/121/CNCPT/glossary.htm" \l "GUID-D2210901-18F2-43C0-B0CC-6953E94CD7EF)** or **[function](https://docs.oracle.com/database/121/CNCPT/glossary.htm" \l "GUID-5611828A-3621-4527-AEB1-12017A454E94)** is a schema object that consists of a set of SQL statements and other PL/SQL constructs, grouped together, stored in the database, and run as a unit to solve a specific problem or to perform a set of related tasks. The principal benefit of server-side programming is that built-in functionality can be deployed anywhere.

Oracle Database can also store program units written in Java. A Java stored procedure is a Java method published to SQL and stored in the database for general use. You can call existing PL/SQL programs from Java and Java programs from PL/SQL.

**Transaction Management**

Oracle Database is designed as a multiuser database. The database must ensure that multiple users can work concurrently without corrupting one another's data.

**Transactions**

A transaction is a logical, atomic unit of work that contains one or more SQL statements. An RDBMS must be able to group SQL statements so that they are either all committed, which means they are applied to the database, or all rolled back, which means they are undone.

An illustration of the need for transactions is a funds transfer from a savings account to a checking account. The transfer consists of the following separate operations:

1. Decrease the savings account.
2. Increase the checking account.
3. Record the transaction in the transaction journal.

Oracle Database guarantees that all three operations succeed or fail as a unit. For example, if a hardware failure prevents a statement in the transaction from executing, then the other statements must be rolled back.

Transactions are one feature that set Oracle Database apart from a file system. If you perform an atomic operation that updates several files, and if the system fails halfway through, then the files will not be consistent. In contrast, a transaction moves an Oracle database from one consistent state to another. The basic principle of a transaction is "all or nothing": an atomic operation succeeds or fails as a whole.

**Data Concurrency**

A requirement of a multiuser RDBMS is the control of data concurrency, which is the simultaneous access of the same data by multiple users.

Without concurrency controls, users could change data improperly, compromising **[data integrity](https://docs.oracle.com/database/121/CNCPT/glossary.htm" \l "GUID-9DE527B3-8901-4F4C-A18F-D2C2C307AFE1).** For example, one user could update a row while a different user simultaneously updates it.

If multiple users access the same data, then one way of managing concurrency is to make users wait. However, the goal of a DBMS is to reduce wait time so it is either nonexistent or negligible. All SQL statements that modify data must proceed with as little interference as possible. Destructive interactions, which are interactions that incorrectly update data or alter underlying data structures, must be avoided.

Oracle Database uses locks to control concurrent access to data. A **[lock](https://docs.oracle.com/database/121/CNCPT/glossary.htm" \l "GUID-6D016291-A487-4F88-BE0B-ACF8FA2AE72C)** is a mechanism that prevents destructive interaction between transactions accessing a shared resource. Locks help ensure data integrity while allowing maximum concurrent access to data.

**Data Consistency**

In Oracle Database, each user must see a consistent view of the data, including visible changes made by a user's own transactions and committed transactions of other users.

For example, the database must prevent the **[lost update](https://docs.oracle.com/database/121/CNCPT/glossary.htm" \l "GUID-F9DA6F63-B518-44C5-B98B-2425680EB498)** problem, which occurs when one transaction sees uncommitted changes made by another concurrent transaction.

Oracle Database always enforces statement-level **[read consistency](https://docs.oracle.com/database/121/CNCPT/glossary.htm" \l "GUID-A2D92FBD-9C36-432E-A44F-0462DB2E5527)**, which guarantees that the data that a single query returns is committed and consistent for a single point in time. Depending on the transaction isolation level, this point is the time at which the statement was opened or the time the transaction began. The Flashback Query feature enables you to specify this point in time explicitly.

The database can also provide read consistency to all queries in a transaction, known as transaction-level read consistency. In this case, each statement in a transaction sees data from the same point in time, which is the time at which the transaction began.

**Oracle SQL**

SQL is a database understandable language designed for the retrieval and management of data in relational database. SQL stands for Structured Query Language.

**SQL is categorized into five**

1) Data Definition Language ( DDL )

2) Data Manipulation Language ( DML )

3) Data Query Language ( DQL )

4) Data Control language ( DCL )

5) Transaction Control Language ( TCL )

**1) Data Definition Language ( DDL ):**

These are used to define the database object. These will not work with data. These commands are **create, alter, rename and drop** the database objects such as tables, Views, synonyms and etc…

**2) Data Manipulation Language ( DML ):**

These are used update the data in the database table such as **INSERT** new record, **DELETE** and **UPDATE** the existing records.

**3) Data Query Language ( DQL ) :**

It is also called as data retrieval language. These are used to query the data form the database as desired. **SELECT** Command is used to query the data form database.

**4) Data Control language ( DCL ) :**

These commands are used to restrict users on the database objects. To have the control over the users while accessing data. Grant Privileges and to taking back ( revoke ) the privileges.

**5) Transaction Control Language ( TCL ) :**

These commands are used to have the control over the data transactions. COMMIT and ROLLBACK the transactions.

**Data types**

It represents the type of data, that can be stored in variable or filed. These are categorized into two.

1) Simple Data Types 2) Composite Data Types

**Simple Data Types ( for column ) :**

**T**hese are built-in data types provided by oracle database software. It is used to maintain one field data.

**Composite Data Types ( for Row / partial row ) :**

These are also built-in data types provided by oracle database software. It is used to maintain the whole record or part of the record ( more than one filed ). These are derived from the simple data type. These are used in with PL/SQL .

**Simple Data Types**

**1) Character Data Types**

Character data types store character (alphanumeric) data, which are words and free-form text, in the database character set or national character set. They are less restrictive than other data types and consequently have fewer properties. For example, character columns can store all alphanumeric values, but NUMBER columns can store only numeric values.

Character data is stored in strings with byte values corresponding to one of the character sets, such as 7-bit ASCII, specified when the database was created. Oracle Database supports both single-byte and multibyte character sets.

These data types are used for character data:

* [CHAR Data Type](https://docs.oracle.com/database/121/SQLRF/sql_elements001.htm" \l "i45647)
* [NCHAR Data Type](https://docs.oracle.com/database/121/SQLRF/sql_elements001.htm" \l "i45672)
* [VARCHAR2 Data Type](https://docs.oracle.com/database/121/SQLRF/sql_elements001.htm" \l "i45694)
* [NVARCHAR2 Data Type](https://docs.oracle.com/database/121/SQLRF/sql_elements001.htm" \l "i45685)

### CHAR Data Type

The CHAR data type specifies a fixed-length character string in the database character set. You specify the database character set when you create your database.

When you create a table with a CHAR column, you specify the column length as size optionally followed by a length qualifier. The qualifier BYTE denotes byte length semantics while the qualifier CHAR denotes character length semantics. In the byte length semantics, size is the number of bytes to store in the

Oracle ensures that all values stored in a CHAR column have the length specified by size in the selected length semantics. If you insert a value that is shorter than the column length, then Oracle blank-pads the value to column length. If you try to insert a value that is too long for the column, then Oracle returns an error. Note that if the column length is expressed in characters, blank-padding does not guarantee that all column values have the same byte length.

You can omit size from the column definition. The default value is 1.

The maximum value of size is 2000, which means 2000 bytes or characters (code points), depending on the selected length semantics. However, independently, the absolute maximum length of any character value that can be stored into a CHAR column is 2000 bytes. For example, even if you define the column length to be 2000 characters, Oracle returns an error if you try to insert a 2000-character value in which one or more code points are wider than 1 byte.

Recommended is less than or equal to 500

### VARCHAR2 Data Type

The VARCHAR2 data type specifies a variable-length character string in the database character set. You specify the database character set when you create your database.

Oracle stores a character value in a VARCHAR2 column exactly as you specify it, without any blank-padding, provided the value does not exceed the length of the column. If you try to insert a value that exceeds the specified length, then Oracle returns an error.

The minimum value of size is 1. The maximum value is:

* 32767 if MAX\_STRING\_SIZE = EXTENDED
* 4000 if MAX\_STRING\_SIZE = STANDARD ( recommended )

### VARCHAR Data Type

Do not use the VARCHAR data type. Use the VARCHAR2 data type instead. Although the VARCHAR data type is currently synonymous with VARCHAR2, the VARCHAR data type is scheduled to be redefined as a separate data type used for variable-length character strings compared with different comparison semantics.

### NVARCHAR2 Data Type

The NVARCHAR2 data type specifies a variable-length character string in the national character set. NVARCHAR2 is a Unicode-only data type.

When you create a table with an NVARCHAR2 column, you must specify the column length as size characters, or more precisely, code points in the national character set. One code point has always 2 bytes.

Oracle stores a character value in an NVARCHAR2 column exactly as you specify it, without any blank-padding, provided the value does not exceed the length of the column. If you try to insert a value that exceeds the specified length, then Oracle returns an error.

The minimum value of size is 1. The maximum value is:

* 16383 if MAX\_STRING\_SIZE = EXTENDED and the national character set is AL16UTF16
* 2000 if MAX\_STRING\_SIZE = STANDARD and the national character set is AL16UTF16
* 32767 if MAX\_STRING\_SIZE = EXTENDED and the national character set is UTF8
* 4000 if MAX\_STRING\_SIZE = STANDARD and the national character set is UTF8

**2) Numeric Data Types**

The Oracle Database numeric data types store positive and negative fixed and floating-point numbers, zero, infinity, and values that are the undefined result of an operation—"not a number" or NAN. For information on specifying numeric data types as literals, refer to ["Numeric Literals"](https://docs.oracle.com/database/121/SQLRF/sql_elements003.htm" \l "i139891).

### NUMBER Data Type

The NUMBER data type stores zero as well as positive and negative fixed numbers with absolute values from 1.0 x 10-130 to but not including 1.0 x 10126. If you specify an arithmetic expression whose value has an absolute value greater than or equal to 1.0 x 10126, then Oracle returns an error. Each NUMBER value requires from 1 to 22 bytes.

Specify a fixed-point number using the following form:

NUMBER(p, s)

where: p is the precision, or the maximum number of significant decimal digits, where the most significant digit is the left-most nonzero digit, and the least significant digit is the right-most known digit. Oracle guarantees the portability of numbers with precision of up to 20 base-100 digits, which is equivalent to 39 or 40 decimal digits depending on the position of the decimal point.

* s is the scale, or the number of digits from the decimal point to the least significant digit. The scale can range from -84 to 127.
  + Positive scale is the number of significant digits to the right of the decimal point to and including the least significant digit.
  + Negative scale is the number of significant digits to the left of the decimal point, to but not including the least significant digit. For negative scale the least significant digit is on the left side of the decimal point, because the actual data is rounded to the specified number of places to the left of the decimal point. For example, a specification of (10,-2) means to round to hundreds.

Scale can be greater than precision, most commonly when e notation is used. When scale is greater than precision, the precision specifies the maximum number of significant digits to the right of the decimal point. For example, a column defined as NUMBER(4,5) requires a zero for the first digit after the decimal point and rounds all values past the fifth digit after the decimal point.

It is good practice to specify the scale and precision of a fixed-point number column for extra integrity checking on input. Specifying scale and precision does not force all values to a fixed length. If a value exceeds the precision, then Oracle returns an error. If a value exceeds the scale, then Oracle rounds it.

Specify an integer using the following form:

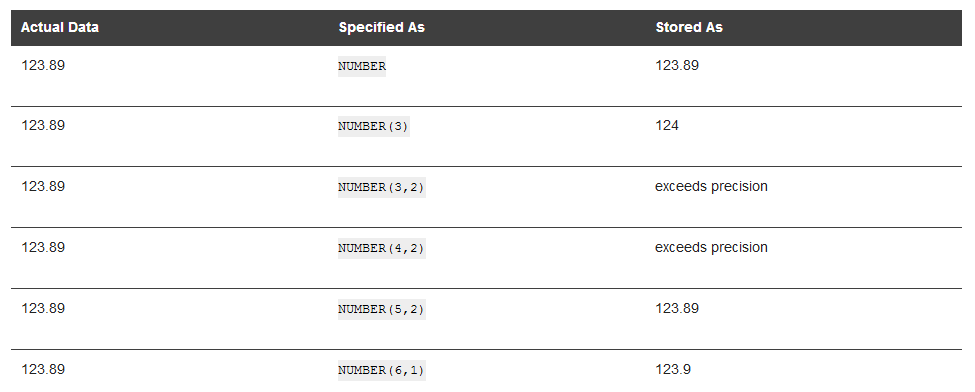
NUMBER(p)

This represents a fixed-point number with precision p and scale 0 and is equivalent to NUMBER(p,0).

Specify a floating-point number using the following form:

NUMBER

The absence of precision and scale designators specifies the maximum range and precision for an Oracle number.



### FLOAT Data Type

The FLOAT data type is a subtype of NUMBER. It can be specified with or without precision, which has the same definition it has for NUMBER and can range from 1 to 126. Scale cannot be specified, but is interpreted from the data. Each FLOAT value requires from 1 to 22 bytes.

To convert from binary to decimal precision, multiply n by 0.30103. To convert from decimal to binary precision, multiply the decimal precision by 3.32193. The maximum of 126 digits of binary precision is roughly equivalent to 38 digits of decimal precision.

**5) long :** It is used to store both character and numeric value of max 2 gb.

**6) date :** It is used to store system date and time . default format in oracle is 🡪 **DD-MM-YY**

Eg:

hire\_date date ;

birth\_date date ;

**7) raw :**

it is a binary data type to store binary data such as Images. It max size that can be accommodate is 2000 bytes.

**8) long raw** :

It is also a binary data type. It can accommodate huge amount of data ( max : 4 GB )

**9) large object s ( LOBs) :**

these were introduced in Oracle-8 version. Lobs can accommodate the data max 4 GB.

1) CLOBs:

Character Large Objects : to store text files.

2) BLOBs:

Binary Large Objects ( to store Binary files )

3) CNLOBs:

(Character Numeric Large Objects )

## Extended Data Types (Beginning with Oracle Database 12c )

## Rowid Data Types

Each row in the database has an address. The rows in heap-organized tables that are native to Oracle Database have row addresses called rowids. You can examine a rowid row address by querying the pseudocolumn ROWID. Values of this pseudocolumn are strings representing the address of each row. These strings have the data type ROWID. You can also create tables and clusters that contain actual columns having the ROWID data type. Oracle Database does not guarantee that the values of such columns are valid rowids. Rowids contain the following information:

* The data block of the data file containing the row. The length of this string depends on your operating system.
* The row in the data block.
* The database file containing the row. The first data file has the number 1. The length of this string depends on your operating system.
* The data object number, which is an identification number assigned to every database segment. You can retrieve the data object number from the data dictionary views USER\_OBJECTS, DBA\_OBJECTS, and ALL\_OBJECTS. Objects that share the same segment (clustered tables in the same cluster, for example) have the same object number.

Rowids are stored as base 64 values that can contain the characters A-Z, a-z, 0-9, and the plus sign (+) and forward slash (/). Rowids are not available directly. You can use the supplied package DBMS\_ROWID to interpret rowid contents. The package functions extract and provide information on the four rowid elements listed above.

### UROWID Data Type

The rows of some tables have addresses that are not physical or permanent or were not generated by Oracle Database. For example, the row addresses of index-organized tables are stored in index leaves, which can move. Rowids of foreign tables (such as DB2 tables accessed through a gateway) are not standard Oracle rowids.

Oracle uses universal rowids (urowids) to store the addresses of index-organized and foreign tables. Index-organized tables have logical urowids and foreign tables have foreign urowids. Both types of urowid are stored in the ROWID pseudocolumn (as are the physical rowids of heap-organized tables).

Oracle creates logical rowids based on the primary key of the table. The logical rowids do not change as long as the primary key does not change. The ROWID pseudocolumn of an index-organized table has a data type of UROWID. You can access this pseudocolumn as you would the ROWID pseudocolumn of a heap-organized table (using a SELECT ... ROWID statement). If you want to store the rowids of an index-organized table, then you can define a column of type UROWID for the table and retrieve the value of the ROWID pseudocolumn into that column.

**Data Definition Language**

These commands are 1) CREATE, 2) ALTER, 3) RENAME 4) DROP and 5) TRUNCATE

**1) CREATION OF A TABLE**

**Rules to create a table :**

1) Table name should be start with alphabet

2) Spaces should not be used.

3) Valid chars are 🡪 A-Z , a-z , 0-9, # , $ , \_ ( under score )

4) Max length of the table name is 30 chars

5) Oracle key words and reserve words should not be used as table name.

6) No two database objects such as tables, views, synonym and etc… exists with same name

7) Duplicate columns are not allowed in table

**Note :** table names, column names, data types are not case sensitive, where as data is a case sensitive

**Syntax :**

create < Table-Name> ( Column1 datatype(size), Col2 datatype (size), ……….);

eg:

SQL> CREATE TABLE My\_Emp( empno number(10),

ename varchar2(20),

salary number(10,2),

comm number(5) ) ;

sql> select \* from tab;

SQL> DESC MY\_EMP;

Name Null? Type

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

EMPNO NUMBER(10)

ENAME VARCHAR2(20)

SALARY NUMBER(10,2)

COMM NUMBER(5)

**desc** 🡪 is to display the structure of the table

SQL> CREATE TABLE Item( Item\_Code varchar2(10),

Item\_Name varchar2(20),

Item\_Price number(10,2));

SQL> DESC ITEM;

Name Null? Type

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

ITEM\_CODE VARCHAR2(10)

ITEM\_NAME VARCHAR2(20)

ITEM\_PRICE NUMBER(10,2)

SQL> CREATE TABLE MY\_DEPT ( deptno number(10),

dname varchar2(15),

loc varchar2(20));

SQL> DESC MY\_DEPT;

Name Null? Type

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

DEPTNO NUMBER(10)

DNAME VARCHAR2(15)

LOC VARCHAR2(20)

**SQL> CL SCR ;** 🡪 **To clear the screen**

**2) ALTERING A TABLE DEFINITION**

ALTER is a command to alter the table definition ( structure )

ALETR used for following operations

1) Add columns 2) Modify the column definition 3) Rename Column 4) Drop Columns

**1) Adding Columns**

**ADD** is command to add the columns

**Syntax :**

**Adding single Column**

ALTER TABLE <table\_name> ADD <column\_name> DATATYPE(size)

Eg:

before alter table MY\_EMP

SQL> DESC MY\_EMP;

Name Null? Type

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

EMPNO NUMBER(10)

ENAME VARCHAR2(20)

SALARY NUMBER(10,2)

COMM NUMBER(5)

SQL> ALTER TABLE My\_Emp ADD deptno number(10);

After alter

SQL> DESC My\_Emp;

Name Null? Type

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

EMPNO NUMBER(10)

ENAME VARCHAR2(20)

SALARY NUMBER(10,2)

COMM NUMBER(5)

DEPTNO NUMBER(10)

**Note : New Column will be the last column in a table.**

**Adding multiple columns**

ALTER TABLE <table\_name> ADD ( Col1 DATATYPE(size), Col2 DATATYPE (size), …., … );

Eg: **Before alter ITEM table**

SQL> DESC ITEM;

Name Null? Type

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

ITEM\_CODE VARCHAR2(10)

ITEM\_NAME VARCHAR2(20)

ITEM\_PRICE NUMBER(10,2)

SQL > ALTER TABLE ITEM ADD ( Item\_Batch number(10), Item\_Exp\_Date date );

**After alter ITEM table**

SQL> desc item;

Name Null? Type

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

ITEM\_CODE VARCHAR2(10)

ITEM\_NAME VARCHAR2(20)

ITEM\_PRICE NUMBER(10,2)

ITEM\_BATCH NUMBER(10)

ITEM\_EXP\_DATE DATE

**2) Modifying Column Definition**

This operation is to increase /decrease the size column or to change the data type of the column.

MODIFY is command to modify the table definition.

**Syntax :**

**Single Column**

ALTER TABLE < table\_name> MODIFY <column\_name> datatype (size)

**Multiple columns**

ALTER TABLE < table\_name> MODIFY ( Col1 datatype(size), Col2 datatype(size),….,…..);

SQL> ALTER TABLE My\_EMP MODIFY ename varchar2(30);

valid if new size is sufficient for existing data / column is empty

SQL> ALTER TABLE My\_EMP MODIFY (ename varchar2(30), salary number(10,2), COMM number(8) );

**Change of data type of the column is valid, if columns is empty.**

**3) Renaming a Column**

**Syntax :**

ALTER TABLE < table\_name> RENAME COLUMN <old\_name> to <new\_name>

Eg:

SQL> ALTER TABLE My\_EMP RENAME COLUMN ename to emp\_name;

Note : Multiple columns cannot be renamed at a time.

**4) Dropping a Columns**

1) Any column of the table can be dropped from the table

2) Single column or Multiple columns can be dropped at time

3) Once column has been dropped, the data in the column will also be removed

4) We cannot get back the Dropped columns

5) Table should contain at least one column after dropping a column.

**Syntax :**

**Single Column:**

ALTER TABLE < table\_name> DROP COLUMN <column\_name>

**Multiple Columns :**

ALTER TABLE < table\_name> DROP ( Col1, Col2, Col3,…);

Eg:

SQL > ALTER TABLE My\_EMP DROP COLUMN emp\_name ;

SQL> ALTER TABLE My\_EMP DROP (salary, comm, deptno);

**Overview :**

**1) Adding Columns**

**Adding single Column**

ALTER TABLE <table\_name> ADD <column\_name> DATATYPE(size)

**Adding multiple columns**

ALTER TABLE <table\_name> ADD ( Col1 DATATYPE(size), Col2 DATATYPE (size), …., …);

**2) Modifying Column Definition**

**Single Column**

ALTER TABLE < table\_name> MODIFY <column\_name> datatype (size)

**Multiple columns**

ALTER TABLE < table\_name> MODIFY ( Col1 datatype(size), Col2 datatype(size),…., …..);

**3) Renaming a Column**

**Only single column**

ALTER TABLE < table\_name> RENAME **COLUMN** <old\_name> to <new\_name>

**4) Dropping a Columns**

**Single Column:**

ALTER TABLE < table\_name> DROP **COLUMN** <column\_name>

**Multiple Columns :**

ALTER TABLE < table\_name> DROP ( Col1, Col2, Col3,…);

**Note: while renaming and dropping single columns, syntax should have COLUMN Keyword for other operations not required.**

**3) RENAMING A TABLE**

**Syntax :**

RENAME <old\_name> to <new\_name>

Eg:

SQL> rename My\_Emp to Emp\_Test

SQL> desc My\_Emp;

ERROR: ORA-04043: object MY\_Emp does not exist

**4) DROPPING A TABLE**

**Syntax :**

DROP TABLE <table\_name>

SQL> drop table My\_Dept;

Table dropped.

SQL> desc My\_Dept;

ERROR:

ORA-04043: object My\_Dept does not exist

In the old version of ORACLE once table has been dropped, we can not get back the dropped table

**RECYCLE BIN:**

**Recycle Bin** concept has been introduced in the new Version ORACLE – 10g to get back the dropped table.

When any table dropped it will be moved to Recycle Bin. When table has been moved to recycle bin one object will created for the table in recycle bin 1) Original Name 2) Object Name

SQL> desc recyclebin

NAME NULL ? TYPE

ORIGINAL\_NAME -----

OBJECT\_NAME ----

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

SQL> select Original\_name, Object\_name from recyclebin;

displays the data in the recyclebin

SQL> save recycle.sql

SQL> show recyclebin

displays the data in the recyclebin

SQL> purge recyclebin;

removes the all tables in the recyclebin

SQL>@recycle.sql;

ORIGINAL\_NAME OBJECT\_NAME

My\_Dept ------

SQL> drop table My\_Emp;

SQL>@recycle.sql;

ORIGINAL\_NAME OBJECT\_NAME

My\_Dept ------

My\_Emp ------

SQL> purge recyclebin;

**Flash back table**  to get back the dropped tables.

Once purge on recycle bin completed, we cannot get back the tables.

SQL> flashback table My\_Emp;

**Data Dictionary for tables**

USER\_TABLES 🡪 is data dictionary, which maintains the information about all table in a login user.

SQL> select \* from tab;

it shows all the **tables** in a current login user

SQL> select table\_name from user\_tables;

it shows all **tables** in a current login user

**Note : All DDL Commands are auto commit. The operations performed using DDL commands are automatically saved.**

**Data Query Language ( DQL )**

SELECT is a DQL operation

SELECT is a query statement, which will be used to fetch the data from the table

**Working with SELECT Statement**

**Syntax:**

Select \* from <table\_name> ;

( \* )🡪 all the fields from the table

**Fetching particular fields from the table**

Select Col1, Col2, Col3, ….. FROM <table\_name>

Eg :

SQL> SELECT \* FROM EMP;

SQL> SELECT EMPNO, ENAME, SAL, COMM FROM EMP;

**Column alias names :**

1) It is assigning temporary name to columns while displaying output

2) Alias name should be in double quotations, if it contains chars other than the following

A-Z, a – z , 0-9, # , $ , under score ( **--** )

**Syntax:**

SELECT Col1 as alis\_name ,

Col2 as alias\_name,

Col3 as alias\_name FROM <table\_name>

Here use of **( as )** is an optional

SQL > SELECT EMPNO as EmpNumber,

ENAME as Name,

SAL as salary ,

COMM as commission

FROM EMP ;

**( OR )**

SQL> SELECT EMPNO EmpNumber,

ENAME Name,

SAL salary ,

COMM commission

FROM EMP ;

**Data Manipulation Language ( DML )**

It provides the following commands to update the data in the database tables

1) INSERT 🡪 To Insert the records into the database table

2) UPDATE🡪 To Modify the already existing record in the table

3) DELETE 🡪 To Delete the already existing record in the table

**INSERTING RECORDS INTO THE TABLE**

**Syntax:**

INSERT INTO <table\_name> VALUES (value1, value2, value3, ….., …….., ….. );

**Char and date type should be used in single quotation.**

SQL> desc emp;

Name Null? Type

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

EMPNO NOT NULL NUMBER(4)

ENAME CHAR(10)

JOB CHAR(9)

MGR NUMBER(4)

HIREDATE DATE

SAL NUMBER(7,2)

COMM NUMBER(7,2)

DEPTNO NOT NULL NUMBER(2)

SQL> insert into emp values( 1001, 'mnaro','software engineer', 7369, '23-jul-2009', 5600, 350, 20);

SQL> select \* from emp;

EMPNO ENAME JOB MGR HIREDATE SAL COMM DEPTNO

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

7839 KING PRESIDENT 17-NOV-81 5000 250 10

7698 BLAKE MANAGER 7839 01-MAY-81 2850 30

7782 CLARK MANAGER 7839 09-JUN-81 2450 300 10

7566 JONES MANAGER 7839 02-APR-81 2975 200 20

1001 mnaro software engineer 7369 23-JUL-09 5600 350 20

SQL> set num 4 ; 🡪 to width of the output column ( num value 2 to 50 )

SQL> insert into emp values( 1002, 'Sree Ram', 'clerk', 1002, '10-jun-2010', 2500, 450, 20);

SQL> select \* from emp;

SQL> insert into dept values (50,'FINANCE','DELHI');

SQL> select \* from dept;

SQL> insert into dept values (60,'EDP','HYDERABAD');

SQL> select \* from dept;

DEPTNO DNAME LOC

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

10 ACCOUNTING NEW YORK

20 RESEARCH DALLAS

30 SALES CHICAGO

40 OPERATIONS BOSTON

50 FINANCE DELHI

60 EDP HYDERABAD

6 rows selected.

**Create the some table and test it**

SQL> CREATE TABLE emp\_test ( empno number(10),

ename varchar2(30),

salary number(10,2),

deptno number(10)

);

SQL> SELECT \* FROM emp\_test;

no rows selected

SQL> INSERT INTO EMP\_TEST VALUES(1001, 'MNRAO', 5500, 20);

SQL> SELECT \* FROM EMP\_TEST;

SQL> INSERT INTO EMP\_TEST VALUES(1002,'ABCD',6500,10);

SQL> SELECT \* FROM EMP\_TEST;

**Inserting Null value into the table**

It is in two ways

1) Implicit Method 2) Explicit Method

**Implicit Method**

INSER INTO <table\_name> (Col1, col2, Col3 ) VALUES ( val1, val2, val3);

Eg:

SQL> INSERT INTO EMP\_TEST(EMPNO, ENAME, DEPTNO) VALUES(1010,'SIRISHA',20) ;

SQL> SELECT \* FROM EMP\_TEST ;

EMPNO ENAME SALARY DEPTNO

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

1001 MNRAO 5500 20

1002 ABCD 6500 10

1010 SIRISHA null 20

In the above result no value for salary of 1010 employee.

SQL> INSERT INTO EMP\_TEST ( EMPNO, SALARY) VALUES(1012, 5500);

SQL> SELECT \* FROM EMP\_TEST ;

SQL> INSERT INTO EMP\_TEST ( EMPNO, ENAME, SALARY) VALUES(1014, 'RAMESH', 7800) ;

SQL> SELECT \* FROM EMP\_TEST ;

**Explicit Method :**

**Syntax :**

**INSERT INTO <table\_name> VALUES( value1, value2, NULL, value3, null, alue4, …., …);**

**Eg:**

SQL> INSERT INTO EMP\_TEST VALUES(1020,'RAMA RAO',8500, NULL) ;

SQL> SELECT \* FROM EMP\_TEST ;

SQL> INSERT INTO EMP\_TEST VALUES(1025, NULL, NULL, NULL) ;

SQL> SELECT \* FROM EMP\_TEST ;

**Inserting records dynamically**

Oracle provides substation variables method to insert the records dynamically

**Syntax :**

INSERT INTO <table\_name> values( &col1, &col2, &col3, …, …., …. ) (or)

&Var1, &var2, &var3, …, …., …. )

Note : The column or variable name should be in single quotations for char and date type data

SQL> CREATE TABLE STUDENT( ROLLNO NUMBER(10),

NAME VARCHAR(20),

CLASS VARCHAR2(10),

SECTION VARCHAR2(10) ) ;

SQL> INSERT INTO STUDENT VALUES(

&RNO,

&NAME,

&CLASS,

&SEC ) ;

ERROR at line 5:

ORA-00984: column not allowed here

**Correct one**

SQL> INSERT INTO STUDENT VALUES( &RNO,

'&NAME',

'&CLASS',

'&SEC') ;

SQL> SELECT \* FROM STUDENT;

All DML operations will not be saved immediately

When the user exited from the session all the DML operations will be saved.

To save the operations automatically immediately after the transaction, it should be committed;

Oracle provides following TCL commands to have the control over the transactions on the data base.

**COMMIT and ROLLBACK**

**1) COMMIT🡪** To commit the transaction, to make transaction as permanent

**2) ROLLBACK🡪**to undo the transaction

SQL> COMMIT;🡪 To commit the transaction

SQL> INSERT INTO STUDENT VALUES (1003 , 'RAMU' , 'C6' , 'A' );

SQL> INSERT INTO STUDENT VALUES (1005 , 'SHYAM' , 'C3' , 'B' )

SQL> SELECT \* FROM STUDENT;

ROLLNO NAME CLASS SECTION

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

1003 RAMU C6 A

1005 SHYAM C3 B

SQL> COMMIT;

Commit complete.

SQL> SELECT \* FROM STUDENT;

SQL> INSERT INTO STUDENT VALUES ( 1006, 'CHANDANA', 'C4', 'D') ;

SQL> INSERT INTO STUDENT VALUES ( 1007,'KRISHNA', 'C1', 'C') ;

SQL> SELECT \* FROM STUDENT;

ROLLNO NAME CLASS SECTION

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

1003 RAMU C6 A

1005 SHYAM C3 B

1006 CHANDANA C4 D

1007 KRISHNA C1 C

SQL> ROLLBACK;🡪 to undo transaction till recent commit

Rollback complete.

SQL> SELECT \* FROM STUDENT;

ROLLNO NAME CLASS SECTION

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

1003 RAMU C6 A

1005 SHYAM C3 B

Note : once committed we can not roll back the transaction

SQL> SET AUTOCOMMIT ON ; 🡪 To commit the transaction automatically

SQL> SET AUTOCOMMIT OFF ; 🡪 to off the auto commit

**DELETING RECORDS FROM THE TABLE**

**Syntax:**

DELETE FROM <table\_name> 🡪 to delete all records from the table

Eg:

SQL> DELETE FROM STUDENT;

SQL> SELECT \* FROM STUDENT;

no rows selected

**Deleting particular records**

Oracle provides the where clause to delete particular records. It will be covered as part of oracle clauses.

**TRUNCATE**

**TRUNCATE operation on the Table**

It is a table level DDL operation.

**TRUNCATE :**

1) It will drop and re-creates the table

2) Conditional data can not be deleted by using truncate ( i.e using where clause )

3) Truncate operation can not be roll backed.

4) Rollback segments will not be maintained by truncated

**Syntax :**

TRUCATE TABLE <table\_name>

SQL > TRUNCATE TABLE EMP ;

SQL > SELECT \* FROM EMP ;

No Records found

**Difference between DELETE AND TRUNCATE**

|  |  |
| --- | --- |
| **DELETE** | **TRUNCATE** |
| 1) It is a DML Operation | 1) It is a DDL Operation |
| 2) Conditional data can be deleted  i.e Where clause can be used | 2) Can not be |
| 3) Operation can be roll backed | 3) Operation can no be roll backed |
| 4) When the delete operation has been performed, the deleted records will be moved into Roll Back segments | 4) Roll back segments will not be used with this Operation |
| 5) It will not release the space, which is occupied by the deleted records | 5) It will release the entire space occupied by the table and re-creates the table at new location |

**MODIFYING RECORDS**

UPDATE🡪 to modify the records in a table

**Syntax :**

UPDATE <table\_name> SET Col1 = value, Col2=value, Col3=val

Eg:

SQL> SELECT \* FROM EMP\_TEST;

EMPNO ENAME SALARY DEPTNO

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

1001 MNRAO 5500 20

1002 ABCD 6500 10

1010 SIRISHA 20

1012 5500

1014 RAMESH 7800

1020 RAMA RAO 8500

1025

7 Rows selected.

SQL> UPDATE EMP\_TEST SET SALARY=5000 ;

7 rows updated.

SQL> SELECT \* FROM EMP\_TEST;

EMPNO ENAME SALARY DEPTNO

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

1001 MNRAO 5000 20

1002 ABCD 5000 10

1010 SIRISHA 5000 20

1012 5000

1014 RAMESH 5000

1020 RAMA RAO 5000

1025 5000

7 rows selected.

SQL> UPDATE EMP\_TEST SET SALARY=SALARY+150;

7 rows updated.

SQL> SELECT \* FROM EMP\_TEST;

EMPNO ENAME SALARY DEPTNO

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

1001 MNRAO 5150 20

1002 ABCD 5150 10

1010 SIRISHA 5150 20

1012 5150

1014 RAMESH 5150

1020 RAMA RAO 5150

1025 5150

7 rows selected.

**Modifying particular records**

Oracle provides the where clause to modify particular records based on some conditions. It will be covered as part of oracle clauses.

**OPERATORS**

**1) Arithmetic Operators**

**+** Addition

**-**  subtraction

**\***  Multiplication

**/**  Division

**2) Relational Operators**

These are used to compare the values

**>**  Greater than

**>=**  Greater than or equal to

**<**  Less than

**<=**  Less than or equal to

**=**  Equal to

**!=** or **<>**  Not equal to

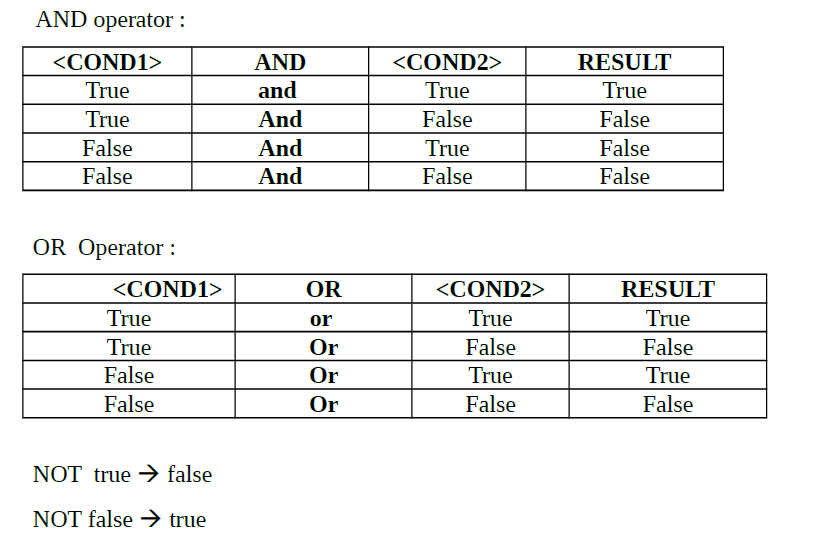
**3) Logical Operators**

These are used to compare the conditions

**and 🡪** returns true if both conditions are true other wise false

or 🡪 returns true if any one of the condition true otherwise false

not 🡪 returns the reverse value ( it is called negation operator )



**ORACLE CLAUSES**

**There are of four types clauses**

1) WHERE Clause

2) ORDER BY Clause

3) GROUP BY Clause

4) HAVING Clause

**These above said clauses works in association with operators**

**Where clause :**

It is a conditional clause. It works based on some conditions. It can be used with Select, Delete and Update statements.

**Select statement with where clause**

**Syntax:**

SELECT Col1, Col2, Col3, …… FROM <table\_name> WHERE <cond>

Here first priority will be given to where clause,

Result of the Select statement depends on where clause

If where clause returns true, then select will fetch the data from the table;

**Examples**

**To display details of employee whose number = 7839**

SQL> SELECT \* FROM EMP WHERE EMPNO=7839;

**To display details of employee whose number > =7500**

SQL > SELECT \* FROM EMP WHERE EMPNO>=7500

**To display details of employee whose name is SCOTT**

SQL>SELECT \* FROM EMP WHERE ENAME='SCOTT'

**Eg2:** to display details of employee whose JOB is MANAGER

SQL> SELECT \* FROM EMP WHERE JOB='MANAGER' ;

**Data in the table is a case sensitive but table name, column names and keywords are not case sensitive**

SQL> SELECT \* FROM EMP WHERE JOB='ANALYST'

SQL> SELECT \* FROM EMP WHERE JOB='analyst'

No rows selected

As the data in the table is UPPER CASE ( ANALYST ) but the data in the condition is LOWER CASE

(analyst )

**To display all clerks and managers**

SQL> SELECT \* FROM EMP WHERE JOB='CLERK' AND JOB='MANAGER' ;

**To display details of employee who is getting salary > = 5000**

SQL> SELECT \* FROM EMP WHERE SAL>=5000;

**To display details of employee who is getting salary < 3000**

SQL> SELECT \* FROM EMP WHERE SAL < 3000

**To display details of employee who is getting salary between 2000 and 4000**

SQL> SELECT \* FROM EMP WHERE SAL > = 3000 and SAL < =4000

**To display details of employees who are getting Commission**

SQL> SELECT \* FROM EMP WHERE COMM !=NULL

No rows selected ( here NULL should be used with operators )

We should write the following query

SELECT \* FROM EMP WHERE COMM **IS NOT** NULL

**To display the details of employees, who are working in deptno – 20**

SQL> SELECT \* FROM EMP WHERE DEPTNO=20;

**To display the details of employees who are joined earlier to 01-JAN-2000**

SQL > SELECT \* FROM EMP WHERE HIREDATE < '01-JAN-2000'

**To display details of employee whose experience is more than 10 years**

**Fetching data based on condition on multiple columns**

**To display details of 20th dept employee who is getting salary more than 3000**

SQL> SELECT \* FROM EMP WHERE DEPTNO=20 AND SAL>=3000 ;

**To display details of Managers working in deptno – 20**

SQL> SELECT \* FROM EMP WHERE JOB='MANAGER' AND DEPTNO=20

**Deleting Particular Records ( Conditional Operation )**

DELETE with WHERE clause

**Syntax :** DELETE FROM <table\_name> WHERE <condition>

**To delete all records from the table**

SQL> DELETE FROM EMP;

**To delete records of employees, who are working in deptno = 30**

SQL> DELETE FROM EMP WHERE DEPTNO=30 ;

**To delete records of employees who are joined earlier to 01-JAN-2000**

SQL>DELETE FROM EMP\_TEST WHERE HIREDATE< '01-JAN-2000'

**To delete records of employees, who are getting sal > = 5000 and sal <=8000 AND working in deptno=20**

SQL>DELETE FROM EMP WHERE DEPTNO=20 AND (SAL>=2000 AND SAL<=4000 )

**Modifying Particular Records (Conditional operation )**

UPDATE WITH WHERE CLAUSE

**Syntax :**

UPDATE <table\_name> SET Col1=value , Col2 = value WHERE <condition>

SQL> UPDATE EMP SET SAL=5000 ;

SQL > SELECT \* FROM EMP;

**Increment of Rs.250/- to all employees of the company**

SQL> UPDATE EMP SET SAL=SAL+250 ;

SQL > SELECT \* FROM EMP;

**Increment of Rs.500/- to MANAGERS, ANALYST**

SQL> UPDATE EMP SET SAL=SAL + 500 WHERE JOB=’MANAGER’ OR JOB=’ANALYST ;

**Increment of Rs.500/- to MANAGERS, ANALYST , who are working in deptno – 20**

SQL> UPDATE EMP SET SAL=SAL + 500 WHERE ( JOB='MANAGER' OR JOB='ANALYST' ) AND DEPTNO=20;

**Modify the JOB of employee as SALESMAN whose empno – 7566**

SQL> UPDATE EMP SET JOB='SALESMAN' WHERE EMPNO=7566 ;

**Increment of Rs.500/- to all employees, who are joined earlier to 01-JAN-2000**

SQL> UPDATE EMP SET SAL = SAL + 500 WHERE HIREDATE < '01-JAN-2000' ;

**SPECIAL OPERATORS OF ORACLE**

1) IN 2) NOT IN

3) BETWEEN 4) NOT BETWEEN

5) LIKE 6) NOT LIKE

7) IS NULL 8) IS NOT NULL

IN 🡪 List of values

BETWEEN 🡪 Range of Values

LIKE 🡪Searching for particular pattern

IS NULL 🡪 NULL values

**IN Operator**

**To display employees working departments – 10, 30 , 40 and 50**

SQL> SELECT \* FROM EMP WHERE DEPTNO IN (10, 20, 40, 50)

**To display employees working in departments except of 10, 30 , 40 and 50**

SQL> SELECT \* FROM EMP WHERE DEPTNO NOT IN (10, 20, 40, 50)

**To display details of MANAGERS , ANALYST AND CLERK**

SQL>SELECT \* FROM EMP WHERE JOB IN('MANAGER','ANALYST', 'CLERK') ;

**To display details of all employees except of MANAGERS , ANALYST AND CLERK**

SQL>SELECT \* FROM EMP WHERE JOB NOT IN('MANAGER','ANALYST', 'CLERK');

**BETWEEN Operator :**

It works on range of values.

It will consider the values including low value and high value ( **>= and <=** )

**To display details of employees whose salary in range of 2650 and 3200 (sal >=2650 and sal <=3200)**

Sql>SELECT \* FROM EMP WHERE SAL BETWEEN 2650 AND 3200

**NOT BETWEEN Operator**

It works on range of values. It will not consider the low value and high value ( < **and >**)

**To display details of employees whose salary not in the range of 2650 to 3200**

SELECT \* FROM EMP WHERE SAL NOT BETWEEN 2650 AND 3200

**LIKE Operator**

1) it is a searching pattern command

2) It woks in association with following chars

1) %

2) \_ **(** under score )

%--> represents zero or more no.of chars

\_--> represents single char

**To display all employees whose names are starting with ‘ S ‘**

SQL> SELECT \* FROM EMP WHERE ENAME LIKE 'S%' ;

**To display all employees whose names are ending with ‘ S ‘**

SQL> SELECT \* FROM EMP WHERE ENAME LIKE '%S' ;

**To display all employees whose name containing ‘ S ‘**

SQL> SELECT \* FROM EMP WHERE ENAME LIKE '%S%' ;

**To display all employees whose name containing ‘RAM ‘ ( sequence )**

SQL> SELECT \* FROM EMP WHERE ENAME LIKE '%RAM%' ;

**To display all employees whose name containing second char as ‘L ‘**

SQL>SELECT \* FROM EMP WHERE ENAME LIKE '\_L%'

**To display all employees who are joined in the month of ‘DEC’**

SQL> SELECT \* FROM EMP WHERE HIREDATE LIKE '%DEC%' ;

**IS NULL🡪 checks for null values**

**To display all employees who are not getting commission**

SQL> SELECT \* FROM EMP WHERE COMM IS NULL

**To display all employees who are getting commission**

SQL> SELECT \* FROM EMP WHERE COMM IS NOT NULL

Note: NULL should not be used with operators such as =, != , < and >

**ORDER BY Clause**

It sorts the data either in Ascending or in Descending order.

Default order is ascending order.

**Syntax :**

SELECT <\* / List of Cols > FROM <table\_name> ORDER BY <column\_name> [ ASC/DESC]

**To display the employee details based on salary in a ascending order**

SQL> SELECT \* FROM EMP ORDER BY SAL ;

**(or)**

SQL> SELECT \* FROM EMP ORDER BY SAL ASC ;

Since default order is an ascending, use of **ASC** is an optional

**To display the employee details based on salary in a descending order**

SQL> SELECT \* FROM EMP ORDER BY SAL DESC ;

**Use of WHERE and ORDER BY Clauses**

**Syntax :**

SELECT <\* / List of Cols > FROM <table\_name>

WHERE <condition>

ORDER BY <column\_name> [ ASC/DESC]

**Priority Of execution :**

WHERE

SELECT

ORDER BY

SQL> SELECT \* FROM EMP ORDER BY HIREDATE ;

Order by multiple columns;

If data matching in any two records, default sorting is based on next column.

Sorting based multiple columns.

Syntax:

SELECT <\* / List of Cols > FROM <table\_name> ORDER BY <col1>, <col2> [ ASC/DESC];

Eg:

Sql > select \* from emp order by sal, deptno;

Here sal column and deptno column both are ascending order.

Sql> select \* from emp order by sal, deptno desc;

Here sal column ascending and deptno column is descending order.

Sql> select \* from emp order by deptno, sal desc, comm desc ;

Using where and order by and also descending order.

SQL> select \* from emp where sal>=5000 and deptno in(10,30) and job='SALESMAN' order by deptno desc, sal desc, comm desc ;

**FUCNTIONS**

It performs required task and returns some value

Oracle provides two types of functions

1) Built – In Functions 2) User defined functions

**Built-in functions🡪** are pre-defined functions provided by the oracle library

**User defined functions🡪** these will be developed by the SQL developer ( will be covered in PL/ SQL )

**Built-in functions**

These are of two types

1) Single Row Functions 2) Group Functions

**Single Row Functions**

These are of five types

1) Numeric Functions

2) Character Functions

3) Date Functions

4) Miscellaneous Functions

**Numeric Functions :**

These functions works with numeric value and returns numeric value

DUAL 🡪 is a system table which contains only one row and one column.

1) ABS(N)🡪 Returns absolute value of N **(** +ve Value )

SQL> SELECT ABS(10.25) FROM DUAL ; 🡪 **10.25**

Abs() is used to get the difference between two values as positive value.

SQL> SELECT ABS(-10.25) FROM DUAL ;🡪 **10.25**

2) POWER( x, y ) 🡪 It returns a value x to the power of y ( **x, y** )

SQL> SELECT POWER(4,2) FROM DUAL ; 🡪 **16**

3) SQRT(N) 🡪 Returns Square Root value of N

SQL> SELECT SQRT(16) FROM DUAL 🡪 **4**

5) SIN(D) 🡪 Returns value SIN of D 🡪 D is in degrees

SQL> SELECT SIN(0) FROM DUAL 🡪 **0**

SQL> SELECT SIN(90) FROM DUAL 🡪 **0.893996664 ( it will be rounded to 1)**

6) COS(D) 🡪 Returns value COS of D 🡪 D is in degrees

SQL> SELECT COS(0) FROM DUAL 🡪 **1**

7) EXP(N) 🡪 Returns Exponential value of N

SQL> SELECT EXP(10) FROM DUAL

8) MOD( X, Y) 🡪 Returns remainder value

SQL> SELECT MOD(10,3) FROM DUAL 🡪 1

9) CEIL(N) 🡪 Returns a Value rounded to next integer

SQL> SELECT CEIL(10.6) FROM DUAL 🡪 11

SQL> SELECT CEIL(10.2) FROM DUAL 🡪 11

10) FLOOR (N) 🡪 Returns a value rounded to previous integer

SQL> SELECT FLOOR(10.2) FROM DUAL 🡪 10

SQL> SELECT FLOOR(10.9) FROM DUAL 🡪 10

11) ROUND(M, N) 🡪 Return a value rounded nearest integer

(or )

ROUND(M)

N 🡪 Number of digits after integer

SQL> SELECT ROUND(10.687) FROM DUAL 🡪 11

SQL> SELECT ROUND(10.687, 2 ) FROM DUAL 🡪 10.69

SQL> SELECT ROUND(10.1356) FROM DUAL 🡪 10

SQL> SELECT ROUND(10.1356, 2) FROM DUAL🡪 10.14

SQL> SELECT ROUND(10.5) FROM DUAL🡪 11

12) TRUNCATE(M, N) 🡪 It will truncate the data in the decimal part. If N is not supplied then it

returns value M, if N values is supplied then truncates data after N no.of digits in decimal part.

SQL> SELECT TRUNC(10.736) FROM DUAL ; 🡪 10

SQL> SELECT TRUNC(10.736, 2) FROM DUAL ;🡪 10.73

SQL> SELECT **ROUND**(10.736, 2) FROM DUAL ;🡪 10.74

**Character Functions :**

These functions works with character data and returns character of numeric data

1) LOWER ( String ) 🡪 Returns lower case String

SQL> SELECT LOWER('ORACLE') FROM DUAL 🡪 oracle

SQL> SELECT LOWER('oracle') FROM DUAL 🡪 oracle

2) UPPER(String) 🡪 Returns upper case string.

SQL> SELECT UPPER (‘oracle') FROM DUAL 🡪 ORACLE

SQL> SELECT UPPER ('ORACLE') FROM DUAL 🡪 ORACLE

3) INITCAP( String ) 🡪 Returns String with Initial Letter as upper case and remaining lower case

SQL> SELECT INITCAP('oracle') FROM DUAL 🡪 Oracle

SQL> SELECT INITCAP('oracle corp') FROM DUAL 🡪 Oracle Corp

4) LENGTH ( String ) 🡪 Returns length of the string

SQL> SELECT LENGTH('ORACLE') FROM DUAL 🡪 6

5) ASCII(Char) 🡪 Returns ASCII value of String

SQL> SELECT ASCII('A') FROM DUAL 🡪 65

SQL> SELECT ASCII('AB') FROM DUAL 🡪 65

6) LTRIM(String ) 🡪 It removes the all leading spaces

LTRIM(String, SubString ) 🡪 it trims sub string from the beginning

SQL> SELECT LTRIM('abcxyz', 'abc') FROM DUAL 🡪 xyz

7) RTRIM( String ) 🡪 It removes the all trialing spaces

RTRIM(String, SubString ) 🡪 it trims sub string from the end

SQL> SELECT RTRIM('abcxyz', 'xyz') FROM DUAL🡪 abc

TRIM(String ) 🡪 trimming spaces on both sides.

8) TRANSLATE(String, FromChar, ToChar );  to translate the String FormChar to ToChar

SQL> SELECT TRANSLATE('ABBCCADDBB', 'B', 'Z') FROM DUAL 🡪 AZZCCADDZZ

SQL> SELECT TRANSLATE('ABBCCADDBB', 'ABC', 'XYZ') FROM DUAL 🡪

XYYZZXDDYY

Replacements are A 🡪 X, B🡪Y, C🡪 Z

9) REPLACE( MainString, ReplacedString, ReplacingString );

SQL> SELECT REPLACE ('abcxyz', 'abc', 'def') FROM DUAL 🡪 defxyz

It searches for the sequence of chars, then replacing .

10) SUBSTR(String, StartPos, Nchars)  returns No.of chars from the position to end of the string

SQL> SELECT SUBSTR('abcdefghij’, 4) FROM DUAL 🡪 defghij

(returns chars from the position to end of the string )

SQL>SELECT SUBSTR('abcdefghi',4,2) FROM DUAL 🡪 de

**To terminate last char.**

SELECT SUBSTR('hello',1,length('hello')-1) FROM DUAL;

SQL>SELECT SUBSTR('abcdefghi', -1, 2) FROM DUAL 🡪 i

SQL>SELECT SUBSTR('abcdefghi',-4,3) FROM DUAL 🡪 fgh

11) LPAD(String/Number, N , Char ) 🡪 To fill the left empty spaces of output with require char

String/Number : output

N 🡪 no.of spaces in the output

Char🡪 filling char of empty spaces

SQL> SELECT LPAD(1234, 7, '\*') FROM DUAL 🡪 \*\*\*1234

12) RPAD(String/Number, N , Char )  To fill the right empty spaces of output with require char

SQL> SELECT LPAD(1234,7,'\*') FROM DUAL🡪 1234\*\*\*

**Date Functions**

These functions works with data value and returns numeric data

1) MONTHS\_BETWEEN 2) ADD\_MONTHS 3) NEXTDAY 4) LAST\_DAY

**MONTHS\_BETWEEN( d1, d2 )**  Returns no.of months between d1 and d2 ( d1-d2)

SQL> SELECT MONTHS\_BETWEEN( '30-MAR-2005', '30-MAR-2004') FROM DUAL  12

SQL> SELECT MONTHS\_BETWEEN( '30-MAR-2004', '30-MAR-2005') FROM DUAL  -12

**ADD\_MONTHS(date, +/- N )**  it returns a new date by adding / subtracting N months

SQL> SELECT ADD\_MONTHS( '30-OCT-2004', 4) FROM DUAL  28-FEB-05

**NEXT\_DAY(date, WeekDay)**  It returns the day on the immediate WeekDay specified

SQL > SELECT NEXT\_DAY( '31-DEC-2004', 'wed') FROM DUAL  05-JAN-05

LAST\_DAY(Date)  It returns last day in the month of the specified date

SQL> SELECT LAST\_DAY( '10-FEB-2010') FROM DUAL  28-FEB-10

**Date Formats :**

DD 🡪 01 to 31

DDD 🡪 01 to 366

DY 🡪 MON to SUN

DAY🡪 Monday to Sunday

MM 🡪 01 to 12

MON 🡪 JAN to DEC

MONTH 🡪 January TO December

YY 🡪 00 to 99 year

YYYY 🡪 00 to 9999

HH 🡪 1 to 12 hrs

HH24 🡪 00 to 23 hrs

MI 🡪 00 to 59 min

SS 🡪00 to 59 sec

WW 🡪 Weak Of The Year ( 1 YEAR = 52 WEEKS )

**Date conversion**

It can be converted from **date to char / char to date**

Functions are 1) TO\_CHAR 2) TO\_DATE

TO\_CHAR( date , DateFormat)  It will generate the in DateFormat Specified

SQL> SELECT TO\_CHAR(SYSDATE, 'DD-MM-YYYY') FROM DUAL

25-03-2011

SQL> SELECT TO\_CHAR(SYSDATE, 'DD-MON-YYYY') FROM DUAL

25-MAR-2011

SQL> SELECT TO\_CHAR(SYSDATE, 'DD-MON-YYYY HH24:MI:SS') FROM DUAL;

25-MAR-2011 04:45:27

SQL> SELECT TO\_CHAR(SYSDATE, 'HH:MI:SS') FROM DUAL

04:46:41

SQL> SELECT TO\_CHAR(SYSDATE, 'HH24:MI:SS') FROM DUAL

16:46:41

SQL> SELECT TO\_CHAR(SYSDATE, 'WW') FROM DUAL

12

( Prepared on March 25, 2011

SQL> SELECT TO\_CHAR(SYSDATE, 'DD/MM/YYYY') FROM DUAL

25/03/2011

**TO\_DATE** 🡪 To convert from Char format to oracle date format

**Syntax :**

TO\_DATE( Varchar, DateFormat);

SQL> SELECT TO\_DATE('15-JUN-2010', 'DD-MON-YYYY') FROM DUAL ;

15-JUN-10

SQL> SELECT TO\_DATE('JUNE/20/2005', 'MONTH/DD/YYYY') FROM DUAL ;

20-JUN-05

SQL> SELECT TO\_DATE('12', 'DD') FROM DUAL ;

12-MAR-11

**Miscellaneous Functions**

1) USER 2) UID 3) LEAST 4) GREATEST 5) NVL 6) NVL2 7) NULLIF

8) COALESCE

SQL> SELECT **USER** FROM DUAL  SCOTT

SQL> SELECT **UID** FROM DUAL  239

SQL> SELECT **LEAST**(20, -10, 30 , 50 ) FROM DUAL  **-10**

SQL> SELECT LEAST('A', 'B','C', 'D') FROM DUAL  **A**

SQL> SELECT LEAST('A', 'B','C', 'D', 6) FROM DUAL  6

**NVL( Column, Value )** 🡪 It will replace the NULL value in the Column with required **Value** specified

SQL> SELECT \* FROM EMP  It shows the empty for NULL Columns

SQL> SELECT empno, ename, sal, comm FROM EMP

EMPNO ENAME SAL comm

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

7839 KING 5200

7698 BLAKE 3050

7782 CLARK 2650

7566 JONES 3175

7654 MARTIN 1450 1400

7499 ALLEN 1800 300

7844 TURNER 1700

SQL> SELECT empno, ename, sal, NVL(comm,0) FROM EMP

EMPNO ENAME SAL NVL(COMM,0)

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

7839 KING 5200 0

7698 BLAKE 3050 0

7782 CLARK 2650 0

7566 JONES 3175 0

7654 MARTIN 1450 1400

7499 ALLEN 1800 300

SQL> SELECT sal\*NULL FROM EMP  All shows with NULL value

Any mathematical expression, which contains NULL value will be evaluated to NULL

SQL> SELECT sal + comm FROM EMP  result shows NULL value for the records, which has COMM as NULL

SQL> SELECT sal + nvl (comm,0) FROM EMP 

**NVL2(Column, exp1, exp2 )**  It returns exp1 value if Column value is not null else it returns exp2 value

SQL> SELECT NVL2(comm, sal+comm, sal) FROM EMP

**NULLIF( Expr1, Expr2 )**  It Returns null if both the expressions are same else it returns Expr1

value

SQL> SELECT NULLIF(10\*50, 250\*2) FROM DUAL  returns all NULL values

SQL> SELECT NULLIF(10\*5, 250\*2) FROM DUAL  returns 1st expr value ( 50 )

**COALESCE( Value1, Value2, Value3 )**  It returns 1st Not NULL value

SQL> SELECT COALESCE(25\*10, 3\*6, 4\*5) FROM DUAL  returns 250

SQL> SELECT COALESCE(25\*NULL, 3\*6, 4\*NULL) FROM DUAL  returns 18

**GROUP**

It is a collection of records

A table can be considered as group or

Table can be divided into no.of groups

**GROUP FUNCTIONS**

These functions returns numeric value

1) COUNT(\*)  returns no.of records in a group

2) COUNT(Col)  returns No.of Not NULL Columns

3) SUM( Col )  returns sum of the column values

4) MIN( Col )  returns min value of column

5) MAX(Col)  returns max value of column

6) AVG(Col)  returns average value of the column

7) STDDEV(Col)  returns standard deviation of Column

8) VARIANCE(Col)  returns variance of Column

SQL > SELECT COUNT(\*) FROM EMP 🡪 14 or any value

SQL> SELECT COUNT(COMM) FROM EMP 🡪 4 or any value

**GROUP BY Clause 🡪** This clause is used to grouping the Rows of the table

**Syntax :**

SELECT <Column\_List> , < group\_functions > FROM < table\_Name>

WHERE <Condition>

GROUP BY < group by expr >

ORDER By < Column or gourp\_function >

ASC/DESC

**Grouping based on deptno**

SQL> SELECT DEPTNO, SUM(sal) SAL\_SAM, COUNT(\*) TOTAL\_REC FROM EMP GROUP BY DEPTNO

DEPTNO SAL\_SAM TOTAL\_REC

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

10 8750 3

20 10875 5

30 9400 6

Grouping based on deptno and order based on sum of salaries

SQL> SELECT DEPTNO, SUM(sal) SAL\_SAM, COUNT(\*) TOTAL\_REC FROM EMP

GROUP BY DEPTNO

ORDER BY SUM(SAL) DESC ;

SQL> SELECT JOB, SUM(sal) SAL\_SAM, MIN(SAL) MIN\_SAL FROM EMP

WHERE JOB IN ( 'CLERK', 'MANAGER', 'ANALYST')

GROUP BY JOB

ORDER BY SUM(SAL) DESC ;

SQL> SELECT DEPTNO, SUM(sal) SAL\_SAM, COUNT(\*) TOTAL\_REC FROM EMP

WHERE COUNT(\*) >=3

GROUP BY DEPTNO ;

Error since Group Functions are not allowed in where clause.

**HAVING Clause**

This clause is used to apply the condition on GROUP FUCTIONS

**Syntax :**

SELECT <Column\_list>, <Group\_functions > FROM < Table\_name>

WHERE <Condition>

GROUP BY <group by expression >

HAVING <Condition >

ORDER BY < Column/ group function >

ASC/DESC

**Order of execution :**

WHERE

SELECT

GROUP BY

HAVING

ORDER BY

To divide the table into groups based on deptno and to display the groups if no.of records in group are greater than or equal to 5

SQL> SELECT DEPTNO, SUM(sal) SAL\_SAM, AVG(SAL) AVG\_SAL, COUNT(\*) TOTAL\_REC FROM EMP GROUP BY DEPTNO HAVING COUNT(\*) >=5 ;

SQL> SELECT JOB, SUM(sal) SAL\_SAM, AVG(SAL) AVG\_SAL, COUNT(\*) TOTAL\_REC FROM EMP GROUP BY JOB HAVING COUNT(\*) >=2 ;

**SET OPERATORS**

* These operators are used to combine the outputs of two or more queries
* Result of the queries, which will be joined by SET OPERATORS should have the same no.of columns and Same data type

**These operators are**

1) UNION ALL 2) UNION

3) INTERSECT 4) MINUS

1) **UNION ALL** 🡪 It will combine the outputs of two or more queries

SQL> SELECT DEPTNO FROM EMP

UNION ALL

SELECT DEPTNO FROM DEPT

18 rows selected.

SQL> SELECT DEPTNO FROM EMP WHERE DEPTNO=10

UNION ALL

SELECT DEPTNO FROM DEPT WHERE DEPTNO=20

4 Rows selected

SQL> SELECT JOB FROM EMP WHERE DEPTNO=10

UNION ALL

SELECT JOB FROM EMP WHERE DEPTNO=20

8 Rows selected

2) **UNION** 🡪 It will combine the outputs of two or more queries and filters the duplicate

SQL> SELECT DEPTNO FROM EMP

UNION

SELECT DEPTNO FROM DEPT

4 rows selected.

SQL> SELECT DEPTNO FROM EMP WHERE DEPTNO=10

UNION

SELECT DEPTNO FROM DEPT WHERE DEPTNO=20

2 Rows selected

SQL> SELECT JOB FROM EMP WHERE DEPTNO=10

UNION ALL

SELECT JOB FROM EMP WHERE DEPTNO=20

4 Rows selected

**INTERSECT** 🡪 It shows common output of two or more queries

SQL> SELECT DEPTNO FROM EMP

INTERSECT

SELECT DEPTNO FROM DEPT

4 rows selected.

SQL> SELECT DEPTNO FROM EMP WHERE DEPTNO=10

INTERSECT

SELECT DEPTNO FROM EMP WHERE DEPTNO=20

No Rows selected

SQL> SELECT JOB FROM EMP WHERE DEPTNO=10

INTERSECT

SELECT JOB FROM EMP WHERE DEPTNO=20

2 Rows Selected

**MINUS** 🡪 It will display the out put unique first query

SQL> SELECT DEPTNO FROM EMP

MINUS

SELECT DEPTNO FROM DEPT

No rows selected.

SQL> SELECT DEPTNO FROM EMP WHERE DEPTNO=10

INTERSECT

SELECT DEPTNO FROM EMP WHERE DEPTNO=20

1 Row selected

SQL> SELECT JOB FROM EMP WHERE DEPTNO=10

MINUS

SELECT JOB FROM EMP WHERE DEPTNO=20

1 Row selected

**SUB QUERIES**

It is a nested SELECT statement ( SELECT with in another SELECT )

Sub query is the select statement embedded in clause ( where or having clause ) of another select statement

**Types of Sub Queries**

1) Single Row Sub Query

2) Multi Row Sub Query

**Single Row Sub Query :** A sub query, which returns only one value to outer query

**To get the dept name of employee whose empno = 7782**

SQL>SELECT DNAME FROM DEPT WHERE DEPTNO=(SELECT DEPTNO FROM EMP WHERE

EMPNO=7782)

**To display names of employees, who job is same as SMITH**

SQL> SELECT ENAME FROM EMP WHERE JOB = (SELECT JOB FROM EMP WHERE

ENAME='SMITH') ;

**To display employees, who are getting the salary more than or equal to the max salary of 30th Dept**

SQL> SELECT ENAME FROM EMP WHERE SAL>=(SELECT MAX(SAL) FROM EMP WHERE

DEPTNO=30)

**Multi Row Sub Queries :** It returns more than one value to the outer query

Following operators are used with multi row sub query.

1) IN 2) ANY 3) ALL 4) EXISTS

**IN🡪 List of values**

**To display all employees who are mangers for other employees**

SQL> SELECT \* FROM EMP WHERE EMPNO IN ( SELECT DISTINCT MGR FROM EMP)

**To display employee details from emp table, if the employee is not a managers of other employees**

SQL> SELECT \* FROM EMP WHERE EMPNO IN ( SELECT EMPNO FROM EMP MINUS SELECT

DISTINCT MGR FROM EMP)

ANY🡪 List of values

>ANY , < ANY

**> ANY ( List of values ) :** It picks the min value form list of values from table

To display all employees data from emp table if sal of employee is more than min sal of 10th dept

SQL> SELECT \* FROM EMP WHERE SAL > ( SELECT MAX(SAL) FROM EMP WHERE DEPTNO=10)

( or )

SQL> SELECT \* FROM EMP WHERE SAL >ANY ( SELECT SAL FROM EMP WHERE DEPTNO=10)

< ANY **( List of values ) :** It picks the max value form list of values from table

**To display employee details if sal of employee less than max sal of 20th dept**

SQL> SELECT \* FROM EMP WHERE SAL < ( SELECT MIN(SAL) FROM EMP WHERE

DEPTNO=20)

**( or )**

SQL > SELECT \* FROM EMP WHERE SAL <ANY ( SELECT SAL FROM EMP WHERE DEPTNO=20)

**> ALL** ( list of values ) : It picks the max value from the list of values and display all vales from table which are more than max value from list .

To display details employees, if sal of employee is more than max sal of 20th dept

SQL> SELECT \* FROM EMP WHERE SAL > ( SELECT MAX(SAL) FROM EMP WHERE

DEPTNO=20)

**( or )**

SQL> SELECT \* FROM EMP WHERE SAL >ALL ( SELECT SAL FROM EMP WHERE DEPTNO=20)

**<ALL ( List of values )**  It picks the min value form list of values and display all values from table which

are less than min value in list.

**To display details of employees from emp table, if salary of employee is less than min salary of 10th**

**dept**

SQL> SELECT \* FROM EMP WHERE SAL < ( SELECT MIN(SAL) FROM EMP WHERE

DEPTNO=10)

**( or )**

SQL> SELECT \* FROM EMP WHERE SAL < ALL( SELECT MIN(SAL) FROM EMP WHERE

DEPTNO=10)

**exists🡪** It is a Boolean operator returns either success or failure of sub query

**To display all the MANAGERS of emp table, if there are MANAGERS presents**

SQL> SELECT \* FROM EMP WHERE JOB='MANAGER' AND EXISTS ( SELECT COUNT(\*) FROM

EMP WHERE JOB='MANAGER' GROUP BY JOB )

**Co-Related Sub Queries :** If a sub-query is evaluated or executed repeatedly for each every record retrieved by the outer query that sub-query is called as Co-Related sub-query.

In sub-queries, outer-query depends on inner-query where as in co-related sub-queries inner query depends

on outer-query. If outer query has retrieved ‘N’ records inner query has to execute for ‘N’ times.

To display details of employees if sal of employee is more than average salary of his job ?

SQL > SELECT E.EMPNO, E.ENAME, E.JOB, E.SAL FROM EMP E WHERE E.SAL>(SELECT AVG(SAL) FROM EMP WHERE JOB=E.JOB )

**Creation of a table from other tables with data / with out data**

SQL> CREATE TABLE TEST AS SELECT \* FROM EMP-🡪 it creates test table same as emp and imports data into test

SQL > SELECT \* FROM TEST ;

16 Rows selected

SQL> CREATE TABLE TEST1 AS SELECT \* FROM EMP WHERE 5=6 ;

SQL > SELECT \* FROM TEST1;

No Records

SQL > CREATE TABLE TEST2 AS SELECT EMPNO, ENAME, SAL,

**E.DEPTNO, DNAME, LOC FROM EMP E, DEPT**

**CONSTRAINTS**

These are used to set some conditions on table columns

**1) UNIQUE 🡪** It will not allow duplicate values but allows NULL values

**2) NOT NULL 🡪**It will not allow NULL values but allows duplicate values

**3) PRIMARY KEY 🡪**It is a UNIQUE, NOT NULL and INDEX

**4) CHECK 🡪** used for data validation

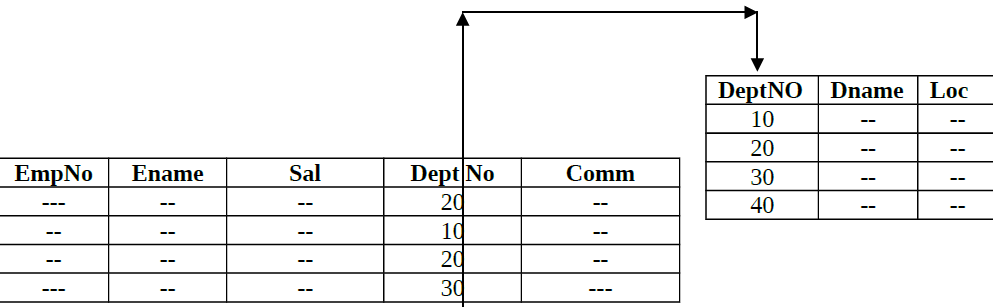
**5) FOREIGN KEY 🡪** It is used to establish the parent and child relationship (or)

Master and Detail relationship between the tables

Foreign key constraints of child table linked to primary key or unique constraint column master table.

Foreign key allows duplicate and null values.

It allows values which are in primary key or unique constraint Column of master table.



The above constraints can be defined at

1) Column Level and 2) Table Level

**Column Level :**

Constraints can be defined along with column at the time of creating table. Only one

Column can be defined as Primary Key when defining at the column level. Composite

Primary Key can not be defined at the Column Level.

**Table Level :**

Constraints can be defined at the end after defining all columns of the table, while creating

Table.

We can add the constraints on the already existing table, those are called table level

Composite Primary Key ( More than one primary key ) is allowed in the table level

Definition

**Defining Constraints at Column Level**

**Syntax :**

CREATE TABLE <table\_name> (

Col1 datatype(size) CONSTRAINT <const\_name> CONSTRAINT\_TYPE,

Col2 datatype(size) CONSTRAINT <const\_name> CONSTRAINT\_TYPE,

-------,

----- );

Eg:

**CREATE TABLE EMP\_TEST**(

empno number(10) **CONSTRAINT** CON\_EMPNO\_PK **PRIMARY KEY,**

name VARCHAR2(30) **CONSTRAINT** CON\_ENAME\_NN **NOT NULL**,

sal number(10) **CONSTRAINT** CON\_SAL\_CK **CHECK**(sal>=5000),

deptno number(10) **CONSTRAINT** CON\_DEPTNO\_FK **REFERENCES** DEPT(deptno) **ON DELETE CASCADE** ) ;

**Data Dictionary view for constraints**  it maintains constraints information

1) USER\_CONSTRAINTS

2) USER\_CONS\_COLUMN

SQL> DESC USER\_CONSTRAINTS;

SQL> DESC USER\_CONS\_COLUMN ;

SQL> SELECT TABLE\_NAME,

CONSTRAINT\_NAME,

CONSTRAINT\_TYPE

FROM USER\_CONSTRAINTS

WHERE TABLE\_NAME='EMP\_TEST' ;

**TABLE\_NAME CONSTRAINT\_NAME C**

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

EMP\_TEST CON\_ENAME\_NN C

EMP\_TEST CON\_SAL\_CK C

EMP\_TEST CON\_EMPNO\_PK P

EMP\_TEST CON\_DEPTNO\_FK R

TABLE\_NAME ------------COLUMN\_NAME -------------CONSTRAINT\_NAME

EMP\_TEST DEPTNO CON\_DEPTNO\_FK

EMP\_TEST EMPNO CON\_EMPNO\_PK

EMP\_TEST ENAME CON\_ENAME\_NN

EMP\_TEST SAL CON\_SAL\_CK

**Advantage of using ON DELETE CASCADE :**

If the master record deleted from the Detailed / Parent table then, then dependent records in

Master /Child table will also be deleted.

Insert the records into tables and test them

**Defining Constraints at table level**

**Syntax :**

CREATE TABLE <table\_name> ( Coli1 datatype(size), Col2 datatype(size), …., …., ……,

CONSTRAINT <constraint\_name> CONSTRAINT\_TYPE

( Col\_name),

CONSTRAINT <constraint\_name> CONSTRAINT\_TYPE

(Col\_name),

----------------,

------------------ ) ;

Eg:

SQL> CREATE TABLE DEPT\_TEST( deptno number(10), dname varchar2(20), loc varchar2(30),

CONSTRAINT CON\_DEPTNO\_PK PRIMARY KEY(deptno), CONSTRAINT CON\_DEPTNO\_UQ UNIQUE(dname) ) ;

**Nameless constraints**

CREATE TABLE <table\_name> ( Coli1 datatype(size), Col2 datatype(size), …., …., ……,

CONSTRAINT\_TYPE ( Col\_name),

CONSTRAINT\_TYPE (Col\_name),

-------------------------------------------,

------------------------------------------- ) ;

Eg:

CREATE TABLE DEPT\_TEST( deptno number(10), dname varchar2(20), loc varchar2(30),

PRIMARY KEY(deptno), UNIQUE(dname) ) ;

**Adding Constraints on existing table**

**Syntax :**

**Single constraint**

ALTER TABLE <table\_name> ADD CONSTRAINT <constraint\_name> CONSTRAINT\_TYPE (col\_name)

**Multiple Constraints:**

ALTER TABLE <table\_name> ADD ( CONSTRAINT <constraint\_name> CONSTRAINT\_TYPE

(col\_name),

CONSTRAINT <constraint\_name> CONSTRAINT\_TYPE

(col\_name)

---------------------------------------------------------------------------,

--------------------------------------------------------------------------- ) ;

**Create the table without any constraints**

SQL> CREATE TABLE DEPT\_TEST( deptno number(10), dname varchar2(20), loc varchar2(30) ) ;

**Define the constraints on the table**

SQL> ALTER TABLE DEPT\_TEST ADD ( CONSTRAINT CON\_DEPTNO\_PK PRIMARY KEY (deptno), CONSTRAINT CON\_DNAME\_UQ UNIQUE (dname) ) ;

**Adding name less constraints to table**

ALTER TABLE <table\_name> ADD ( CONSTRAINT\_TYPE (col\_name),

CONSTRAINT\_TYPE (col\_name),

-------------------------------------------,

------------------------------------------- ) ;

Eg:

ALTER TABLE DEPT\_TEST ADD ( PRIMARY KEY (deptno),

UNIQUE (dname) ) ;

**Defining Composite Primary Key:**

If two or more Primary Keys are defined on the same table, then it is called as composite primary key

SQL> CREATE TABLE DEPT\_TEST ( DEPTNO NUMBER(10), DNAME VARCHAR2(20), LOC VARCHAR2(30), PRIMARY KEY(DEPTNO, DNAME) );

When composite primary key has been defined, while inserting value into the table at least one of the primary key columns value should not be same as with already existing record.

Eg :

In the above DEPT\_TEST table deptno and dname both the values are matching with already

existing record then it will not be inserted into the table. At least either deptno or dname value differing with existing record then insertion is allowed.

**Disabling constraints**

**Syntax :**

**ALTER TABLE <table\_name> DISABLE CONSTRAINT <constraint\_name >**

SQL> CREATE TABLE DEPT\_TEST ( deptno number(10), dname varchar2(20), loc varchar2(30),

CONSTRAINT CON\_DEPTNO\_PK PRIMARY KEY(deptno),

CONSTRAINT CON\_DEPTNO\_UQ UNIQUE(dname) ) ;

SQL> ALTER TABLE DEPT\_TEST DISABLE CONSTRAINT CON\_DEPTNO\_PK ;

**Note :** Primary key constraint of master table can not be disabled with out disabling foreign key constraint on child table

**Enabling constraints**

**Syntax :**

ALTER TABLE <table\_name> ENABLE CONSTRAINT <constraint\_name >

SQL> ALTER TABLE DEPT\_TEST ENABLE CONSTRAINT CON\_DEPTNO\_PK ;

**Note :** 1) if data in the table violating the constraints, then constraints can not be enabled

2) Foreign key constraint of child table cannot be enabled without enabling Primary Key constraint on the master table.

**Dropping Constraints**

**Syntax :**

ALTER TABLE <table\_name> DROP CONSTRAINT <constraint\_name> ;

SQL> ALTER TABLE DEPT\_TEST DROP CONSTRAINT CON\_DEPTNO\_UQ ;

**Note :** Primary Key constraint of master table can not be dropped with out dropping foreign key constraint of child table .

**Dropping Primary Key constraint of master table & foreign key constraint of child table simultaneously**

**Syntax :**

ALTER TABLE <master\_table\_name> DROP PRIMARY KEY CASCADE

SQL> ALTER TABLE DEPT\_TEST DROP PRIMARY KEY CASCADE

**Self referential integrity**

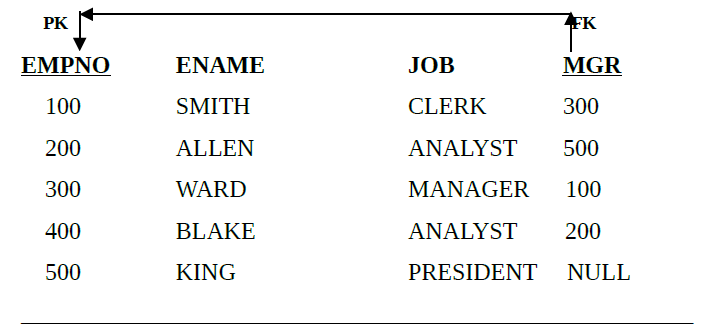
Primary key and Foreign key both will be in the same table and foreign key depends on the primary Key

SQL> CREATE TABLE EMP\_TEST ( EMPNO NUMBER(5) PRIMARY KEY,

ENAME VARCHAR2(10),

JOB VARCHAR2(10),

MGR NUMBER(9) REFERENCES EMP\_TEST(EMPNO) ) ;



Note :

1) First insert the data into EMPNO, ENAME, JOB COLUMNS

2) update MGR Column, the values available in the EMPNO columns can be inserted into MGR column as it is a foreign key referring to primary key column EMPNO.

**JOINS**

Join is a one type of query that combines the rows from two or more table

**These are**

1) Cross Join 2) Equi Join (or) Inner Join 3) Outer Join

4) Self Join 5) Non-Equi Join

**Cross Join :** It joins every row in the 1st table with every row in the 2nd table.

Output of Cross Join is called as Cartesian product

**Syntax :**

Select table1.Col1, table1.Col2, -------------, Table2.Col1, Table2.Col2, -------- from table1, table2

**( OR )**

**Oracle 9i and higher versions**

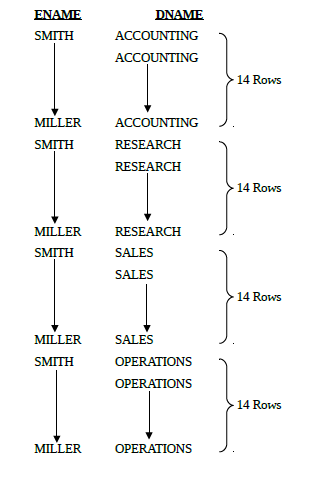
SELECT table1.Col1, table1.Col2, -------------, Table2.Col1, Table2.Col2, -------- FROM table1 CROSS

JOIN table2

If same column presents in the both tables then column should be referred with table name ( table.column)

Eg:

SQL> SELECT ENAME, DNAME FROM EMP, DEPT



56 Rows Selected

**Reverse**

SQL> SELECT ENAME, DNAME FROM EMP CROSS JOIN DEPT;

Cross Join will take column value of 1st row in the 2nd table and joins with all rows of 1st table

Next time it will take column value of 2nd row in the 2nd table and joins with all rows of 1st table

Next time it will take column value of 3rd row in the 2nd table and joins with all rows of 1st table

It will repeat the same process till end of rows in the second table.

**EQUI JOIN** or **INNER JOIN** :

If the rows from two or more tables joined based on equality condition then, it is called as **EQUI JOIN.**

It is join based equal condition

It will fetch the matched records from the both tables.

To obtain equi-join between table they must have common columns

**Syntax :**

Select table1.Col1, table1.Col2, -------------, Table2.Col1, Table2.Col2, -------- from table1, table2 Where table1.Column = table2.Column

**( OR )**

**Oracle 9i and higher versions**

SELECT table1.Col1, table1.Col2, -------------, Table2.Col1, Table2.Col2, -------- FROM table1 INNER JOIN table2 on table1.column = table2.column

**To display ename, dname of all employees of emp table.**

SQL> SELECT ENAME, DNAME FROM EMP, DEPT WHERE EMP.DEPTNO=DEPT.DEPTNO ;

ENAME DNAME

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

SMITH RESEARCH

ALLEN SALES

WARD SALES

JONES RESEARCH

MARTIN SALES

BLAKE SALES

CLARK ACCOUNTING

SCOTT RESEARCH

KING ACCOUNTING

TURNER SALES

ADAMS RESEARCH

JAMES SALES

FORD RESEARCH

MILLER ACCOUNTING

14 Rows selected.

**To display empno, ename, dpetno, dname and loc of all employees of emp table.**

SQL> SELECT EMPNO, ENAME, EMP.DEPTNO EMPDNO, DEPT.DEPTNO DEPTDNO, DNAME, LOC FROM EMP, DEPT WHERE EMP.DEPTNO=DEPT.DEPTNO ;

**( OR )**

SQL> SELECT EMPNO, ENAME, EMP.DEPTNO EMPDNO,DEPT.DEPTNO DEPTDNO, DNAME, LOC FROM EMP INNER JOIN DEPT ON EMP.DEPTNO=DEPT.DEPTNO ;

**Equi Join will display all matched records only.**

**To dispaly empno, ename of emp table and all cloumns of dept table**

SQL> SELECT EMPNO, ENAME, DEPT.\* FROM EMP, DEPT

**To dispaly all cloumns of emp and dept table**

SQL> SELECT EMP.\*, DEPT.\* FROM EMP, DEPT

**OUTER JOINS:**

1) LEFT OUTER JOIN

2) RIGHT OUTER JOIN

3) FULL OUTER JOIN

**LEFT OUTER JOIN**

**Syntax :**

Select table1.Col1, table1.Col2, -----------, Table2.Col1, Table2.Col2, -------- from table1 LEFT OUTER JOIN table2 On table1.Column = table2.Column

LEFT OUTER JOIN gives all records from left side table and matched records from the right side table.

And unmatched records null values ( right side )

.Eg:

SQL > SELECT EMPNO, ENAME, EMP.DEPTNO EMPDNO, DEPT.DEPTNO DEPTDNO, DNAME, LOC FROM EMP **LEFT OUTER JOIN** DEPT ON EMP.DEPTNO=DEPT.DEPTNO ;

The above query displays the matched records from **EMP** and **DEPT** table and unmatched records from

**EMP** table

**RIGHT OUTER JOIN**

RIGHT OUTER JOIN gives all records from right side table and matched records from the left side table

and unmatched records are null values ( left side )

SQL > SELECT EMPNO, ENAME,EMP.DEPTNO EMPDNO, DEPT.DEPTNO DEPTDNO, DNAME, LOC FROM EMP RIGHT OUTER JOIN DEPT ON EMP.DEPTNO=DEPT.DEPTNO;

The above query displays the matched records from **EMP** and **DEPT** table and unmatched records from **DEPT** table

**FULL OUTER JOIN**

It give all matched and unmatched records from both tables, if any unmatched records are null value.

SQL > SELECT EMPNO, ENAME,EMP.DEPTNO EMPDNO, DEPT.DEPTNO DEPTDNO, DNAME, LOC FROM EMP FULL OUTER JOIN DEPT ON EMP.DEPTNO=DEPT.DEPTNO;

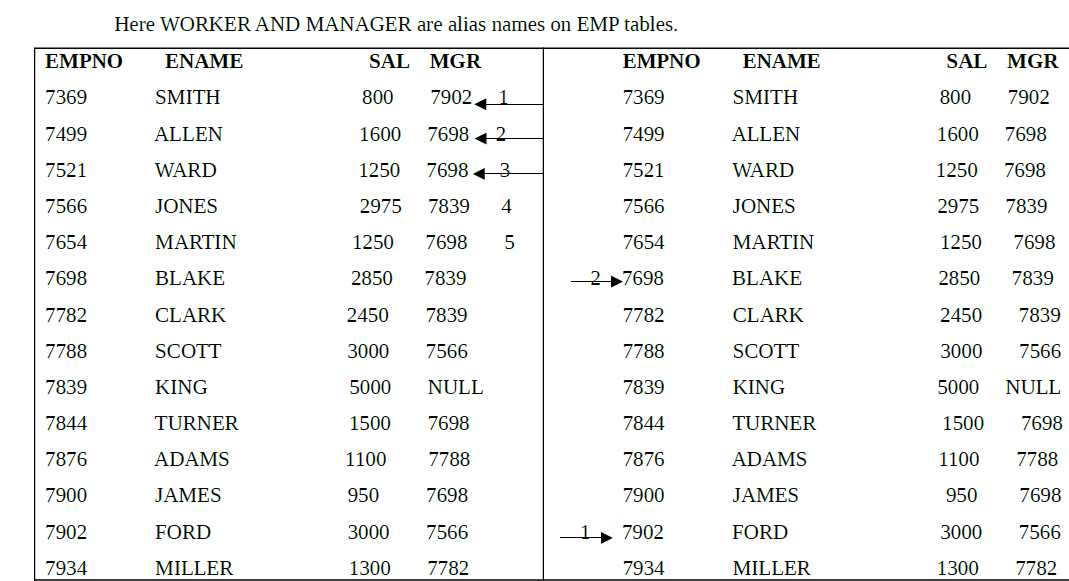
The above query displays the matched and unmatched records from **EMP** and **DEPT** tables .

**SELF JOIN**

To join the table itself

**To display name of the employee and his boss name from emp table .**

SQL> SELECT WORKER.ENAME, MANAGER.ENAME from EMP WORKER, EMP MANAGER where WORKER.MGR = MANAGER.EMPNO;



**EMPLOYEE BOSS**

SMITH FORD

ALLEN BLAKE

WARD BLAKE

JONES KING

MARTIN BLAKE

BLAKE KING

CLARK KING

SCOTT JONES

TURNER BLAKE

ADAMS SCOTT

JAMES BLAKE

FORD JONES

MILLER CLARK

13 Rows Selected.

SQL> SELECT WORKER.EMPNO ENO , WORKER.ENAME EMPLOYEE, MANAGER.ENAME BOSS,

MANAGER.EMPNO BNO FROM EMP WORKER, EMP MANAGER WHERE WORKER.MGR=MANAGER.EMPNO ;

**NON - EQUI JOIN**

If the rows from two or more tables are joined using operators( < , <= , > , >= ), then it is called as NONEQUIJOIN

SQL> SELECT EMPNO, ENAME, JOB, SAL, GRADE FROM EMP, SALGRADE WHERE SAL >=LOWSAL AND SAL <= HIGHSAL ;

**( 0R )**

SQL> SELECT EMPNO, ENAME, JOB, SAL, GRADE FROM EMP, SALGRADE WHERE SAL BETWEEN LOWSAL AND HIGHSAL ;

**SEQUENCES**

Sequence is a database object, which is used generate Unique Id automatically.

**Syntax:**

CREATE SEQUENCE <sequence\_name>

INCREMENT BY N

START WITH N

MINVALUE N

MAXVALUE N

CYCLE / NOCYCLE

CACHE / NOCACHE ;

START WITH N : Defined the starting value of sequence ( Default value is 1 ).

INCREMENT BY N : Defines the increment value of Id for the next record

MINVALUE N : Defines the minmum value of the sequence ( default value is 0 )

MAXVALUE N : Defines the maxmum value generated by the sequence

CYCLE : Specifies that, the Id should be generated from the starting value after reaching Max Value.

NOCYCLE : Specifies that, the Id should be generated after reaching Max Value. ( it is default )

CACHE : It defines the number of integers to be keep in memory. Default cache value is 20.

Sequence is associated with two pseudo columns 1) curval 2) nextval

Curval 🡪 specifies the current value in the sequence

Nextval 🡪 specifies next value in the sequence . It is used to generate starting value of sequence.

SQL> CREATE SEQUENCE SEQ1 ;

SQL> SELECT SEQ1.NEXTVAL FROM DUAL ; 🡪 1

SQL> SELECT SEQ1.CURRVAL FROM DUAL ; 🡪 1

SQL> CREATE SEQUENCE SEQ\_DEPT INCREMENT BY 10

START WITH 10

MINVALUE 10 MAXVALUE 90

CACHE 8

**Sequence with Cycle option :**

SQL> CREATE SEQUENCE SEQ\_DEPT1 INCREMENT BY 10

START WITH 10

MINVALUE 10 MAXVALUE 90

CYCLE CACHE 8 ;

When CYCLE option is specified, then CACHE Value must be less than one cycle.

SQL> CREATE TABLE DEPT\_TEST AS SELECT \* FROM DEPT WHERE 4=5;

It will create the empty DEPT\_TEST table as the condition is false

SQL> SELECT \* FROM DEPT\_TEST;

no rows selected

SQL> INSERT INTO DEPT\_TEST VALUES ( SEQ\_DEPT.NEXTVAL, 'SALES', 'HYDERABAD') ;

SQL> SELECT \* FROM DEPT\_TEST;

DEPTNO DNAME LOC

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

10 SALES HYDERABAD

SQL> INSERT INTO DEPT\_TEST VALUES ( SEQ\_DEPT.NEXTVAL, 'OPERATIONS', 'DELHI') ;

SQL> SELECT \* FROM DEPT\_TEST;

DEPTNO DNAME LOC

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

10 SALES HYDERABAD

20 OPERATIONS DELHI

SQL> INSERT INTO DEPT\_TEST VALUES ( SEQ\_DEPT.NEXTVAL, 'DISTRIBUTION', 'MUMBAI') ;

SQL> SELECT \* FROM DEPT\_TEST;

DEPTNO DNAME LOC

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

10 SALES HYDERABAD

20 OPERATIONS DELHI

30 DISTRIBUTION MUMBAI

**Altering Sequence :**

ALTER SEQUENCE <sequence\_name>

INCREMENT BY N

START WITH N

MINVALUE N

MAXVALUE N

CYCLE / NOCYCLE

CACHE / NOCACHE ;

**Data dictionary for sequence**  : USER\_SEQUENCES

SQL> SELECT \* FROM USER\_SEQUENCES

**Dropping a Sequence**

SQL> DROP SEQUENCE SEQ\_DEPT ;

**Data Control Language ( DCL)**

These commands are used to grant the privileges or to take privileges back.

**Commands are**

1. GRANT 🡪 To grant the privileges to user
2. Revoke 🡪 to take privileges back.

Oracle Provides two types of privileges

1) System Level Privileges 2) Object Level Privileges

**System Level Privileges :** These privileges are to perform the action against database. These privileges will be granted to the user by the Database Administrator ( DBA ) .

These are

1. Create Session
2. Create Table
3. Create Materialized View

**Object Level Privileges :** These privileges are to perform the action against database objects. These privileges will be granted by the user to another user.

These are

SELECT, INSERT, UPDATE and DELETE

**Role :** It is a database object. It is a group of privileges.

**Syntax :** CREATE ROLE <role\_name>

**Login as DBA :**

SQL> CONN SYS as sysdba ;

Password : sys

SQL> CREATE ROLE ROLE\_XX ;

Role created.

Dropping role

SQL> DROP ROLE ROLE\_XX ;

**Note :** Connect, resource and DBA are default roles provided by the Oracle.

**CREATING A USER**

**Syntax :**

CREATE USER <user\_name> IDENTIFIED BY <password>

GRANT CONNECT, RESOURCE TO <user\_name> IDENTIFIED BY <password>

**( OR )**

GRANT CONNECT, RESOURCE TO <user\_name>

LOGIN as ADMIN :

SQL> CONN SYSTEM/MANAGER@VIS ;

SQL> SHOW USER ;

Eg:

SQL> CREATE USER MNRAO IDENTIFIED BY MNRAO123 ;

SQL> GRANT CONNECT, RESOURCE TO MNRAO ;

( OR )

SQL> GRANT CONNECT, RESOURCE TO MNRAO IDENTIFIED BY MNRAO123 ;

**Verifying Users :**

SQL> SELECT USERNAME FROM DBA\_USERS WHERE USERNAME = 'MNRAO' ;

**Changing Password of User**

Password can be changed by the users themselves or by DBA.

**Syntax :**

ALTER USER <user\_name> IDENTIFIED BY <password>

SQL> ALTER USER MNRAO IDENTIFIED BY MNRAOXX ;

Disabling user : Only it can be by the DBA.

**Login as DBA :**

SQL> CONN SYSTEM/MANAGER@VIS ;

SQL> REVOKE CONNECT, RESOURCE FROM MNRAO ;

SQL> CONN MNRAO/MNRAOXX@VIS

ERROR:

ORA-01045: user MNRAO lacks CREATE SESSION privilege; logon denied

Note : Minimum privilege required to connect to oracle database is “ **CREATE SESSION** “

**Enabling the user**

**Login as DBA :**

SQL> CONN SYSTEM/MANAGER@VIS ;

SQL> GRANT CONNECT, RESOURCE TO MNRAO ;

**Locking User**

**Syntax :**

ALTER USER <user\_name> ACCOUNt LOCK;

SQL> ALTER USER MNRAO ACCOUNT LOCK ;

SQL> CONN MNRAO/MNRAOXX@VIS ;

ERROR:

ORA-28000: the account is locked

Warning: You are no longer connected to ORACLE.

**Checking Account Status :**

SQL> SELECT USERNAME, ACCOUNT\_STATUS FROM DBA\_USERS WHERE USERNAME='MNRAO' ;

**USERNAME ACCOUNT\_STATUS**

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

MNRAO LOCKED

**Un Locking the user account :**

**Syntax :**

ALTER USER <user\_name> ACCOUNT UNLOCK

**Login as DBA :**

SQL> CONN SYSTEM/MANAGER@VIS ;

SQL> ALTER USER MNRAO ACCOUNT UNLOCK ;

SQL> SELECT USERNAME, ACCOUNT\_STATUS FROM DBA\_USERS WHERE USERNAME='MNRAO' ;

**USERNAME ACCOUNT\_STATUS**

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

MNRAO OPEN

**Dropping Users:**

**Syntax :**

SQL> DROP USER <user\_name>

SQL> DROP USER MNRAO ;

**Working with object level privileges**

Eg: Login users are

1) scott / tiger 2) mnrao / mnrao123 3) ram/ram123

Tables in scott user

1) emp 2) dept

SQL> CONN SYSTEM/MANAGER@VIS;

SQL> CREATE USER MNRAO IDENTIFIED BY MNRAO123 ;

SQL> GRANT CONNECT, RESOURCE TO MNRAO ;

SQL> CREATE USER RAM IDENTIFIED BY RAM123

SQL> GRANT CONNECT, RESOURCE TO RAM

Login as SCOTT/TIGER

SQL> CONN SCOTT/TIGER@VIS

SQL> GRANT SELECT, INSERT ON EMP TO MNRAO ;

SQL> GRANT UPDATE, DELETE ON scott.DEPT TO RAM ;

SQL> SELECT \* FROM USER\_TAB\_PRIVS WHERE TABLE\_NAME IN ( 'EMP', 'DEPT' ) ;

**GRANTEE OWNER TABLE\_NAME GRANTOR PRIVILEGE**

MNRAO SCOTT EMP SCOTT SELECT

MNRAO SCOTT EMP SCOTT INSERT

RAM SCOTT DEPT SCOTT UPDATE

RAM SCOTT DEPT SCOTT DELETE \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**LOGIN AS MNRAO :**

SQL> CONN MNRAO/MNRAO123@VIS

SQL> SELECT \* FROM SCOTT.EMP ;

SQL > INSERT INTO SCOTT.EMP ( EMPNO, ENAME, SAL ) VALUES ( 3001, ‘ABC’, 5550 );

LOGIN AS RAM

SQL> CONN RAM/RAM123@VIS

SQL> SELECT \* FROM SCOTT.DEPT ;

Error since no privileges

SQL> UPDATE SCOTT.DEPT SET LOC=’HYD’ WHERE DEPTNO = 20 ;

SQL> DELETE FROM SCOTT.DEPT WHERE DEPTNO = 40 ;

**Taking privileges back**

Login as scott

SQL> CONN SCOTT/TIGER@VIS

SQL> REVOKE SELECT, INSERT ON EMP FROM MNRAO ;

SQL> REVOKE UPDATE, DELETE ON EMP FROM RAM ;

**Granting Privileges with grant option :**

SQL> GRANT SELECT, UPDATE ON EMP TO MNRAO WITH GRANT OPTION ;

Now the user MNRAO can grant the SELECT, UPDATE ON SCOTT.EMP to other users

Now login as MNRAO

SQL> CONN MNRAO/MNRAO123@VIS

SQL> GRANT SELECT, UPDATE ON SCOTT.EMP TO RAM ;

Now login as SCOTT

SQL> REVOKE SELECT, UPDATE ON EMP FROM MNRAO ;

Both user MNRAO and RAM will loose the privileges SELECT, UPDATE on SCOTT.EMP table.

**Granting privileges on selected columns of the table**

Login as scott

SQL> CONN SCOTT/TIGER@VIS

SQL> GRANT INSERT ( EMPNO, ENAME ) ON EMP TO MNRAO ;

SQL> GRANT UPDATE ( SAL ) ON EMP TO MNRAO;

**Taking the privileges back**

SQL> REVOKE INSERT, UPDATE ON EMP FROM MNRAO ;

**VIEWS**

🡪 View is a logical table based on a table or another view

🡪 It is like a window through which the data in a table can be viewed.

🡪 View is a store in data dictionary in the form of select statement.

🡪 View will not have any data. It is to only to work on the table.

**Types of views**

1) Simple View 2) Complex View 3) Materialized View

**Simple View :**

**🡪**It is a view based on single table.

🡪 It will not work with any arithmetic expressions

🡪 All DML operations are allowed on a **simple view**

DML Operations performed on the view will effect the table.

**Syntax :**

CREATE VIEW <view\_name> as

SELECT STATEMENT [ WITH CHECK OPTIONS ] [WITH READ ONLY ]

SQL> CREATE VIEW EMP\_VIEW AS

SELECT \* FROM EMP ;

SQL> SELECT \* FROM EMP\_VIEW ;

( OR )

SQL> SELECT \* FROM EMP ;

SQL> INSERT INTO EMP\_VIEW (EMPNO, ENAME, SAL ) VALUES ( 1003, 'RAMESH', 6700) ;

SQL> SELECT \* FROM EMP WHERE EMPNO=1003 ;

SQL> CREATE OR REPLACE VIEW DEPT\_10\_VIEW AS SELECT \* FROM EMP WHERE DEPTNO=10 ;

SQL> SELECT \* FROM DEPT\_10\_VIEW ;

SQL> SELECT COUNT(\*) FROM EMP ;

14

SQL> SELECT COUNT(\*) FROM DEPT\_10\_VIEW

3

SQL> INSERT INTO DEPT\_10\_VIEW ( EMPNO, ENAME, DEPTNO) VALUES (1005, 'SANGEETHA', 10) ;

SQL> INSERT INTO DEPT\_10\_VIEW (EMPNO, ENAME, DEPTNO) VALUES (1006, 'GEETHA', 20) ;

SQL> INSERT INTO DEPT\_10\_VIEW ( EMPNO, ENAME, DEPTNO) VALUES (1007, 'RAVI', 30) ;

SQL> SELECT COUNT(\*) FROM EMP ;

17 ( 14 + 3 )

SQL> SELECT COUNT(\*) FROM DEPT\_10\_VIEW

1. ( 3 + 1 )

SQL> UPDATE DEPT\_10\_VIEW SET SAL=SAL+255

4 Rows updated

Only 10th dept will be updated.

SQL> DELETE FROM DEPT\_10\_VIEW ;

4 Rows deleted

Only 10th dept will be deleted.

**Creating with check options**

SQL> CREATE VIEW DEPT\_20\_VIEW AS

SELECT \* FROM EMP WHERE DEPTNO =20

WITH CHECK OPTION ;

SQL> SELECT \* FROM DEPT\_20\_VIEW ;

Only 20th dept will be displayed

SQL> INSERT INTO DEPT\_20\_VIEW ( EMPNO, ENAME, DEPTNO ) VALUES ( 1010,'RAMU', 20) ;

1 Row created

SQL> INSERT INTO DEPT\_20\_VIEW ( EMPNO, ENAME, DEPTNO ) VALUES ( 1011,'SUNIL', 30) ;

ERROR at line 1:

ORA-01402: view WITH CHECK OPTION where-clause violation

SQL> UPDATE DEPT\_20\_VIEW SET SAL=SAL+233 ;

ONLY 20th dept will be updated.

SQL> UPDATE DEPT\_20\_VIEW SET SAL=SAL+233 WHERE DEPTNO=30 ;

0 Rows updated as the view works with only 20th dept

**Creating View with read only option**

🡪 Read Only view allows only SELECT statement.

🡪 No DML operations can be performed through the read only view.

SQL> CREATE OR REPLACE VIEW EMP\_READ\_VIEW AS

SELECT \* FROM EMP

WITH READ ONLY ;

SQL> SELECT \* FROM EMP\_READ\_VIEW ;

SQL> DELETE FROM EMP\_READ\_VIEW ;

ERROR at line 1:

ORA-01752: cannot delete from view without exactly one key-preserved table

**FORCE VIEW:**

It is a view with out table

SQL> CREATE TABLE EMP\_TEST AS SELECT \* FROM EMP ;

SQL> SELECT \* FROM EMP\_TEST ;

14 ROWS SELECTED

SQL> CREATE OR REPLACE VIEW EMP\_TEST\_VIEW AS SELECT \* FROM EMP\_TEST ;

View created.

SQL> SELECT \* FROM EMP\_TEST\_VIEW ;

14 ROWS SELECTED

SQL> DROP TABLE EMP\_TEST ;

Table dropped.

SQL> DROP VIEW EMP\_TEST\_VIEW

View dropped.

In the above commands the table EMP\_ TEST and the view EMP\_TEST\_VIEW have been dropped.

Now try to create view on table EMP\_TEST

SQL> CREATE OR REPLACE VIEW EMP\_TEST\_VIEW AS SELECT \* FROM EMP\_TEST ;

**( \* )**

ERROR at line 1:

ORA-00942: table or view does not exist

**Create the view forcibly**

SQL> CREATE OR REPLACE FORCE VIEW EMP\_TEST\_VIEW AS SELECT \* FROM EMP\_TEST;

**Warning:** View created with compilation errors.

SQL> SELECT \* FROM EMP\_TEST\_VIEW

Error since no table.

SQL> CREATE TABLE EMP\_TEST AS SELECT \* FROM EMP ;

SQL> SELECT \* FROM EMP\_TEST\_VIEW ;

14 Rows selected

A table can be dropped even views are created on the table.

SQL> DROP TABLE EMP\_TEST ;

Table dropped

SQL> SELECT \* FROM EMP\_TEST\_VIEW ;

Error since table dropped.

To enable the above view , create table with same as the view

**Creating view from another view**

SQL> CREATE OR REPLACE VIEW V1 AS SELECT \* FROM EMP ;

SQL> CREATE OR REPLACE VIEW V2 AS SELECT \* FROM V1 ;

View created.

SQL> CREATE OR REPLACE VIEW **V\_XX** AS SELECT EMPNO, ENAME, DEPTNO FROM EMP

View created.

SQL> CREATE OR REPLACE VIEW **V\_YY** AS SELECT EMPNO, ENAME, SAL FROM **V\_XX** ;

Error since SAL column is not part of **V\_XX**

**Complex views**

🡪 It is a view created with group functions

🡪 It is a view created on multiple tables.

🡪 if a view created with group functions, the it is a read only view. Hence we can not perform DML

Operations on the table through the table.

**Creating view with group functions :**

SQL> CREATE OR REPLACE VIEW GRP\_VIEW AS SELECT DEPTNO, SUM(SAL) SAL\_SUM, AVG(SAL) AVG\_SAL, COUNT(\*) REC\_COUNT FROM EMP GROUP BY DEPTNO ;

SQL> SELECT \* FROM GRP\_VIEW ;

**Creating view based on two tables :**

SQL> CREATE OR REPLACE VIEW EMP\_DEPT\_VIEW AS SELECT EMPNO, ENAME, DEPT.DEPTNO, DNAME, LOC FROM EMP, DEPT WHERE EMP.DEPTNO=DEPT.DEPTNO ;

SQL> SELECT \* FROM EMP\_DEPT\_VIEW ;

EMPNO ENAME DEPTNO DNAME LOC

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

7369 SMITH 20 RESEARCH DALLAS

7499 ALLEN 30 SALES CHICAGO

7521 WARD 30 SALES CHICAGO

7566 JONES 20 RESEARCH DALLAS

7654 MARTIN 30 SALES CHICAGO

7698 BLAKE 30 SALES CHICAGO

7782 CLARK 10 ACCOUNTING NEW YORK

7788 SCOTT 20 RESEARCH DALLAS

7839 KING 10 ACCOUNTING NEW YORK

7844 TURNER 30 SALES CHICAGO

7876 ADAMS 20 RESEARCH DALLAS

7900 JAMES 30 SALES CHICAGO

7902 FORD 20 RESEARCH DALLAS

7934 MILLER 10 ACCOUNTING NEW YORK

SQL> INSERT INTO EMP\_DEPT\_VIEW (EMPNO, ENAME ) VALUES (2001, 'XYZ') ;

Record will be inserted into the child table emp through the view

SQL> INSERT INTO EMP\_DEPT\_VIEW (DEPTNO, DNAME, LOC ) VALUES (70, 'MARKETING', 'DELHI') ;

Error since Record can not be inserted into the master table ( i.e dept )

SQL> INSERT INTO EMP\_DEPT\_VIEW VALUES (2002,'ABC',80, 'DISTRIBUTION', 'MUMBAI') ;

cannot modify more than one base table through a join view

**MATERIALIZED VIEW**

🡪 Materialized view contains both definition and data.

🡪 as the Materialized view contains data it demands space.

🡪 as the simple and complex views does not contain any data, they will not demand any space.

🡪 DML Operations can not be performed on the Materialized View. It is a read only view.

🡪 to create materialized view, user must have the “ CREATE MATERIALIZED VIEW “ privilege

🡪 Materialized view will be refreshed automatically based on the data in a table.

A Materialized view can be refreshed in to ways

1) Complete 2) Fast

**Row Id :**  It is Unique Id number allocated by the oracle, when a record inserted into the table. This Id will also be stored into the table along with table. If record is deleted, then corresponding Row Id will also be deleted.

SQL> SELECT ROWID, EMPNO, ENAME FROM EMP ;

**ROWID** **EMPNO** **ENAME**

AAAzG2AAdAAAdOIAAA 7369 SMITH

AAAzG2AAdAAAdOIAAB 7499 ALLEN

AAAzG2AAdAAAdOIAAC 7521 WARD

AAAzG2AAdAAAdOIAAD 7566 JONES

AAAzG2AAdAAAdOIAAE 7654 MARTIN

AAAzG2AAdAAAdOIAAF 7698 BLAKE

AAAzG2AAdAAAdOIAAG 7782 CLARK

AAAzG2AAdAAAdOIAAH 7788 SCOTT

AAAzG2AAdAAAdOIAAI 7839 KING

AAAzG2AAdAAAdOIAAJ 7844 TURNER

AAAzG2AAdAAAdOIAAK 7876 ADAMS

AAAzG2AAdAAAdOIAAL 7900 JAMES

AAAzG2AAdAAAdOIAAM 7902 FORD

AAAzG2AAdAAAdOIAAN 7934 MILLER

**Creating Materialized view with complete refreshment option**

**Step1 : login as sys admin**

SQL> CONN SYS as sysdba ;

Password : sys

**Step2: grant the privilege to scott user to create materialized view**

SQL> GRANT CREATE MATERIALIZED VIEW TO SCOTT ;

SQL> CREATE MATERIALIZED VIEW DEPT\_MV REFRESH COMPLETE WITH ROWID START WITH SYSDATE NEXT SYSDATE+1/(24\*60\*60) AS SELECT \* FROM DEPT ;

Snap shot created.

SQL> SELECT COUNT(\*) FROM DEPT ;

4

SQL> SELECT COUNT (\*) FROM DEPT\_MV ;

4

SQL> INSERT INTO DEPT ( DEPTNO) VALUES (50) ;

SQL> SELECT COUNT(\*) FROM DEPT ;

5

SQL> SQL> SELECT COUNT(\*) FROM DEPT\_MV ;

4

Since materialized view has not been refreshed

SQL> COMMIT ;

SQL> SQL> SELECT COUNT(\*) FROM DEPT\_MV ;

5

After commit data physically committed to the table. Hence materialized view will also be refreshed.

**SYNONYMS**

🡪 It is a pointer to table or view

🡪 It can not be created for selected columns or rows of a table.

🡪 DML operations performed on the synonym will effect the table.

**Types of synonyms**

1) Private synonym 2) Public synonym

Private synonym created by the user

**Syntax :**

CREATE SYNONYM <synonym\_name> for table / view

Login as DBA

SQL> CONN SYSTEM/MANAGER@VIS ;

SQL> SHOW USER

SYSTEM

SQL> GRANT CREATE SYNONYM TO SCOTT ;

Login as scott

SQL> CONN SCOTT/TIGER@VIS;

SQL> CREATE SYNONYM PRI\_SYN FOR DEPT ;

SQL> SELECT \* FROM PRI\_SYN; 🡪 shows the details of DEPT table

DEPTNO DNAME LOC

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

10 ACCOUNTING NEW YORK

20 RESEARCH DALLAS

30 SALES CHICAGO

40 OPERATIONS BOSTON

SQL> DESC PRI\_SYN ; 🡪 Shows the description of DEPT table.

SQL> DELETE FROM PRI\_SYN WHERE DEPTNO = 30 ; 🡪 record from the DEPT table will be deleted.

**Public Synonym**

Synonym created by user with “ CREATE PUBLIC SYNONYM “ privilege

It can be used by all the users

SYSTEM> GRANT CREATE PUBLIC SYNONYM TO SCOTT ;

SCOTT> CREATE PUBLIC SYNONYM PUB\_SYN FOR DEPT ;

SCOTT> SELECT \* FROM PUB\_SYN ; 🡪 It shows all the records of DEPT table

MNRAO> SELECT \* FROM PUB\_SYN ; 🡪 It shows all the records of DEPT table

MNRAO> SELECT \* FROM PRI\_SYN ;

Error since synonym is a private

SCOTT> GRANT SELECT ON PRI\_SYN TO MNRAO ;

MNRAO> SELECT \* FROM SCOTT.PRI\_SYN ; 🡪 It shows all the records of DEPT table

**Data Dictionary for Synonyms :** USER\_SYNONYMS

SQL> SELECT \* FROM USER\_SYNONYMS ;

**Dropping a synonym**

SQL> DROP SYNONYM <synonym\_name >

SCOTT> DROP SYNONYM PRI\_SYN ;

**PUBLIC synonym can be dropped by only system administrator**

**Syntax :**

DROP PUBLIC SYNONYM <Synonym\_name>

SYSTEM > DROP PUBLIC SYNONYM PUB\_SYN ;

**INDEXES**

🡪 It is a database object used to improve the performance of SELECT statement.

🡪 Index requires some extra space in addition to space occupied by actual contents of table.

🡪 Indexes will be created on columns, which are being used frequently with WHERE clause

🡪 Index will improve the performance of SELECT statement but decrease the performance of DML statements.

**Types of Indexes**

1) Unique Index 2) Composite Index

3) Descending Index 4) Reverse Key Index

5) Functional based Index

**1) Unique Index :**

🡪 It is an Index on single column with UNIQUE CONSTRAINT .

🡪 It will not allow duplicate values.

SQL> CREATE UNIQUE INDEX UQ\_IND ON DEPT ( DNAME );

**2) Composite Index :** 🡪 It is on more than one column of the table.

SQL> CREATE INDEX COMP\_IND ON EMP ( EMPNO, ENAME ) ;

**3) Descending Index :** 🡪 It is an Index created on columns to display in sorted order.

SQL> CREATE INDEX DESC\_IND ON EMP (ENAME ASC , SAL DESC ) ;

**4) Reverse Key Index :** 🡪 It is an Index created on columns containing data in sequential order.

SQL> CREATE INDEX REV\_IND ON EMP ( EMPNO ) REVERSE ;

**5) Functional based Index :** 🡪 It is created on Arithmetic Expressions on the columns of the table.

SQL> CREATE INDEX FUN\_IND ON EMP ( SAL+COMM ) ;

**Data Dictionary for Indexes**

1) USER\_INDEXES 2) USER\_IND\_COLUMNS

**Dropping Indexes :**

**Syntax :**

DROP INDEX <index\_name>

SQL> DROP INDEX UQ\_IND ;

SQL> DROP INDEX DESC\_IND ;

**CLUSTERS**

Cluster is a database object, which is used to improve the performance of select statement based on MASTER and DETAILED tables.

It hold the common column of the MASTER and DETAILED tables.

SQL> CREATE CLUSTER CL1 ( DEPTNO NUMBER(9) ) ;

SQL> CREATE INDEX CL\_IND ON CLUSTER CL1 ;

SQL> CREATE TABLE DEPT\_TEST ( DEPTNO NUMBER (9) PRIMARY KEY,

DNAME VARCHAR2(20), LOC VARCHAR2(40) ) CLUSTER CL1 ( DEPTNO ) ;

SQL> CREATE TABLE EMP\_TEST ( EMPNO NUMBER(10),

ENAME VARCHAR2(10),

DEPTNO NUMBER(9) ) CLUSTER CL1 ( DEPTNO ) ;

SQL> ALTER TABLE EMP\_TEST ADD FOREIGN KEY (DEPTNO) REFERENCES DEPT\_TEST(DEPTNO);

**Data Dictionary for view for Cluster :** USER\_CLUSTERS

SQL> SELECT TABLE\_NAME , CLUSTER\_NAME FROM USER\_TABLES

WHERE TABLE\_NAME IN ( ‘EMP\_TEST’, ‘DEPT\_TEST ) ;

**Dropping a Cluster :**

To drop a cluster it should be empty

**Syntax :**

DROP CLUSTER <cluster\_name>

**Dropping a Cluster including Tables:**

**Syntax :**

DROP CLUSTER <cluster\_name> INCLUDING TABLES

SQL> DROP CLUSTER CL1 INCLUDING TABLES ;

**PL/SQL**

**Introduction to PL/SQL**

* PL/SQL stands for Procedural Language for SQL
* PL/SQL introduced in oracle from oracle version 6.
* PL/SQL combines the data processing power of procedural language and Data manipulation power of SQL
* Multiple statements can be executed at a time.
* If supports variables to hold data
* Supports conditional statements to execute statements conditionally
* Set of statements can be executed required number of times using loops
* If supports modularity and reusability of programs through subprograms

**Versions of PL/SQL**

SQL> select \* from v$version;

Display the version of oracle and PL/SQL

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Oracle Version | 6.0 | 7.0 | 7.1 | 7.2 | 7.3 | 8.0 | 8.1 | 9.1 | 10.1 | 11.1 | 11.2 |
| PL/SQL version | 1.0 | 2.0 | 2.1 | 2.2 | 2.3 | 8.0 | 8.1 | 9.1 | 10.1 | 11.1 | 11.2 |

PL/SQL program is called PL/SQL block

**PL/SQL block structure**

DECLARE

<VARIABLE DECLARATION>

BEGIN

<EXEUTABLE STATEMENTS>

EXCEPTION

<EXCEPTION PORT>

END;

VALID VARIABLE DECLARATIONS

vno number(5);

vname varchar(20);

**Initialization:**

vcom number(5):= 5000;

vdno number (5) not null :=10;

vjob varchar(15) *default* ‘analyst ’;

VALIED EXECUTABLE STATEMENTS

* All DDL ,DML statements
* Select <column-list> into <variable-list> from <table-name> [where <condition>];

**COMMENT OPERATORS**

- - Single line comment operator

/\*

Multiline comment operator

\*/

**DBMS –OUTPUT .PUT\_LINE**

It is used to display message or value in variable

Syntax:

dbms\_output.put\_line (‘message’);

or

dbms\_output.put\_line(variable);

or

dbms\_output.put\_line(‘message’ || variable);

Note: default dbms\_ouput.put\_line is in disable mode.

*ACTIVATING dbms\_output.put\_line*

SQL>set serveroutput on;

This activation valid only for one session.

? WAP to display message

begin

*dbms\_output.put\_line (‘welcome to PL/SQL’);*

end;

/

O/p -🡪 welcome to PL/SQL

**WAP to display values in variables**

declare

A number (5):=10;

B number (5):= 20;

Begin

dbms\_output.put\_line ('value in a:' || a);

dbms\_output.put\_line ('value in b:' || b);

end;

/

set serveroutput on;

**WAP to real values into variables & display the values**

Declare

A number (5):=&a;

B number (5):=&b;

Begin

dbms\_output.put\_line ('value in a:' || a);

dbms\_output.put\_line ('value in b:' || b);

end;

/

SQL> set verify off;

SQL>/

O/p 🡪 verify the differences

**WAP to Swap values of variables**

declare

a number (5):=10;

b number (5):=20;

c number (5);

Begin

dbms\_output.put\_line ('before swapping' );

dbms\_output.put\_line ('value in a:' || a);

dbms\_output.put\_line ('value in b:' || b);

c:=a;

a:=b;

b:=c;

dbms\_output.put\_line ('After swapping' );

dbms\_output.put\_line ('value in a:' || a);

dbms\_output.put\_line ('value in b:' || b);

end;

/

***WAP to display ename, job for given employee***

declare

Vno number (5):=&vno;

Vname varchar (15);

Vjob varchar (15);

Begin

Select ename, job into vname, vjob from emp where empno=vno;

dbms\_output.put\_line (vno ||' ' || vname ||' '||vjob);

end;

/

***WAP to display sal, hiredate for given employee***

*declare*

*Vno number (5):=&vno;*

*Vsal number (5);*

*Vdoj date;*

*Begin*

*Select sal, hiredate into vsal, vdoj from emp where empno=vno;*

*dbms\_output.put\_line (vno ||' ' || vsal ||' ' ||vdoj);*

*end;*

*/*

***WAP to display* ename, job, sal from given employee**

*declare*

*Vno number (5) :=&vno;*

*Vname varchar (10);*

*Vjob varchar (10);*

*Vsal number (5);*

*Begin*

*Select ename, job, sal into vname, vjob, vsal from emp where empno=vno;*

*dbms\_output.put\_line (vno ||' ' || vname ||' ' ||vjob ||' ' || vsal);*

*end;*

*/*

***WAP to display dname, loc for given dept form dept table***

*declare*

*Vdno number (5) := &vdno;*

*Vdname varchar (10);*

*Vloc varchar (10);*

*Begin*

*Select dname, loc into vdname,vloc from dept where deptno = vdno;*

*dbms\_output.put\_line (vdno || ' ' || vdname || ' ' || vloc );*

*end;*

*/*

*Declare*

*Vno number (5):=7499;*

*Vjob varchar (2);*

*Begin*

*Select job into vjob from emp where empno = vno;*

*Dbms\_output.put\_line(vno||' '||vjob);*

*End;*

*/*

*Error since vjob size is not sufficient to hold job of emp*( Buffer size too small )

***Attributes:***

1. % type ( column type )
2. % rowtype ( row type )
3. % type 🡪used to declare variable to refer particular column of table

**Syntax:**

Variable-name table.column %type

EX:

1. vno emp.empno % type

Note : Datatype & size of vno is same as datatype of empno of emp table

2) Vjob emp.job%type;

3) Vdoj emp.hiredate%type;

2) %rowtype---- used to declare variable to refer entire row of table

Syntax:

Variable\_name table\_name%rowtype;

EX: 1) d dept %rowtype;

2) e emp%rowtype;

**WAP to display given dept data from dept table**

declare

d dept%rowtype;

Begin

Select \* into d from dept where deptno=10;

Dbms\_output.put\_line(d.deptno||' '||d.dname||' '||d.loc);

End;

/

**WAP to display given employee data from emp table**

declare

e emp%rowtype;

Begin

Select \* into e from emp where empno = &eno;

Dbms\_output.put\_line(e.empno||' '||e.ename||' '||e.job||' '||e.sal||' '||e.deptno);

End;

/

**WAP to display ename, hiredate for given employee**

Declare

Vno emp.empno%type:=&vno;

Vname emp.ename%type;

Vdoj emp.hiredate%type;

Begin

Select ename,hiredate into vname, vdoj from emp where empno = vno;

Dbms\_output.put\_line(vno||’ ‘||vname||’ ‘||vdoj);

End;

/

**CONDITIONAL STATEMENTS:**

Statements in programs can be executed conditionally

1. IF:

Syntax: if (condition)

then

\_\_\_\_

\_\_\_\_

\_\_\_\_

end if;

1. IF \_ THEN\_ELSE:

Syntax: if (condition)

then

\_\_\_\_

\_\_\_\_

\_\_\_\_

else

\_\_\_\_

\_\_\_\_

\_\_\_\_

end if;

Given condition true, then statements under if are executed otherwise statements under else are executed

1. LADDER IF:

Syntax: if (condition 1)

then

\_\_\_\_

\_\_\_\_

\_\_\_\_

else if (condition 2)

then

\_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

else if (condition 3)

then

\_\_\_\_

\_\_\_\_

\_\_\_\_

else

\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

End if;

End if;

End if;

All the conditions false, then statements under else are executed

**WAP to compare two numbers**

declare

a number(5):=&a;

b Number(5):=&b;

begin

If (a=b)

then

Dbms\_output.put\_line ('both are equal');

Else if (a > b) then

Dbms\_output.put\_line (a||' is big');

Else

Dbms\_output.put\_line (b||' is big');

End if;

End if;

End;

/

**WAP to display given employee data if sal > 1000;**

declare

Vno emp.empno %type :=&vno;

Vname emp.ename % type;

Vsal emp.sal %type;

Begin

Select ename, sal into vname, vsal from emp where empno=vno;

If (vsal >1000) then

Dbms\_output.put\_line(vno || ' ' || vname || ' ' || vsal);

End if;

End;

/

**WAP to check given employee commission null or not**

declare

Vno emp.empno % type := &vno;

Vcomm emp.comm %type;

Begin

Select comm into vcomm from emp where empno=Vno;

If(vcomm is null) then

Dbms\_output.put\_line(vno || ' comm is null ' );

else

Dbms\_output.put\_line(vno || ' ' ||vcomm);

End if;

End;

**WAP to check given number is even or odd**

declare

N number (5):=&n;

Begin

If( mod(n,2)=0 ) then

Dbms\_output.put\_line('given number is even');

Else

Dbms\_output.put\_line('given number is odd');

End if ;

End;

**WAP to find net salary of given employee**

declare

Vno emp.empno % type := &vno;

Vsal emp.sal %type;

Vcomm emp.comm %type;

Net emp.sal%type;

Begin

Select sal, comm into vsal ,vcomm from emp where empno = vno;

If(vcomm is null) then

Net:=vsal;

Dbms\_output.put\_line('net salary :'|| net);

Else

Net:=vsal+vcomm;

Dbms\_output.put\_line('net salary :'|| net);

End if;

End;

**LOOPS:-**

Loops are used to execute iterative statements

1)Simple loop

2)while loop

3)Numeric for loop

1)Simple loop

Syntax :

loop

----------

----------

-----------

End loop;

Above loop is infinite loop

**Adding breaking condition to simple loop**

Syntax:-

loop

----------

----------

-----------

Exit when (condition);

End loop;

* Control comes out from loop given condition is true

**WAP to print 1 to 10 numbers using simple loop**

declare

I number(5) := 1;

Begin

Loop

Dbms\_output.put\_line ( i);

I:=i+1;

Exit when I > 10 ;

End loop;

End;

/

While loop

Syntax:

While (condition)

loop

-----

------

-----

End loop;

Given condition is true, then statements under while are executed.

**WAP to print numbers up to given number using while loop**

declare

N number(5) := &n;

i number(5) :=1;

begin

while (I <= n ) loop

Dbms\_output.put\_line ( i);

I:=i+1;

End loop;

End;

/

**WAp to find even numbers sum and odd number sum upto given number using while**

SQL> declare

I number (5) :=1;

n number (5):=&NUM;

Es number (5):=0;

Os number(5):=0;

Begin

While(i <= n )

loop

If (mod (I ,2)= 0)

then

Es :=es+i;

Else

Os :=os+i;

End if;

i := i+1 ;

End loop;

Dbms\_output.put\_line ('even sum:' ||es );

Dbms\_output.put\_line ('odd sum: '|| os);

End;

**Numeric for loop**

Syntax: -

For <variable > in lowbond : highbond loop

--------

---------

-----------

End loop;

Example:-

For I in 1 .. 10 loop

---------

----------

----------

End loop;

* Variable I is implicitly declared variable called indexed variable
* Declaration of I is not required in declare part of PL/SQL block
* Arithmetic operations are not allowed on i.
* “ . . “ is range operator.

Eg:

declare

begin

For I in 10 .. 20 loop

Dbms\_output.put\_line (i);

End loop;

end;

**CURSORS**

**Active set: - set of records returned by the select statement**

Context area :- area of memory allocated by oracle to process SQl statement context are holds active set

**Cursor** :

It is handle or pointer to context area.

PL/SQL program control context area using cursor

**Types of cursors**

1. Static cursor
2. Dynamic or ref cursor

1)Static cursor

Select statement associated with cursor is constant through the program

Type of Static cursor

1.explicit cursor

2.implicit cursor

1. explicit cursor : cursor created by user

Steps to process explicit cursor

1 declare cursor

2 open cursor for query

3 Fetch data from cursor

4 close cursor

**Declare cursor :-**

Cursor declaration detains the name of cursor and select statement associated with it.

🡪Cursor should be declared part of in declare part of PL/SQL block

Syntax:-

Cursor <cursor name > for select statement

Eg:

Cursor cl is select \* from dept;

**Opening cursor:**

cursor should be opened in executable form of PL/SQL block

* If cursor is opened, data related to select statement transferred from database to memory i.e ;

Active set is determine active set pointer or cursor pointing to first record of active set (fetching data).

Syntax :-

Open <cursor name>;

Ex:

Open cl;

**Fetching data from cursor :**

fetch operation is used to picks record from active set and immediately dump into PL/SQL variables. After each fetch active set point or cursor increase to next record

Syntax:-

Fetch < cursor-name> into <PL/SQL variables>

**Closing cursor:-**

After fetching all records cursor can be closed. Data cannot be fetched from closed cursor

Syntax:

Close <cursor - name>;

Close cl;

Cursor attributes:-

1) cursor\_name % found 🡪returns true if cursor contains data for fetching

2) cursor\_name % not found 🡪 returns true if cursor will not any data.

3) Cursor\_name % isopen 🡪 returns true if cursor is opened

4) cursor\_name%rowcount🡪 returns number of records fetched by cursor

**WAP to display data from Dept table using cursor**

declare

Cursor cl is select \* from dept;

Vdno dept.deptno % type;

Vdname dept.dname%type;

Vloc dept.loc%type;

Begin

Open cl;

Loop

Fetch cl into vdno , vdname, vloc ;

Exit when cl % notfound;

Dbms\_output.put\_line(vdno ||' ' || vdname || ' ' || vloc);

End loop;

Close cl;

End;

/

**WAP to display data from emp table using cursor**

declare

Cursor cl is select \* from emp;

E emp%rowtype;

Begin

Open cl;

Loop

Fetch cl into e;

Exit when cl % notfound;

Dbms\_output.put\_line(e.empno || ' ' ||e.ename || ' ' ||e.job || ' '||e.deptno);

End loop;

Close cl;

End;

/

**WAP to display all managers data from emp table**

declare

Cursor cl is select \* from emp where job = 'manager';

E emp%rowtype;

Begin

Open cl;

Loop

Fetch cl into e;

Exit when cl % notfound;

Dbms\_output.put\_line(e.empno || ' ' ||e.ename || ' ' ||e.job || ' ' ||e.deptno);

End loop ;

Close cl;

End;

/

**WAp to find count & experience for all employee in emp table**

declare

Cursor cl is select \* from emp;

E emp%rowtype;

Net number(5);

Exp number(5);

Begin

Open cl;

Dbms\_output.put\_line('No of records fetched :' || cl%rowcount);

Loop

Fetch cl into e;

Exit when cl % notfound;

Net :=e.sal + nvl(e.comm,0);

Exp:= round(months\_between(sysdate, e.hiredate)/12);

Dbms\_output.put\_line(e.empno || ' ' || e.sal || ' ' || e.comm || ' '||e.hiredate || ' ' ||net || ' ' || exp);

End loop;

Dbms\_output.put\_line('No of records fetched : '|| cl%rowcount);

Close cl;

End;

**Modify the salaries of employees based on job**

declare

Cursor cl is select \* from emp;

E emp%rowtype;

Begin

Open cl;

Loop

Fetch cl into e;

Exit when cl % notfound;

If(e.job='MANAGER') then

update emp set sal = sal +5000 where empno=e.empno;

Dbms\_output.put\_line(e.job || ' salary modified');

Else if (e.job='ANALYST') then

Update emp set sal=sal+2500 where empno=e.empno;

Dbms\_output.put\_line(e.job ||' salary modified');

Else if(e.job='PRESIDENT') THEN

Update emp set sal=sal+1000 where empno=e.empno;

Dbms\_output.put\_line(e.job ||' salary modified');

ELSE

Update emp set sal=sal+1000 where empno=e.empno;

Dbms\_output.put\_line(E.EMPNO || ' ' || e.ename || ' ' ||e.job || ' ' ||e.deptno);

End if ;

end if;

end if;

End loop;

Close cl;

End;

**Cursors with parameters:**

**Syntax :**

Cursor <cursor\_name>(para1 data\_type, para2 data\_type ,…) Is select statement;

**Eg1:**

Cursor cl (dno number) Is select \* from emp where deptno=dno;

**Eg2:**

Cursor cl (vjob varchar, vdno number) Is select \* from emp where job = vob and deptno=vdno;

**Syntax to open cursor**

**Eg1:**

Open cursor\_name ( &paral, &para2,----);

**Eg2:**

Open cursor\_name (value1,value2,----);

**WAP to display given dept employees data using cursor with parameter.**

declare

Cursor cl(vdno number) is select \* from emp where deptno=vdno;

E emp%rowtype;

Begin

Open cl(&vdno);

Loop

Fetch cl into e;

Exit when cl %notfound;

Dbms\_output.put\_line(E.EMPNO || ' ' || e.ename || ' ' ||e.deptno);

End loop;

Close cl;

End;

**WAP to display all 20th dept and ANALYSIS data using cursor with parameter**

Declare

Cursor c1 (vdno number, vjob varchar )Is select \* from emp where deptno=vdno and job=vjob;

E emp%rowtype;

Begin

Open c1(20,'ANALYST');

Loop

Fetch c1 into e;

Exit when c1%notfound;

Dbms\_output.put\_line(e.empno|| ' ' ||e.ename || ' ' ||e.job || ' ' ||e.sal || ' ' ||e.deptno);

End loop;

Close c1;

End;

/

**Opening cursor more than once in program**

Example:

declare

Cursor c1 (vdno number) is select \* from emp where deptno=vdno;

E emp%rowtype;

Begin

Open c1(&vdno);

Loop

Fetch c1 into e;

Exit when c1%notfound;

If(e.sal<1000) then

update emp set sal=sal+1000 where empno=e.empno;

dbms\_output.put\_line(e.empno || ' salary modified');

end if;

end loop;

close c1;

open c1(&vdno);

loop

fetch c1 into e;

exit when c1%notfound;

if(e.comm is null) then

update emp set comm=500 where empno=e.empno;

dbms\_output.put\_line(e.empno || ' commission modified');

end if;

end loop;

close c1;

end;

/

**Cursor for loop**

**Syntax**

For <variable> in <cursor-name> loop

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

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

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

End loop;

Example :

For I in c1 loop

-----------

-----------

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

End loop;

Advantage of cursor for loop

If cursor for loop is used, following cursor operations not required in the program.

1) Declaring variable to hold data

2) Opening cursor

3) Fetching data from cursor

4) Closing cursor

**WAP to display data from dept table using cursor for loop**

declare

Cursor c1 is select \* from dept;

Begin

For I in c1 loop

Dbms\_output.put\_line(i.deptno || ' ' || i.dname || ' ' || i.loc);

End loop;

End;

/

**WAP to find net salary for all employees in emp table**

declare

Cursor c1 is select \* from emp;

net number(5);

Begin

For I in c1 loop

net := i.sal + nvl(i.comm, 0);

Dbms\_output.put\_line(i.empno ||' ' || i.ename || ' ' || i.sal || ' ' || net);

End loop;

End;

/

**WAP to find experience for all employees belongs to given job;**

declare

Cursor c1(vjob varchar ) Is select \* from emp where job = vjob;

exp number(5);

Begin

For i in c1('&vjob') loop

exp :=round(months\_between(sysdate, i.hiredate)/12);

Dbms\_output.put\_line(i.empno || ' ' || i.ename || ' ' || i.hiredate || ' ' ||exp);

End loop;

End;

/

**WAP to display data from salgrade table using cursor for loop**

declare

Cursor c1 is select \* from salgrade;

Begin

For I in c1 loop

Dbms\_output.put\_line(i.grade || ' ' || i.losal || ' ' ||i.hisal);

End loop;

End;

/

**Implicit cursor or SQL cursor**

Cursor created by oracle while processing SQL statements. Oracle automatically close cursor after processing SQL statement.

Cursor operations open, fetch etc. are not valid for implicit cursor

**Implicit cursor attributes**

1)SQL%found -🡪 returns true if DML operation success.

2)SQL%not found 🡪returns true if DML operation fails

3)SQL%rowcount 🡪returns true no of record processed

SQL cursors are used to know the status of DML operations in the program

**WAP to know the status of insert operation**

declare

Begin

Insert into dept values (95,'exports','hyd');

If (SQL%found ) then

Dbms\_output.put\_line('Record inserted');

End if;

End;

/

set serveroutput on;

set verify off;

**WAP to know the status of update statement**

declare

Vdno emp.empno%type :=&vno;

Begin

Update emp set sal =sal + 1000 where empno=vdno;

If(SQl %found) then

Dbms\_output.put\_line('employee sal modified');

Else

Dbms\_output.put\_line('no data for given number');

End if;

End;

**WAP to count the number of the records deleted from table**

declare

Vdno emp.empno%type :=&vno;

Begin

Delete from emp where deptno=vdno;

Dbms\_output.put\_line(' No of records deleted: ' || SQL % rowcount);

Rollback;

End;

**Dynamic or ref cursor** : the select statement associated with cursor is changing at the time of running program.

**Type of dynamic cursors**

1. Strong cursor
2. Weak cursor

**Steps to process dynamic cursor:**

1. Define ref cursor
2. Declare ref cursor variable
3. Open cursor variable for query
4. Fetch data from ref cursor variable
5. Close ref cursor variable

**Define ref cursor :**

Ref cursor type should be defined in thedeclare part of PL/SQL block

Syntax:-

type ref\_cur\_type is ref cursor return <datatype>

🡪weak cursor will not have return type.

**Declaring ref cursor variable: -**

ref cursor variable should be declared in declare part of PL/SQL block

Syntax :-

Ref\_var ref\_cur\_type;

**Opening ref cursor variable: -**

Ref cursor variable should be opened in executable part of PL/SQL block

Syntax:-

Open ref\_var for select statement.

**Fetching data from ref cursor variable:-**

Syntax:-

Fetch <ref-var> into <PL/SQL variables>

**Closing ref cursor variable**

Syntax:-

Close <ref\_var>;

**Example for strong cursor:-**

declare

Type ref\_cur\_st is ref cursor return emp%rowtype;

E emp%rowtype;

Ref\_stvar ref\_cur\_st;

Begin

Open ref\_stvar for select \* from emp where deptno=10;

Loop

Fetch ref\_stvar into e;

Exit when ref\_stvar%notfound;

Dbms\_output.put\_line(e.empno || ' ' || e.ename || ' ' ||e.job || ' ' || e.deptno);

End loop;

Close ref\_stvar;

Open ref\_stvar for select \*from emp where job='Manager';

Loop

Fetch ref\_stvar into e;

Exit when ref\_stvar%notfound;

If (e.sal<2000) then

Update emp set sal = sal +3000 where empno=e.empno;

Dbms\_output.put\_line(e.empno||' sal modified');

End if;

End loop;

Close ref\_stvar;

End;

/

**Example for weak cursor:**

declare

Type ref\_cursor\_test is ref cursor ;

Ref\_cursor ref\_cursor\_test;

E emp%rowtype;

D dept%rowtype;

Begin

Open ref\_cursor for select \*from dept;

Loop

Fetch ref\_cursor into d;

Exit when ref\_cursor%notfound;

Dbms\_output.put\_line(d.deptno||' '||d.dname||' '||d.loc);

End loop;

Close ref\_cursor;

Open ref\_cursor for select \* from emp;

Loop

Fetch ref\_cursor into e;

Exit when ref\_cursor%notfound;

Dbms\_output.put\_line(e.empno||' '||e.ename||' '||e.job||' '||e.sal||' '||e.deptno);

End loop;

Close ref\_cursor;

End;

/

**Composite data types:**

🡪data types derived from simple data types

🡪These are valid within PL/SQL block

**Types**

1. PL/SQL records
2. PL/SQL tables

PL/SQL records:-

it is collection of dissimilar data types

Syntax:-

type records\_name is record (col1 datatype (size), Col2 datatype (size),

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

-------------------------);

**WAP to display ename, job, sal, hiredate for given employee using PL/SQL record**

declare

type emp\_rec is record (eno number(5),

Name varchar(10),

Job varchar(10),

Sal number(5),

Doj date);

e emp\_rec;

Begin

Select empno, ename, job, sal, hiredate into e.eno, e.name, e.job, e.sal, e.doj from emp where empno=&eno;

Dbms\_output.put\_line(e.eno|| ' ' || e.name || ' ' || e.job || ' ' || e.sal || ' ' || e.doj);

End;

/

**WAP to display HRA, DA, PF for given employee by taking sal as basic**

SQL> declare

type sal\_data is record (eno number (5),

Name varchar(10),

Basic number(5),

HRA emp.sal%type,

DA emp.sal%type,

PF emp.sal%type);

S sal\_data;

Begin

Select empno, ename,sal, sal\*0.35 ,sal\*0.25, sal\*0.15 into s.eno,s.name,s.basic ,s.hra,s.DA,s.PF from emp where empno=&eno;

Dbms\_output.put\_line(s.eno|| ' ' || s.name || ' ' || s.basic || ' ' || s.hra || ' ' || s.da || ' ' || s.PF);

End;

/

**PL/SQL Tables:-**

it is collection of elements of similar DATA types (line array)

Syntax:-

Type <table\_name> is table of <datatype> index by binary\_interger ;

**WAP to display data from dname column of dept table using PL/SQL table**

declare

type dnm\_tab is table of dept.dname%type index by binary\_integer ;

d dnm\_tab;

Cursor c1 is select dname from dept;

i number(5) := 1;

begin

for k in c1 loop

d(i) := k.dname;

I := i+1;

End loop;

For j in 1 .. d.count loop

Dbms\_output.put\_line( d(j));

End loop;

End;

**WAP to display ename, sal for all employees in emp table**

declare

type enm\_tab is table of emp.ename%type index by binary\_integer;

Type enm\_sal is table of emp.sal %type Index by binary\_integer;

N enm\_tab;

S enm\_sal;

Cursor c1 is select ename, sal from emp;

I number (5) :=1;

Begin

For k in c1 loop

N(i) := k.ename;

S(i) := k.sal;

I :=i+1;

End loop;

For j in 1 .. n.count loop

Dbms\_output.put\_line(n(j) || ' ' || s(j));

End loop;

End;

/

**Binding process of assigning values to PL/SQL variables in select statement**

Bulk binding :-

Bulk binding entire collection at once;

“Bulk collect” key word is used for bulk binding

declare

Type dnm\_tab is table of dept.dname%type index by binary\_integer;

d dnm\_tab;

begin

select dname bulk collect into d from dept;

for j in 1 .. d.count loop

Dbms\_output.put\_line(d(j));

End loop;

End;

/

Object:-

it is database component permanently stored in database as data type. It is similar to PL/SQL records but permanent

Syntax:-

Create or replace type <typename> as object

(col1 datatpe (size), col2 datatype(size), …..);

Eg:

create or replace type adrs\_type as object (hno number(5), street varchar(20), City varchar(10), pin number(15));

**Data dictionary view for objects**

User\_types

SQL>desc user\_types;

**Dropping object:-**

**Syntax :**

**Drop type <type\_name>;**

**SQL>drop type adrs\_type;**

**SQL>** **create or replace type address\_type as object (hno varchar(10), street varchar(10), City varchar(10), pin number(5));**

**/**

**SQL>** **create table employee (empno number(5),**

**ename varchar(10), address address\_type);**

**SQL>** **insert into employee values(7369, 'smith',**

**Address\_type('11 a/10' ,'str1', 'hyd', 50012));**

**SQL>** **insert into employee values(7345,'allen',**

**Address\_type('22-b/11' ,'str2', 'sec', 50018));**

SQL>select \* from employee;

**WAP to display all employee belongs to HYD.city**

SQL> select \* from employee e where e.address.city='hyd';

**modify the hno of employee 7369.**

SQL> update employee e set e.address.hno ='33-a/16' where empno=7369;

SQL> select address from employee;

**Delete all employees belongs to sec city**

SQL> delete from employee e where e.address.city = 'sec';

SQL> select address from employee;

Nested object 🡪 object within an object

SQL> create or replace type person\_type as object

(name varchar(10),

Address address\_type);

/

SQL> create table customer ( custid number(5), Cdetails person\_type);

SQL> insert into customer Values (100, person\_type ('smith', address\_type('11-2/10', 'str1', 'hyd',50013)));

SQL> insert into customer Values (200, person\_type('allen', address\_type('11 8/15', 'str2', 'sec',50016)));

SQL> select \* from customer;

Display all records in customer table

**WAP to display all the customer s belongs to ‘HYD’ city .**

SQL> select \* from customer c Where c.cdetails.address.city = 'hyd' ;

**Modify the street name of customer with custid 200;**

SQL> update customer c Set c.cdetails.address.street = 'str5' where custid =200;

SQL> select \* from customer;

**Delete all the customers belongs to SEC city/**

SQL> delete from customer c where c.cdetails.address.city='sec';

SQL> select \* from customer;

**EXCEPTIONS**

All the PL/SQL errors are classified into two types

1. compile time errors
2. runtime errors

**1) Compile time errors:-**

Errors occurred at the time of compiling the program.

🡪These are reported by PL/SQL compile time engine

🡪User has to rectify these errors

**2) Runtime errors:**

Errors occurred at the time of running program

🡪These are reported by PL/SQL Run time engine.

🡪Exceptions are designed to handle run time errors.

🡪Exceptions & exception handlers are the mechanisms by which program reacts deals with run time errors.

Syntax for PL/SQL block with exception part:

Declare

<variable declaration>;

Begin

<executable statements>;

Exception

When <exception1> then

----------

----------

----------

When <exception2 > then

----------

----------

----------

When others then

----------

----------

----------

End;

🡪If any error occurs in the executable part of PL/SQL block control jumps into exception part without executing remaining statements in executable part.

Types of exceptions:

1. Pre-defined exceptions
2. User defined exceptions

Predefined exceptions:

Oracle has several predefined exceptions corresponds to most common oracle errors.

|  |  |
| --- | --- |
| Exception | Description\_ |
| DUP\_VAL\_ON\_INDEX | Violation of unique constraint |
| LOGIN\_DENIED | Invalid user name or password |
| INVALID\_CURSOR | Cursor operation performed on invalid cursor |
| TOO\_MANY\_ROWS | Query return more than one value |
| NO\_DATA\_FOUND | No data in data base |
| CURSOR\_ALREADY\_OPEN | An attempt to open already opened cursor |
| VALUE\_ERROR | Data type or size miss match |
| ZERO\_DEVIDE | Division with zero |

**WAP to handle no\_data\_found exception**

declare

Vno emp.empno%type:=&emp\_num;

Vname emp.ename%type;

Vjob emp.job%type;

Begin

Select ename, job into vname, vjob from emp where empno=vno;

Dbms\_output.put\_line(vno||' ' ||Vname||' '||vjob);

Exception

When no\_data\_found then

Dbms\_output.put\_line('no data for given number');

When others then

Dbms\_output.put\_line('exception in the program');

End;

/

**WAP to handle too\_many\_rows exceptons**

declare

Vno emp.empno%type:=&ENUM;

Vname emp.ename%type;

Vjob emp.job%type;

Vsal emp.sal%type;

Begin

Select ename, job ,sal into vname,vjob, vsal from emp where empno=vno;

Dbms\_output.put\_line(vno||' '||Vname||' '||vjob||' '||vsal);

Exception

When too\_many\_rows then

Dbms\_output.put\_line('duplicate record exists in table');

When No\_data\_found then

Dbms\_output.put\_line('no daa for given number');

When others then

Dbms\_output.put\_line('errors in the program');

End;

/

**WAP to handle value\_error exception**

declare

Vno emp.empno%type:=&ENUM;

Vjob varchar(10);

Begin

Select job into vjob from emp where empno=vno;

Dbms\_output.put\_line(vno||' '||vjob);

Exception

When value\_error then

Dbms\_output.put\_line('issue with variables');

When No\_data\_found then

Dbms\_output.put\_line('no data for given number');

When others then

Dbms\_output.put\_line('errors in the program');

End;

/

**WAP to handle zero\_divide exception**

SQL> declare

a number(5):=&a;

b number(5):=&b;

r number(5);

begin

r:=a/b;

dbms\_output.put\_line('value in r is :'||r);

exception

when zero\_divide then

dbms\_output.put\_line('b value should not be zero');

when others then

dbms\_output.put\_line('error in the program');

end;

/

User defined exceptions:

Exceptions created by user

Exceptions should be declared part and raised in executable part and handled in exception handling part of PL/SQL block

Syntax:

Declare

------

--------

--------

<exception \_name> exception;

------

--------

--------

Begin

------

--------

--------

Raise <exception\_name>;

------

--------

--------

Exception

When <exception\_name>then

------

--------

--------

When <exception2>then

------

--------

--------

When others then

-----

----

------

End;

Eg:

User defined exception

declare

Vno emp.empno%type:=&ENUM;

Vsal emp.sal%type;

Vcom emp.comm%type;

Net number(5);

null\_val\_exc exception;

Begin

Select sal, comm into vsal , vcom from emp where empno=vno;

If (vcom is null)then

Raise null\_val\_exc;

Else

Net:=vsal+vcom;

Dbms\_output.put\_line(vno||' '||vsal||' '||vcom||' '||net);

End if;

Exception

When null\_val\_exc then

Dbms\_output.put\_line(vno||' '||'commission null');

When no\_data\_found then

Dbms\_output.put\_line('no data for given number');

When others then

Dbms\_output.put\_line('error in the program');

End;

SQL>create table emp\_hisal(empno number(5),

Ename varchar(10),

Job varchar(10),

Sal number(5));

Write a program to insert employees data into emp\_hisal table in exception part if sal>=3000;

declare

Vno emp.empno%type:=&ENUM;

Vname emp.ename%type;

Vjob emp.job%type;

Vsal emp.sal%type;

val\_out\_of\_exc exception;

Begin

Select ename,job,sal into vname,vjob,vsal from emp where empno=vno;

If(vsal>=3000)then

Raise val\_out\_of\_exc;

Else

Dbms\_output.put\_line(vno||' '||vname||' '||vjob||' '||vsal);

End if;

Exception

When val\_out\_of\_exc then

Insert into emp\_hisal values(vno,vname,vjob,vsal);

Dbms\_output.put\_line(vno||' Data inserted into table');

When others then

Dbms\_output.put\_line('error in the program');

End;

/

select \* from emp\_hisal;

**SUBPROGRAMS**

**PL/SQL blocks are classified into two types**

1) Anonymous block

2) Named block

**1) Anonymous block** 🡪 set of PL/SQL statements without any name

🡪these are dynamically created &executed only once

🡪these are not saved in database

🡪 Anonymous block cannot be called from another Anonymous block or named block

Eg:

All PL/SQl programs discussed in previous classes.

**2. Named block:-**

🡪Set of PL/SQL statements with a name is called named block

🡪These blocks are saved in database

🡪Named block can be called from another named block or Anonymous block

Eg:

procedures, function, triggers, packages

**Subprograms:-**

Procedures & functions are collectively known as subprograms

🡪Procedure is a named PL/SQL block, takes parameters and perform an action

🡪Procedure contains

1. Header
2. Declare part
3. Executable part
4. Optional exception handling part

Syntax:-

Create or replace procedure < procedure \_name>[(param1 [mode] data\_type, param2[mode] data\_type, ….)}

Is/as

[<variable declaration>]

Begin

<executable session>

[exception <exception section>]

End;

**Creating procedures without parameters**

**WAP to display a message**

SQL >create or replace procedure first\_proc

IS

Begin

Dbms\_output.put\_line('welcome to procedures');

End;

/

Note: procedure created.

**Executing procedure from another block**

begin

first\_proc;

end;

**At command line**

**SQL>**set serveroupt on ;

SQL > exec first\_proc;

o/p : welcome to procedures

**Another Way:**

begin

first\_proc();

end;

**At command line :**

**Sql>** callfirst\_proc();

**WAP to find the sum of two number**

create or replace procedure sum\_proc

Is

A number(5) :=&a;

B number(5) :=&b;

c number(5) ;

begin

c:= a+b;

dbms\_output.put\_line('sum is :' ||c);

end;

/

Note:

procedure created

**execution:**

begin

sum\_proc();

end;

Enter value for a:10

Enter value for b:20;

**Calling procedure from another procedure**

create or replace

procedure my\_fisrt

Is

Begin

dbms\_output.put\_line('Calling one procedure');

my\_second();

end;

**second procedure**

create or replace

procedure my\_second

Is

Begin

dbms\_output.put\_line('Calling Two procedure');

end;

calling :

begin

my\_fisrt();

end;

**Write a Procedure to display data from dept table using cursor.**

create or replace procedure dept\_proc

Is

Cursor c1 is select \* from dept;

D dept %rowtype;

Begin

Open c1;

Loop

Fetch c1 into d;

Exit when c1%notfound;

dbms\_output.put\_line(d.deptno || ' '|| d.dname || ' ' ||d.loc);

end loop;

Close c1;

end;

**execution:**

begin

dept\_proc();

end;

**Creating procedures with parameters:--**

Parameters are used to pass values from calling environment to procedure or from procedure to calling env parameters will have three modes.

1. **IN (default mode )**
2. **OUT**
3. **IN OUT**

|  |  |
| --- | --- |
| **MODE** | **DESCRIPTION** |
| IN | Pass values from calling environment to procedure |
| OUT | return value from procedure to calling environment. |
| IN OUT | Pass values from calling environment to procedure and return value from procedure to calling environment. |

**WAP to find the product of two numbers use in parameter read values**

create or replace procedure num\_prod (a number , b number )

is

C number (5);

Begin

c:=a\*b;

dbms\_output.put\_line('product is : '||c);

end;

/

begin

num\_prod (10,20);

end;

**Executing procedure:**

o/p 🡪 product is : 200

**Calling procedure from anonymous block:-**

declare

M number (5) := &M;

N number (5) := &N;

Begin

num\_prod(M,N);

End;

/

**WAP to display ename, job, sal for given employee use in parameter to read empno**

create or replace procedure fetch\_emp\_data(vno number)

Is

Vname emp.ename%type;

Vjob emp.job%type;

Vsal emp.sal%type;

Begin

SELECT ename, job, sal into Vname,Vjob, Vsal from emp where empno= vno;

Dbms\_output.put\_line(vno|| ' ' || vname || ' ' || vjob || ' ' || vsal);

end;

Begin

fetch\_emp\_data(8001);

end;

WAP procedure to display J**ob of employee.**

**Use in parameters to read empno & use out parameter to display job.**

create or replace procedure fetch\_emp\_job (vno number, vjob out varchar2)

Is

Begin

Select job into vjob from emp where empno=vno;

End;

**Executing procedure**

declare

M number (5) := &M;

myjob varchar2(50);

Begin

fetch\_emp\_job (M, myjob );

dbms\_output.put\_line(myjob);

End;

/

**WAP to find net & Experiance of given employee use in parameter to read empno &use out parameters to display net & exp.**

create or replace procedure net\_exp\_proc

(vno number, n out number, e out number)

Is

Vsal emp.sal%type;

Vcom emp.comm%type;

Vdoj emp.hiredate%type;

Begin

Select sal, comm, hiredate into Vsal, Vcom , Vdoj from emp where empno =vno;

n :=vsal+nvl(vcom,0);

e:=round(months\_betwwn(sysdate,vdoj)/12);

End;

/

**Executing procedure :-**

declare

vnet number;

vexp number;

Begin

net\_exp\_proc(7369, vnet , vexp);

dbms\_output.put\_line(vnet||' '|| vexp);

End;

/

**WAP to find cube of given number using INOUT parameter**

create or replace procedure cube\_proce( n in OUT number)

Is

Begin

n := n\*n\*n;

End;

/

**Executing procedure**

declare

vres number:=&NUM;

Begin

cube\_proce(vres);

dbms\_output.put\_line(vres );

End;

/

**Creating procedure with default values for parameters**

SQL> create sequence s1 increment by 10 start with 10;

SQL> create table mydept as select \* from dept where 1=2;

create or replace

procedure insert\_dept\_record (vdname varchar default 'unkown',

Vloc varchar default 'unkown')

Is

Begin

Insert into mydept values(s1.nextval, vdname, vloc);

End;

**Executing procedure**

**Begin**

**insert\_dept\_record();**

**End;**

**/**

**Begin**

**insert\_dept\_record('** RESEARCH **','** DALLAS **');**

**End;**

**Begin**

**insert\_dept\_record('ACCOUNTING','NEWYORK');**

**End;**

**Begin**

**insert\_dept\_record('** SALES **','** CHICAGO **');**

**End;**

SQl>select \* from mydept;

|  |  |  |
| --- | --- | --- |
| DEPTNO | DNAME | LOC |
| 10 | UNKNOW | UNKNOW |
| 30 | RESEARCH | DALLAS |
| 20 | ACCOUNTING | NEWYORK |
| 40 | SALES | CHICAGO |

**GRANTING EXECUTE PERMISSSION ON PROCEDURE**

SYNTAX :-

Grant execute on procedure\_name to <username>;

SQL>create or replace procedure empdept\_proc

(vno number)

Is

Vname emp.ename%type;

Vdname emp.dname%type;

Begin

Select ename,dname into vname,vdname from emp,dept where emp.deptno=dept.deptno and empno=vno;

Dbms\_output.put\_line(vno|| ‘ ‘ || vname|| ‘ ‘ || vdname);

End;

/

**Executing procedure**

SQL>exec empdept\_proc;

SQL>grant execute on empdept\_proc to steve;

**Steve login**

**SQL>** select \* from scott.dept;

Error since no select privilege

**SQL>** select \* from scott.emp;

Error since no select on emp

SQL>exec scott.empdept\_proc(7369);

o/p : 7369 SMITH RESEARCH

**Data dictionary view for procedure**

user\_source

**displaying text of exiting procedure**

SQL> select text from user\_source where name = ‘EMPDEPT\_PROC’;

**DROPPING PROCEDURE**

SYNTAX:-

Drop procedure < procedure name>;

SQL> Drop procedure insert\_dept\_record;

**FUNCTIONS**

🡪Function is a named PL/SQL block, takes parameter & return value to calling environment

🡪Function contains

1) Head

2) Declare part

3) executable part

4)optimal exeption handling part

🡪function contains return clause in header & return statement in executable part

🡪function can be called from select statement

Syntax:-

Create or replace function <function\_name>

[(para1[mode] datatpe , para2 [mode] datatype, ……..)]

Return datatype

Is / as

<variable declaration>;

Begin

<executable statement>;

Return (value or variable );

Exception

<exception handlers>;

End;

/

**WAP to find the product of two given numbers**

create or replace

function prod\_fun(a number ,b number)

Return number

Is

C number(5);

Begin

C := a\*b;

Return ( c ) ;

End;

**Executing function**

**Sql>** select prod\_fun( 5,6 ) from dual;

(or)

declare

vprod number;

Begin

vprod:=prod\_fun(5,6);

dbms\_output.put\_line(vprod);

End;

/

**WAP to find the number of records in emp table**

create or replace function emp\_count

Return number

Is

C number (5) ;

Begin

Select count (\*) into c from emp;

Return ( c) ;

End;

**execution**

sql>select emp\_count from dual;

**using plsql block,**

declare

vcount number;

Begin

vcount:=emp\_count();

dbms\_output.put\_line(vcount);

End;

/

**WAF to find the average salary of given dept in emp table**

SQL> create or replace function avg\_fun

(vdno number)

Return number

Is

A number(5);

Begin

Select avg(sal) into a from emp where deptno=vdno;

Return ( a );

End;

/

**Execution:**

SQL> select avg\_fun(10) from dual;

**WAF to return job of given employee**

SQL> create or replace function job\_fun

(vno number )

Return varchar

Is

Vjob emp.job%type;

Begin

Select job into vjob from emp where empno = vno;

Retuen (vjob);

End ;

**/**

**Sql>** select job\_fun(8001) from dual;

**Grant execute permission on function**

**Syntax:-**

Grant execute on function\_name to <username>;

**DATA DICTIONARY VIEW FOR FUNCTIONS:-**

User\_source

**Dropping function**

Syntax :

Drop function <function\_name>;

SQL> Drop function job\_fun;

**Procedure vs function**

🡪both procedure & function contains

Header, declare part, executable part and optional exception handing part

* Both will support positional, named notations
* Procedure can return more than one value through its out parameters but function only one value.
* Function contains return clause in HEADER and return statement in EXECTABLE format
* Function always returns value but procedure may or may not return value
* Function can be called from select statement but procedure cannot be called from select statement
* Procedure are used to implement business logics but function are used for numeric calculations

WAP to return entire record of table.

SQL> create or replace procedure emp\_rec

(vno number , e out emp %rowtype )

Is

Begin

Select \* into e from emp where empno=vno;

End;

/

Calling above procedure from anonymous block

declare

Eno emp.emp%type :=&enum;

X emp%rowtype;

Begin

Emp\_rec (eno, x );

Dbms\_output.put\_line(x.empno || ‘ ‘ || x.ename || ‘ ‘||x.deptno );

End;

/

**PACKAGES**

🡪 Packages are PL/SQL construct that allows to store related objects together

* Package contains variable , cursors, exceptions and sub programs specifications
* Every Package contains two parts

1. Package specification or package header
2. Package body

**1.Package specification**

🡪 it contains variable , cursors, exceptions declaration and sub programs specifications

🡪 Variables, cursors, exceptions declared in package header can be used in anonymous block or named block without declaration

**Syntax:-**

Create or replace package < package\_name>

Is /as

Variable declaration;

Cursor declaration;

Exception declaration;

sub program specification;

end [package\_name];

**2.Package body**

If contains the code of sub programs specified in package header

Package body successfully compiled after the successful compilation of Package header

**Syntax:-**

create or replace package body < package\_name>

Is /as

Code of subprograms specified in package header;

End;

/

**WAP to declare Variables, cursors and exceptions**

SQL> create or replace package vpack

Is

A number (5);

B number (5);

e emp%rowtype;

Cursor c1 is select \* from emp;

my\_exce exception;

End;

/

**WA program to find the difference of two numbers using package variables**

SQL> declare

C number(5);

Begin

Vpack.a :=&a;

Vpack.b :=&b;

C:=vpack.a-vpack.b;

Dbms\_output.put\_line(‘difference is : ‘||c);

End;

/

declare

C number(5);

Begin

Vpack.a :=&a;

Vpack.b :=&b;

C:=vpack.a-vpack.b;

Dbms\_output.put\_line('difference is : '||c);

End;

/

**WAP to display emp table data using cursor in package**

declare

Begin

Open Vpack.c1;

Loop

Fetch Vpack.c1 into vpack.e;

Exit when Vpack.c1 % notfound;

Dbms\_output.put\_line(vpack.e.empno || ' '||vpack.e.ename || ' '||vpack.e.job || ' '|| vpack.e.deptno);

End loop;

Close vpack.c1;

End;

/

**WAP to find the sum & product of two values use procedure to find sum and function to find product.**

**Step1:**

create or replace package SP\_PK

as

Procedure sum\_proc(a number, b number);

Function prod\_fun(a number, b number)

Return number;

End SP\_PK;

Step2:

create or replace package body SP\_PK

as

Procedure sum\_proc(a number, b number)

Is

C number(5);

Begin

C:=a+b;

Dbms\_output.put\_line('sum is:'||c);

End;

Function prod\_fun(a number, b number)

Return number

Is

C number(5);

Begin

C:=a\*b;

Return( c);

End;

End SP\_PK;

**Execution:**

begin

SP\_PK.sum\_proc(5,10);

end;

o/p: sum is:15

Executing function in package

declare

vprod number(5);

begin

vprod:=SP\_PK.prod\_fun(5,10);

dbms\_output.put\_line('product is : ' || vprod);

end;

O/P: product is : 50

Calling function from select statement

SQL> select SP\_PK.prod\_fun(5,10) from dual;

o/p: 50

**WAP to find the net & experience of all employees in the emp table use functions to find net and experience in packages**

**Package declaration:**

create or replace package net\_exp\_pack

Is

Function net\_fun(vsal number, vcom number)

Return number;

Function exp\_fun(vdoj date)

Return number;

End net\_exp\_pack;

**Package body**

create or replace package body net\_exp\_pack

IS

Function net\_fun(vsal number, vcom number)

Return number

Is

vnet number(5);

Begin

vnet:=(vsal+nvl(vcom,0));

Return vnet;

End;

Function exp\_fun(vdoj date)

Return number

Is

vexp INTEGER(5);

Begin

vexp:=round(months\_between(sysdate,vdoj)/12);

Return vexp;

End;

End net\_exp\_pack;

Calling functions in package from select statement:

SQL> select empno, sal, comm, hiredate, net\_exp\_pack.net\_fun (sal,comm) net,

net\_exp\_pack.exp\_fun(hiredate) exp from emp;

**empno sal comm hiredate net exp**

8001 6000 100 12-DEC-16 6100 1

7001 5000 100 12-DEC-17 5100 0

7369 800 (null) 17-DEC-80 800 37

7499 1600 350 20-FEB-81 1950 37

7521 9000 500 22-FEB-81 9500 37

7566 2975 50 02-APR-81 3025 37

7654 1250 1400 28-SEP-81 2650 36

7698 2850 50 01-MAY-81 2900 37

7782 2450 50 09-JUN-81 2500 37

7788 3000 50 19-APR-87 3050 31

7839 5000 100 17-NOV-81 5100 36

7844 1500 50 08-SEP-81 1550 36

7876 1100 (null) 23-MAY-87 1100 31

7900 950 (null) 03-DEC-81 950 36

7902 3000 60 03-DEC-81 3060 36

7934 1300 (null) 23-JAN-82 1300 36

SUBPROGRAMS OVERLOADING:

More than one procedure or function can be created with same name and different parameters with in the package

Eg:

**Package declaration:**

create or replace package vpack

Is

Function add\_data(a number,b number)

Return number;

Function add\_data(v1 varchar,v2 varchar)

Return varchar;

End;

/

**Package body:**

create or replace

package body vpack

Is

Function add\_data(a number,b number)

Return number

Is

Begin

Return(a+b);

End;

Function add\_data(v1 varchar,v2 varchar)

Return varchar

Is

Begin

Return(v1||' '||v2);

End;

End vpack;

Calling function in package from select statement

SQL> select vpack.add\_data(5,10) from dual;

o/p:

15

SQL> select vpack.add\_data('oracle','apps') from dual;

o/p: oracle apps

**PL/SQL continue statement**:

Syntax:

Declare

-----

-----

-----

begin

-----

-----

-----

Continue(condition)

-----

-----

-----

End;

declare

Cnt number(5):=0;

Begin

For I in 1..10 loop

Cnt:=Cnt+1;

Dbms\_output.put\_line('value of count:'||Cnt);

Continue when Cnt>1;

Cnt:=Cnt+1;

End loop;

End;

o/p:

value of count:1

value of count:2

value of count:3

value of count:4

value of count:5

value of count:6

value of count:7

value of count:8

value of count:9

value of count:10

**TRIGGERS**

Triggers is named PL/SQL block, executes automatically whenever an event takes place on table

🡪Triggers are used to enforce used defined integrity in database

**Types of triggers**

1. DML triggers
2. Instead triggers
3. System triggers

1. DML triggers:- these triggers fires when a DML operation performed on table
2. Instead triggers:- these triggers valid only on views
3. System triggers:- these triggers fires with system related events such as database start-up or shutdown , when user logged in or logged out. These triggers are also fired with DDL operations

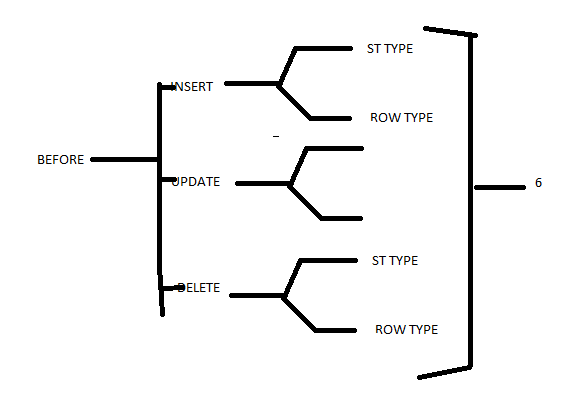
Parts of DML triggers

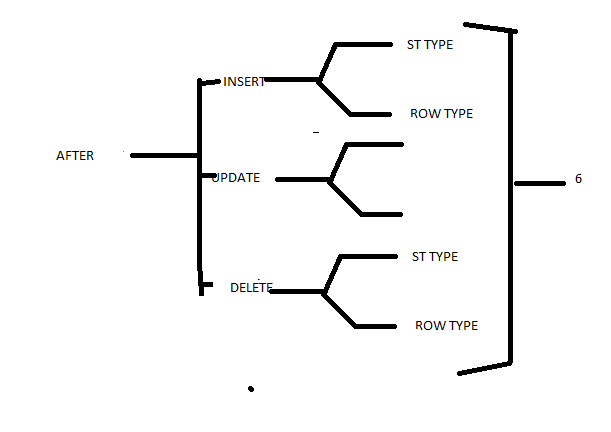
1. Trigger event
2. Trigger timing
3. Trigger type
4. Trigger body

|  |  |  |
| --- | --- | --- |
| Part | Description | possible values |
| Trigger event | which DML operation causes trigger firing | INSERT UPDATE DELETE |
| trigger timing | when trigger fires in relation with triggering event | BEFORE AFTER |
| trigger type | how many times trigger fries | STATEMENT TYPE  ROW TYPE |
| trigger body | what action trigger performs | COMPLETE PL/SQL BLOCK |

Possible DML triggers:

2 timings\*3 events \* 2 types = 12 types





Total =12

* Rowtype trigger fires each &every time when table is effected by triggering event;
* Statement type trigger fires even table is not effected by the triggeing event.

Syntax :

Create or replace trigger <trigger\_name>

Before/after insert or update or delete

On table\_name

[for each row]

[When conditiom]

Declare

<variable declaration>;

Begin

<executable st>

Excepton

<Exception handler>

End;

* Statement type trigger will not have “for each row “ in above syntax
* “Raise\_application\_error” should be used to raise error messages using trigger

Syntax :

Raise\_application\_error(n,’message’);

Range of n is -20,000 to 120999

**Examples on row type triggers**

**WAT to stop all DML operations on smith record in emp table**

create or replace trigger smit\_trgr

Before insert or update or delete on emp

For each row

Begin

If (:new.ename='SMITH' or

:old.ename='SMITH') then

Raise\_application\_error(-20150,'DMLS NOT allowed');

End if;

End;

/

SQL>update emp set sal=sal+1000 where ename='SMITH';

NOTE : ERROR DUE TO TRIGGER

SQL> insert into emp (empno, ename) values (9002,'SMITH');

Note : error due to trigger

**WAT to convert lower case ename into upper case when a record insert into table**

SQL> create or replace trigger convert\_trg

Before insert on emp

For each row

Begin

:new.ename :=upper(:new.ename);

End;

/

SQL > insert into emp (empno,ename) values (9003,'ram');

SQL>select ename from emp where empno=9003;

O/p ->> RAM -🡪 lower case ename converted into upper case due to trigger

WAT to convert upper case ename into lower cases while update operation performed on sal col of emp table

create or replace trigger lower\_trg

Before update of sal on emp

For each row

Begin

:new.ename :=lower (:old.ename);

End;

/

SQL> select ename from emp where empno =7499;

**ENAME**

**ALLEN**

SQL>update emp set sal =sal +1000 where empno=7499;

SQL>select ename from emp where empno=7499;

Ename

Allen

-🡪upper case ename converted into lower case due to trigger

**WAT which is similar to unique, not null constraint or dname column of dept table**

create or replace

trigger uqnn\_trg

Before insert on dept

For each row

Declare

X number(5):=0;

Begin

If (:new.dname is null)then

Raise\_application\_error(-20160,'NULL values not allowed');

End if;

Select count(\*) into x from dept where dname =:new.dname;

If(x>0) then

Raise\_application\_error(-20160,'Duplicates not allowed');

End if;

End;

**Eg:**

**SQL>create table emp\_sal(eno number(5),**

**oldsal number(5),**

**newsal number(5),**

**incr\_date date);**

**WAT to insert old sal, new sal into above table when update performed on sal col of table**

create or replace

trigger sal\_trg

After update of sal

On emp

For each row

Begin

Insert into emp\_sal Values(:old.empno, :old.sal,:new.sal, sysdate);

End;

SQL> update emp set sal = sal+1000 where empno=7369;

SQL>select \* from emp\_sal;

|  |  |  |  |
| --- | --- | --- | --- |
| ENO | OLD SAL | NEW SAL | INCR\_DATE |
| 7369 | 800 | 1800 | 11-JUL-10 |

Eg;

SQL>create table emp\_del(eno number(5),

name varchar(10),job varchar(10),

rem\_date date);

**WAT to copy deleted employees data into above table when records deleted from emp table**

create or replace trigger del\_trg

After delete on emp

For each row

Begin

Insert into emp\_del

Values(:old.empno, :old.ename, :old.job,sysdate);

End;

SQL> delete from emp where empno=7369;

SQL> select \* from emp\_del;

|  |  |  |  |
| --- | --- | --- | --- |
| ENO | NAME | JOB | REM\_DATE |
| 7369 | SMITH | CLERK | 11-JUL-10 |

**Statement type trigger**

Eg:

**WAT to stop all DML operations on emp table before SAM after GPM & SAT or SUN**

create or replace trigger stm\_trg

Before insert or update or delete on emp

Begin

If((to\_char(sysdate,'HH24')

Not between 8 and 18)

Or

To\_char ( sysdate,DY) in ( 'SAT','SUN'));

Raise\_application\_error(-20180,'NO DML allowed in this table');

End if;

End;

/

**Setting user password ( at the command line )**

sqlplus /nolog  
connect / as sysdba  
ALTER USER sys IDENTIFIED BY "new\_password";

system\_user:

GRANT CREATE VIEW TO SCOTT;

INSTEAD TRIGGERS:

* These are valid only on views
* All instead of triggers are rowtype triggers

Sql>create view dept\_view As

Sql>Select\* from dept;

create or replace trigger view\_trg

Instead of insert on dept\_view

For each row

Begin

Delete from dept where deptno=91;

End;

/

SQL>insert into dept\_view ( deptno) values(50);

SQL>select \* from dept;

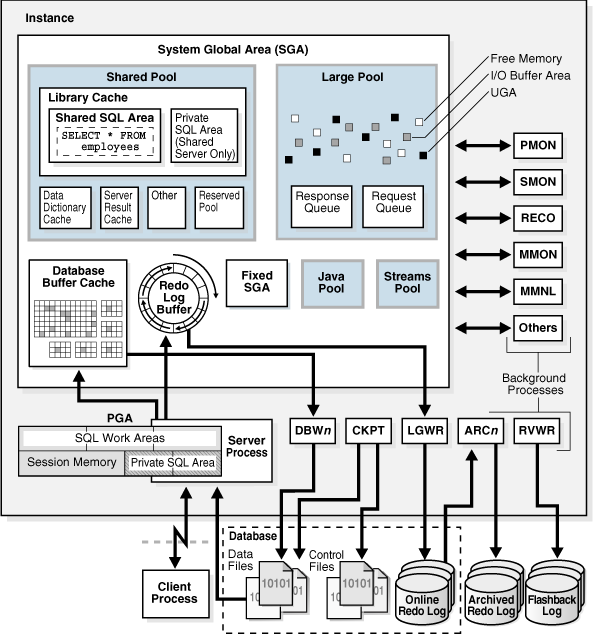
o/p: no rows

since data deleted due to trigger

SQL>after table dept add primary key(deptno);

SQL>after table

**Oracle Database Architecture.**



A database server is the key to information management.

In general, a **[server](https://docs.oracle.com/database/121/CNCPT/glossary.htm" \l "GUID-A3046A66-CC14-47F6-8F91-BDF097F1CA05)** reliably manages a large amount of data in a multiuser environment so that users can concurrently access the same data. A database server also prevents unauthorized access and provides efficient solutions for failure recovery.

***Database and Instance***

An Oracle database server consists of a database and at least one database instance, commonly referred to as simply an instance. Because an instance and a database are so closely connected, the term Oracle database is sometimes used to refer to both instance and database.

In the strictest sense the terms have the following meanings:

* Database

A database is a set of files, located on disk, that store data. These files can exist independently of a database instance.

* Database instance

An instance is a set of memory structures that manage database files. The instance consists of a shared memory area, called the **[system global area (SGA)](https://docs.oracle.com/database/121/CNCPT/glossary.htm" \l "GUID-78C0E867-233A-4857-B9FE-A3852A9B7BDF),** and a set of background processes. An instance can exist independently of database files.

Above [Figure](https://docs.oracle.com/database/121/CNCPT/intro.htm" \l "GUID-8F2EEEC8-0372-4419-88FF-7D77A9C0FCAD__CEGDIFEF) shows a database and its instance. For each user connection to the instance, a **[client process](https://docs.oracle.com/database/121/CNCPT/glossary.htm" \l "GUID-6F3BBCE8-F471-41E8-8C98-F4C62A85862B)** runs the application. Each client process is associated with its own **[server process](https://docs.oracle.com/database/121/CNCPT/glossary.htm" \l "GUID-E660AC1C-B704-4DC1-A35A-DB49EFB34F4A)**. The server process has its own private session memory, known as the **[program global area (PGA)](https://docs.oracle.com/database/121/CNCPT/glossary.htm" \l "GUID-76021F69-AC7A-4D8D-A308-A7B8AC072EEF).**

A database can be considered from both a physical and logical perspective. Physical data is data viewable at the operating system level. For example, operating system utilities such as the Linux ls and ps can list database files and processes. Logical data such as a table is meaningful only for the database. A SQL statement can list the tables in an Oracle database, but an operating system utility cannot.

The database has physical structures and logical structures. Because the physical and logical structures are separate, you can manage the physical storage of data without affecting access to logical storage structures. For example, renaming a physical database file does not rename the tables whose data is stored in this file.

***Database Storage Structures***

An essential task of a relational database is data storage. This section briefly describes the physical and logical storage structures that Oracle Database uses.

***Physical Storage Structures***

The physical database structures are the files that store the data.

When you execute a CREATE DATABASE statement, the following files are created:

* Data files

Every Oracle database has one or more physical data files, which contain all the database data. The data of logical database structures, such as tables and indexes, is physically stored in the data files.

* Control files

Every Oracle database has a **[control file](https://docs.oracle.com/database/121/CNCPT/glossary.htm" \l "GUID-DF498605-D186-451B-87AA-73DF396AD599).** A control file contains metadata specifying the physical structure of the database, including the database name and the names and locations of the database files.

* Online redo log files

Every Oracle Database has an **[online redo log](https://docs.oracle.com/database/121/CNCPT/glossary.htm" \l "GUID-2A8BC112-AB70-4B06-9F85-FE975861CEE0),** which is a set of two or more online redo log files. An online **[redo log](https://docs.oracle.com/database/121/CNCPT/glossary.htm" \l "GUID-F24A4593-5E47-4D06-B9D6-0061D288373C)** is made up of redo entries (also called redo log records), which record all changes made to data.

Many other files are important for the functioning of an Oracle database server. These include parameter files and networking files. Backup files and archived redo log files are offline files important for backup and recovery.

***Logical Storage Structures***

Logical storage structures enable Oracle Database to have fine-grained control of disk space use.

This topic discusses logical storage structures:

* Data blocks

At the finest level of granularity, Oracle Database data is stored in data blocks. One **[data block](https://docs.oracle.com/database/121/CNCPT/glossary.htm" \l "GUID-2141C31D-6752-4260-91CE-99B4CC557247)** corresponds to a specific number of bytes on disk.

* Extents

An **[extent](https://docs.oracle.com/database/121/CNCPT/glossary.htm" \l "GUID-C56D833A-B3D3-4B85-AAB4-334F7CF3F5E9)** is a specific number of logically contiguous data blocks, obtained in a single allocation, used to store a specific type of information.

* Segments

A **[segment](https://docs.oracle.com/database/121/CNCPT/glossary.htm" \l "GUID-EC12AA68-8C89-43B3-B1F9-3AABF7CAEB9F)** is a set of extents allocated for a user object (for example, a table or index), **[undo data](https://docs.oracle.com/database/121/CNCPT/glossary.htm" \l "GUID-297B963A-989C-4720-B061-A2352FF72892)**, or temporary data.

* Tablespaces

A database is divided into logical storage units called tablespaces. A **[tablespace](https://docs.oracle.com/database/121/CNCPT/glossary.htm" \l "GUID-AA66891C-71B2-4D55-8F64-0E427AE24E88)** is the logical container for a segment. Each tablespace consists of at least one data file.

***Database Instance Structures***

An Oracle database uses memory structures and processes to manage and access the database. All memory structures exist in the main memory of the computers that constitute the RDBMS.

When applications connect to an Oracle database, they connect to a [database instance](https://docs.oracle.com/database/121/CNCPT/glossary.htm" \l "GUID-CFB1A30E-76B6-44EA-839E-9E63C8DA31AC). The instance services applications by allocating other memory areas in addition to the SGA, and starting other processes in addition to background processes.

***Oracle Database Processes***

A process is a mechanism in an operating system that can run a series of steps. Some operating systems use the terms job, task, or thread.

For the purposes of this topic, a thread is equivalent to a process. An Oracle database instance has the following types of processes:

* Client processes

These processes are created and maintained to run the software code of an application program or an Oracle tool. Most environments have separate computers for client processes.

* Background processes

These processes consolidate functions that would otherwise be handled by multiple Oracle Database programs running for each client process. Background processes asynchronously perform I/O and monitor other Oracle Database processes to provide increased parallelism for better performance and reliability.

* Server processes

These processes communicate with client processes and interact with Oracle Database to fulfill requests.

Oracle processes include server processes and background processes. In most environments, Oracle processes and client processes run on separate computers.

***Instance Memory Structures***

Oracle Database creates and uses memory structures for program code, data shared among users, and private data areas for each connected user.

The following memory structures are associated with a database instance:

* System Global Area (SGA)

The SGA is a group of shared memory structures that contain data and control information for one database instance. Examples of SGA components include the database buffer cache and shared SQL areas. Starting in Oracle Database 12c Release 1 (12.1.0.2), the SGA can contain an optional [In-](https://docs.oracle.com/database/121/CNCPT/glossary.htm" \l "GUID-A19D33EA-7BAA-42C0-9E13-6AF686E2A976)**[Memory Column Store](https://docs.oracle.com/database/121/CNCPT/glossary.htm" \l "GUID-A19D33EA-7BAA-42C0-9E13-6AF686E2A976)** (IM column store), which enables data to be populated in memory in a **[columnar format](https://docs.oracle.com/database/121/CNCPT/glossary.htm" \l "GUID-DDC39B24-BE95-406A-986F-D760308CA26D).**

* Program Global Areas (PGA)

A PGA is a memory region that contains data and control information for a server or background process. Access to the PGA is exclusive to the process. Each server process and background process has its own PGA.

***Application and Networking Architecture***

To take full advantage of a given computer system or network, Oracle Database enables processing to be split between the database server and the client programs. The computer running the RDBMS handles the database server responsibilities while the computers running the applications handle the interpretation and display of data.

***Application Architecture***

The application architecture is the computing environment in which a database application connects to an Oracle database. The two most common database architectures are client/server and multitier.

* In a **[client/server architecture](https://docs.oracle.com/database/121/CNCPT/glossary.htm" \l "GUID-6F0D8417-F8ED-4D91-BB81-1F86499ADD69),** the client application initiates a request for an operation to be performed on the database server.

The server runs Oracle Database software and handles the functions required for concurrent, shared data access. The server receives and processes requests that originate from clients.

* In a traditional **[multitier architecture](https://docs.oracle.com/database/121/CNCPT/glossary.htm" \l "GUID-A8812375-43CA-4C1A-B376-A59C6FADF50B),** one or more application servers perform parts of the operation.

An **[application server](https://docs.oracle.com/database/121/CNCPT/glossary.htm" \l "GUID-1774BB0F-837E-4651-B1A3-BC468B027B8F)** contains a large part of the application logic, provides access to the data for the client, and performs some query processing. In this way, the load on the database decreases. The application server can serve as an interface between clients and multiple databases and provide an additional level of security.

A **[service-oriented architecture (SOA)](https://docs.oracle.com/database/121/CNCPT/glossary.htm" \l "GUID-2CB01B71-A23D-4127-B54A-E946C23BEDFC)** is a multitier architecture in which application functionality is encapsulated in services. SOA services are usually implemented as Web services. Web services are accessible through HTTP and are based on XML-based standards such as Web Services Description Language (WSDL) and SOAP.

Oracle Database can act as a Web service provider in a traditional multitier or SOA environment.

***Networking Architecture***

Oracle Net Services is the interface between the database and the network communication protocols that facilitate distributed processing and distributed databases.

Communication protocols define the way that data is transmitted and received on a network. Oracle Net Services supports communications on all major network protocols, including TCP/IP, HTTP, FTP, and WebDAV.

**[Oracle Net](https://docs.oracle.com/database/121/CNCPT/glossary.htm" \l "GUID-4F4232CE-6A82-4F39-8565-281AF563F0AE),** a component of Oracle Net Services, establishes and maintains a network session from a client application to a database server. After a network session is established, Oracle Net acts as the data courier for both the client application and the database server, exchanging messages between them. Oracle Net can perform these jobs because it is located on each computer in the network.

An important component of Net Services is the **[Oracle Net Listener](https://docs.oracle.com/database/121/CNCPT/glossary.htm" \l "GUID-504800FA-2159-495B-8E4F-B2D4D938EA85)** (called the listener), which is a process that runs on the database or elsewhere in the network. Client applications send connection requests to the listener, which manages the traffic of these requests to the database. When a connection is established, the client and database communicate directly.

The most common ways to configure an Oracle database to service client requests are:

* Dedicated server architecture

Each client process connects to a **[dedicated server](https://docs.oracle.com/database/121/CNCPT/glossary.htm" \l "GUID-8B14C804-9D68-471C-A581-5AEE673A9FCD)** process. The server process is not shared by any other client for the duration of the client's session. Each new session is assigned a dedicated server process.

* Shared server architecture

The database uses a pool of **[shared server](https://docs.oracle.com/database/121/CNCPT/glossary.htm" \l "GUID-E94CE0E3-CC86-4F46-B8EF-54945F026326)** processes for multiple sessions. A client process communicates with a **[dispatcher](https://docs.oracle.com/database/121/CNCPT/glossary.htm" \l "GUID-D8288F03-1F84-4766-AF5B-B341A714BD3F),** which is a process that enables many clients to connect to the same database instance without the need for a dedicated server process for each client.

***Multitenant Architecture***

Starting in Oracle Database 12c, the multitenant architecture enables an Oracle database to be a multitenant container database (CDB).

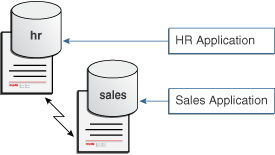
A **[CDB](https://docs.oracle.com/database/121/CNCPT/glossary.htm" \l "GUID-135FF536-DE9B-40CF-9F42-C246762BD77F)** is a single physical database that contains zero, one, or many user-created pluggable databases. A **[pluggable database (PDB)](https://docs.oracle.com/database/121/CNCPT/glossary.htm" \l "GUID-15825EC2-B512-4FD1-A734-AE4DAB3B3AEC)** is a portable collection of schemas, schema objects, and nonschema objects that appears to an **[Oracle Net](https://docs.oracle.com/database/121/CNCPT/glossary.htm" \l "GUID-4F4232CE-6A82-4F39-8565-281AF563F0AE)** client as a non-CDB. A **[non-CDB](https://docs.oracle.com/database/121/CNCPT/glossary.htm" \l "GUID-B2710CAA-8F00-40B3-97AB-4521D2147EE8)** is a traditional Oracle database that cannot contain PDBs.

Starting in Oracle Database 12c, you must create a database as either a CDB or non-CDB. You can plug a non-CDB into a CDB as a PDB. To move a PDB to a non-CDB, you must use Oracle Data Pump.

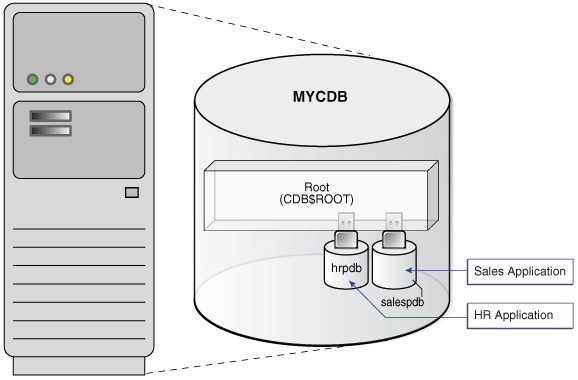
By consolidating multiple physical databases on separate computers into a single database on a single computer, the multitenant architecture provides the following benefits:

* Cost reduction for hardware
* Easier and more rapid movement of data and code
* Easier management and monitoring of the physical database
* Separation of data and code
* Separation of duties between a **[PDB administrator](https://docs.oracle.com/database/121/CNCPT/glossary.htm" \l "GUID-6E44D9F6-9FE1-4FEA-998A-51ACA5775AF5),** who manages only the PDBs to which she or he is granted privileges, and the **[CDB administrator](https://docs.oracle.com/database/121/CNCPT/glossary.htm" \l "GUID-24233CA1-CF2F-4E02-B202-83DB0571DDB9),** who manages the entire CDB

**Below Figure** shows two separate non-CDBs: hr and sales. Each non-CDB has its own memory and set of database files, and resides on its own computer. Each non-CDB has its own dedicated user application.



**[Figure 1-3](https://docs.oracle.com/database/121/CNCPT/intro.htm" \l "GUID-AB84D6C9-4BBE-4D36-992F-2BB85739329F__BHCIEHAI)** shows the same data after being consolidated into the CDB named MYCDB.



Physically, MYCDB is an Oracle database. MYCDB has one database instance (although multiple instances are possible in Oracle Real Application Clusters) and one set of database files, just like a non-CDB.

MYCDB contains two PDBs: hrpdb and salespdb. As shown in Above **Figure,** these PDBs appear to their respective applications just as they did before database consolidation. To administer the CDB itself or any PDB within it, a CDB administrator can connect to the **[CDB root](https://docs.oracle.com/database/121/CNCPT/glossary.htm" \l "GUID-1C63428A-905A-4918-95AA-F3BBA3D2B825),** which is a collection of schemas, schema objects, and nonschema objects to which all PDBs belong.

CDBs and non-CDBs have architectural differences. This manual assumes the architecture of a non-CDB unless otherwise indicated.

**Database Models**

A database model shows the logical structure of a database, including the relationships and constraints that determine how data can be stored and accessed. Individual database models are designed based on the rules and concepts of whichever broader data model the designers adopt. Most data models can be represented by an accompanying database diagram.

## Types of database models

There are many kinds of data models. Some of the most common ones include:

* Hierarchical database model
* Relational model
* Network model
* Object-oriented database model
* Entity-relationship model
* Document model
* Entity-attribute-value model
* Star schema
* The object-relational model, which combines the two that make up its name

You may choose to describe a database with any one of these depending on several factors. The biggest factor is whether the database management system you are using supports a particular model. Most database management systems are built with a particular data model in mind and require their users to adopt that model, although some do support multiple models.

In addition, different models apply to different stages of the database design process. High-level conceptual data models are best for mapping out relationships between data in ways that people perceive that data. Record-based logical models, on the other hand, more closely reflect ways that the data is stored on the server.

Selecting a data model is also a matter of aligning your priorities for the database with the strengths of a particular model, whether those priorities include speed, cost reduction, usability, or something else.

## Relational model

The most common model, the relational model sorts data into tables, also known as relations, each of which consists of columns and rows. Each column lists an attribute of the entity in question, such as price, zip code, or birth date. Together, the attributes in a relation are called a domain. A particular attribute or combination of attributes is chosen as a primary key that can be referred to in other tables, when it’s called a foreign key.

Each row, also called a tuple, includes data about a specific instance of the entity in question, such as a particular employee.

The model also accounts for the types of relationships between those tables, including one-to-one, one-to-many, and many-to-many relationships. Here’s an example:

# Image result for relational data model in dbms images

Within the database, tables can be normalized, or brought to comply with normalization rules that make the database flexible, adaptable, and scalable. When normalized, each piece of data is atomic, or broken into the smallest useful pieces.

Relational databases are typically written in Structured Query Language (SQL). The model was introduced by E.F. Codd in 1970.

## Hierarchical model

The hierarchical model organizes data into a tree-like structure, where each record has a single parent or root. Sibling records are sorted in a particular order. That order is used as the physical order for storing the database. This model is good for describing many real-world relationships.

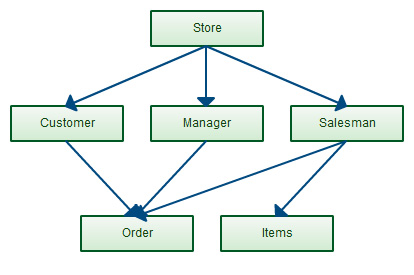
# Image result for hierarchical data model in dbms images

This model was primarily used by IBM’s Information Management Systems in the 60s and 70s, but they are rarely seen today due to certain operational inefficiencies.

## Network model

The network model builds on the hierarchical model by allowing many-to-many relationships between linked records, implying multiple parent records. Based on mathematical set theory, the model is constructed with sets of related records. Each set consists of one owner or parent record and one or more member or child records. A record can be a member or child in multiple sets, allowing this model to convey complex relationships.

It was most popular in the 70s after it was formally defined by the Conference on Data Systems Languages (CODASYL)



## Object-oriented database model

This model defines a database as a collection of objects, or reusable software elements, with associated features and methods. There are several kinds of object-oriented databases:

A **multimedia database** incorporates media, such as images, that could not be stored in a relational database.

A **hypertext database** allows any object to link to any other object. It’s useful for organizing lots of disparate data, but it’s not ideal for numerical analysis.

The object-oriented database model is the best known post-relational database model, since it incorporates tables, but isn’t limited to tables. Such models are also known as hybrid database models.

# 

## Object-relational model

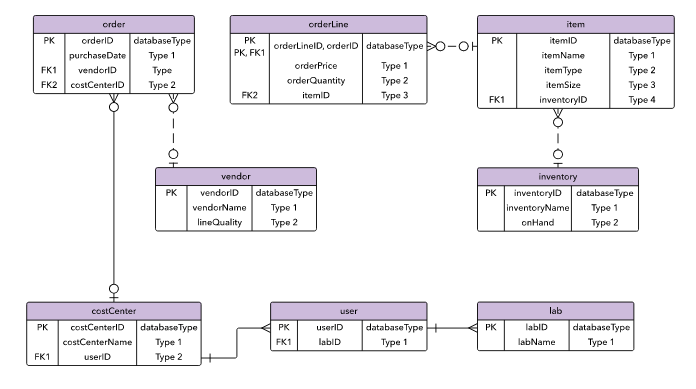
This hybrid database model combines the simplicity of the relational model with some of the advanced functionality of the object-oriented database model. In essence, it allows designers to incorporate objects into the familiar table structure.

Languages and call interfaces include SQL3, vendor languages, ODBC, JDBC, and proprietary call interfaces that are extensions of the languages and interfaces used by the relational model.

## Entity-relationship model

This model captures the relationships between real-world entities much like the network model, but it isn’t as directly tied to the physical structure of the database. Instead, it’s often used for designing a database conceptually.

Here, the people, places, and things about which data points are stored are referred to as entities, each of which has certain attributes that together make up their domain. The cardinality, or relationships between entities, are mapped as well.



A common form of the ER diagram is the star schema, in which a central fact table connects to multiple dimensional tables.

## Other database models

A variety of other database models have been or are still used today.

### Inverted file model

A database built with the inverted file structure is designed to facilitate fast full text searches. In this model, data content is indexed as a series of keys in a lookup table, with the values pointing to the location of the associated files. This structure can provide nearly instantaneous reporting in big data and analytics, for instance.

This model has been used by the ADABAS database management system of Software AG since 1970, and it is still supported today.

### Flat model

The flat model is the earliest, simplest data model. It simply lists all the data in a single table, consisting of columns and rows. In order to access or manipulate the data, the computer has to read the entire flat file into memory, which makes this model inefficient for all but the smallest data sets.

### Multidimensional model

This is a variation of the relational model designed to facilitate improved analytical processing. While the relational model is optimized for online transaction processing (OLTP), this model is designed for online analytical processing (OLAP).

Each cell in a dimensional database contains data about the dimensions tracked by the database. Visually, it’s like a collection of cubes, rather than two-dimensional tables.

### Semi-structured model

In this model, the structural data usually contained in the database schema is embedded with the data itself. Here the distinction between data and schema is vague at best. This model is useful for describing systems, such as certain Web-based data sources, which we treat as databases but cannot constrain with a schema. It’s also useful for describing interactions between databases that don’t adhere to the same schema.

### Context model

This model can incorporate elements from other database models as needed. It cobbles together elements from object-oriented, semi structured, and network models.

### Associative model

This model divides all the data points based on whether they describe an entity or an association. In this model, an entity is anything that exists independently, whereas an association is something that only exists in relation to something else.

The associative model structures the data into two sets:

* A set of items, each with a unique identifier, a name, and a type
* A set of links, each with a unique identifier and the unique identifiers of a source, verb, and target. The stored fact has to do with the source, and each of the three identifiers may refer either to a link or an item.

Other, less common database models include:

Semantic model, which includes information about how the stored data relates to the real world

* XML database, which allows data to be specified and even stored in XML format
* Named graph
* Triplestore

## NoSQL database models:

In addition to the object database model, other non-SQL models have emerged in contrast to the relational model:

The **graph database model**, which is even more flexible than a network model, allowing any node to connect with any other.

The **multivalue model**, which breaks from the relational model by allowing attributes to contain a list of data rather than a single data point.

The **document model**, which is designed for storing and managing documents or semi-structured data, rather than atomic data.