ORACLE

Data :-

Data are raw material and unorganized facts that need to be processed.

Example:-

Students fill an admission form when they get admission in college. The form consists of raw facts about the students. These raw facts are student's name, father name, address etc. The purpose of collecting this data is to maintain the records of the students during their study period in the college.

Information :-

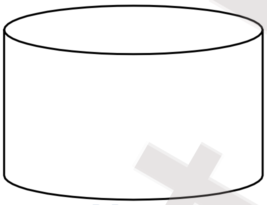
When data are processed , organized, structured or presented in a given context so as to make them useful.They are called information.

Example:-

Data collected from census is used to generate different type of information. The government can use it to determine the literacy rate in the country. Government can use the information in important decision to improve literacy rate.

Database :-

A Database is a collection of meaningful data that is stored in computer system. For example an employee data would generally contain Employee Name, Salary, Address and Job. Such data of multiple employees placed together in one place is termed as Database. DB is represented by using cylinder.



DB

On line Transactional Processing :-

Data is entered initially into OLTP system. The focus of an OLTP system is data entry i.e. insert ,update and delete. An OLTP environment is not suitable for reporting purpose because a normalized model usually involves many tables. Even simple reports requires joining many tables , resulting poor performance.

On line Analytical Processing (Data Warehouse) :-

A data Wrehouse is a DB designed for data retrieval or reporting purpose but not for transaction processing (insert,update,delete).

DBMS :-

A Database management system (DBMS) can be a set of software programs that controls the storage and retrieval of data in a database. It also controls the security and integrity of the database. The DBMS accepts requests for data from the application program and instructs the operating system to transfer the appropriate data.

Data security prevents unauthorised users from viewing or updating the database. Using passwords, users are allowed access to the entire database or subsets of the database, For example, an employee database can contain all the data about an individual employee, but one group of users may be authorised to view only payroll data, while others are allowed access to only work history and medical data. The DBMS can maintain the integrity of the database by not allowing more than one user to update the same record at the same time. Organisations may use one kind of DBMS for daily transaction processing and then move the detail onto another computer that uses another DBMS better suited for analysis.

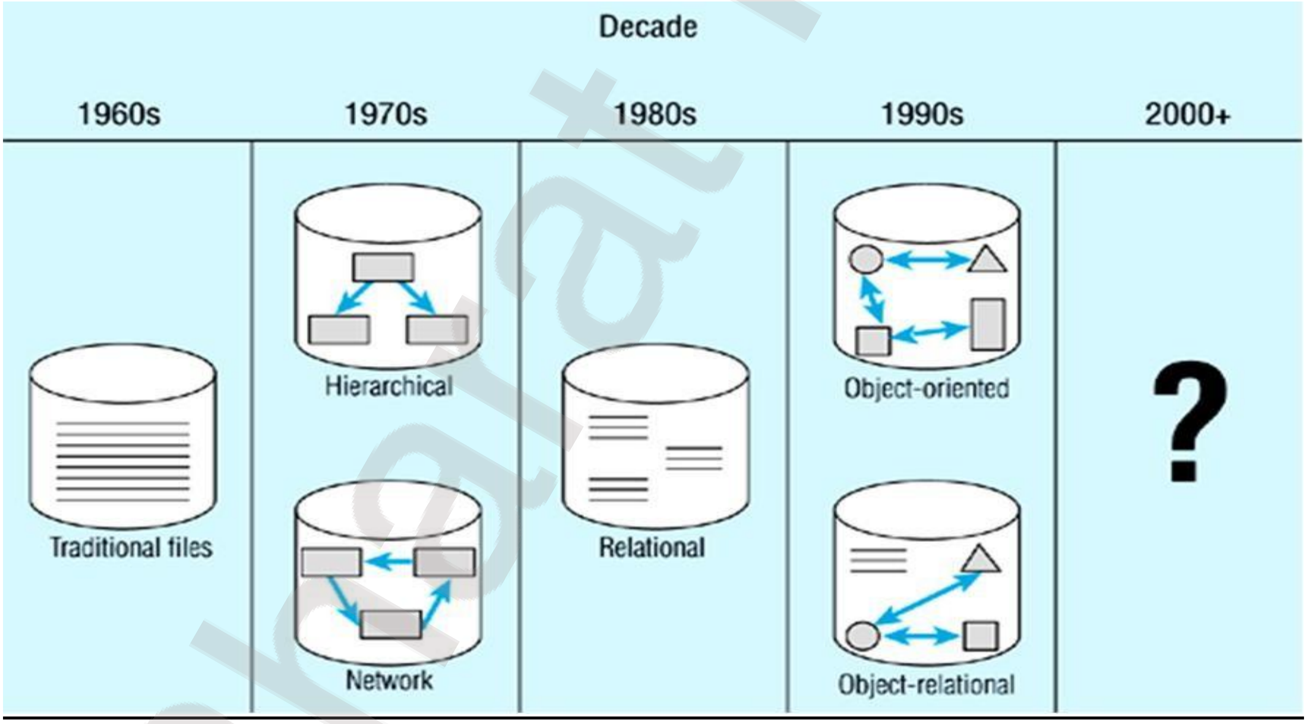
Advantages :-

- Reduced data redundancy
- increased consistency
- Greater data integrity and independence from applications programs
- Improved data access to users through use of host and query languages
- Improved data security
- Reduced data entry, storage, and retrieval costs
- Facilitated development of new applications program

Disadvantages:-

- Database systems are complex, difficult, and time-consuming to design
- Substantial hardware and software start-up costs
- Damage to database affects virtually all applications programs
- Extensive conversion costs in moving form a file-based system to a database system
- Initial training required for all programmers and users

Development of DBMS :-



Types of people involved:-

Three types of people are involved with a general-purpose DBMS:

1. **DBMS developers** - These are the people that design and build the DBMS product, and the only ones

who touch its code. They are typically the employees of a DBMS vendor (e.g., Oracle, IBM, Microsoft)

2. **Application developers** and **Database administrators** - These are the people that design and build a database-based application that uses the DBMS. The latter group members design the needed database and maintain it.
3. **Application's end-users** :- These people know the application and its end-user interfaces, but need neither to know nor to understand the underlying DBMS.

ER MODEL :

an **entity-relationship model** (**ERM**) is a conceptual representation of data. Entity-relationship modeling is a database modeling method, used to produce a type of conceptual schema or semantic data model of a system, often a relational database, and its requirements in a top-down fashion. Diagrams created by this process are called **entity-relationship diagrams**, **ER diagrams**, or **ERDs**.

The building blocks of ER model :-

- **Entities**
- **Attributes**
- **Relationships**

Entity :-

An entity may be a physical object such as a house or a car, an event such as a house sale or a car service, or a concept such as a customer transaction or order.Entities can be thought of as nouns. Examples: a computer, an employee, a vehicle

Entity Set :-

Collection of entities that share common characterstics is called entity set. In ERD entity set is represented by using symbol rectangle.

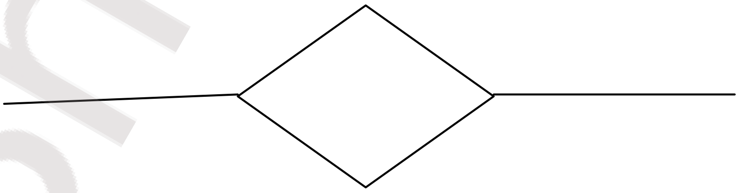
Attributes :-

Entity properties are called attributes,for example Properties of employee entity includes Empno, Ename, Job, Sal etc.

In ERD attributes are represented by using symbol ellipse.

Relationship :-

A relationship is an association between entities of entity sets and it is represented by using



Types of relationships:-

- One-to-one (1:1)
- One-to-many(1:m)

- Many-to-Many(m:n)

One-to-one relationship :-

If one entity of an entity set associated with one entity of another entity set then it is called one-to-one relationship.

One-to-one relationship :-

If one entity of an entity set associated with many entity of another entity set then it is called one-to-many relationship.

many-to-many relationship :-

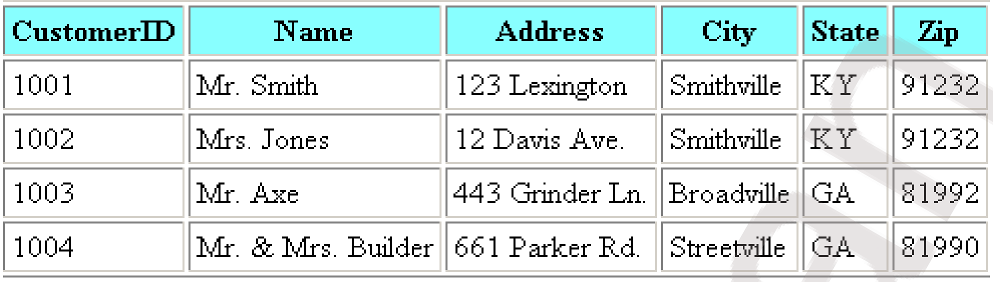
If many entities of an entity set associated with many entities of another entity set then it is called one-to-many relationship.

RDBMS:-

The concept of a relational database was first developed by E.FCODD. The relation, which is a two-dimensional table, is the primary unit of storage in a relational database. A relational database can contain one or more of these tables, with each table consisting of a unique set of rows and columns. A record stored in a table is called row, also known as a tuple, while attributes of the data are defined in columns, or fields, in the table. The characteristics of the data, or the column, relates one record to another. Each column has a unique name and the content within it must be of the same type.
Dr. Codd defined thirteen standards which must be met before a database can be considered to be relational database:
0. A relational DBMS must be able to manage databases entirely through its relational capabilities.

1. **Information rule**– All information in a relational database is represented as values in tables.
2. **Guaranteed access**–Every value in a relational database is guaranteed to be accessible by using a combination of the table name, primary key value, and column name.
3. **Systematic null value support**–The DBMS provides systematic support for the treatment of null values (unknown or inapplicable data), distinct from default values, and independent of any domain.
4. **Active, online relational catalog**–The description of the database and its contents is represented at the logical level as tables and can therefore be queried using the database language.
5. **Comprehensive data sublanguage**–At least one supported language must have a well-defined syntax and be comprehensive. It must support data definition, manipulation, integrity rules, authorization, and transactions.
6. **View updating rule**–All views that are theoretically updatable can be updated through the system.

7. **Set-level insertion, update, and deletion** — The DBMS supports not only setlevel retrievals but also set-level inserts, updates, and deletes.
8. **Physical data independence**–Application programs and ad hoc programs are logically unaffected when physical access methods or storage structures are altered.
9. **Logical data independence**–Application programs and ad hoc programs are logically unaffected, to the extent possible, when changes are made to the table structures.
10. **Integrity independence**–The database language must be capable of defining integrity rules. They must be stored in the online catalog, and they cannot be bypassed.
11. **Distribution independence**–Application programs and ad hoc requests are logically unaffected when data is first distributed or when it is redistributed.
12. **Nonsubversion**–It must not be possible to bypass the integrity rules defined through the database language by using lower-level languages.



CustomerID	Name	Address	City	State	Zip
1001	Mr. Smith	123 Lexington	Smithville	KY	91232
1002	Mrs. Jones	12 Davis Ave.	Smithville	KY	91232
1003	Mr. Axe	443 Grinder Ln.	Broadville	GA	81992
1004	Mr. & Mrs. Builder	661 Parker Rd.	Streetville	GA	81990

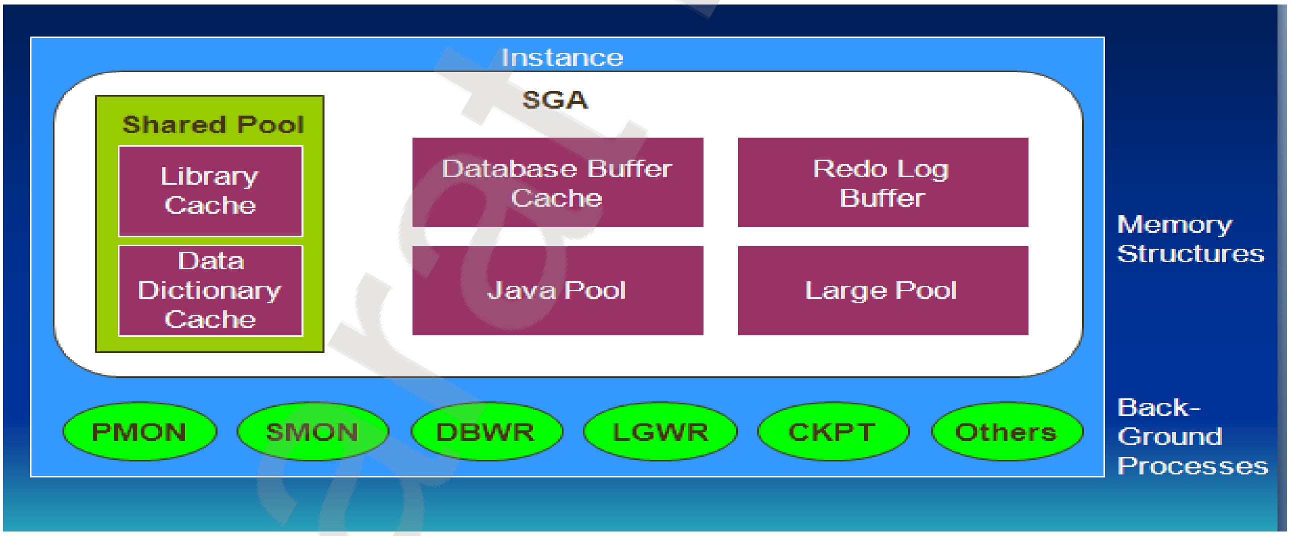
In **RDBMS :-**

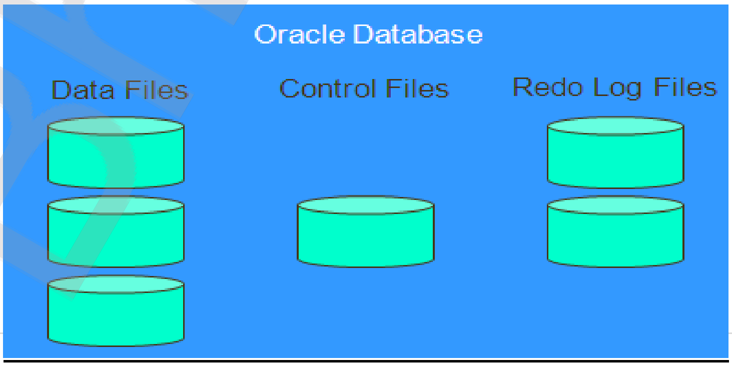
- **A** Database is a collection of tables.
- Each table contains records, which are called tuples
- Each record contains field values
- A **domain** is refers to the possible values each filed can hold
- Each table has a unique identifier called **Primary key**, which is used to access record in a table.
- Tables are related using **Foreign keys**

Oracle Server & Architecture :-

.

The Oracle11g server provides an open, comprehensive, and integrated approach to information management. An Oracle server consists of an Oracle database and an Oracle server instance. Every time a database is started, a system global area (SGA) is allocated, and Oracle background processes are started. The system global area is an area of memory used for database information shared by the database users. The combination of the background processes and memory buffers is called an Oracle instance.





SQL

Structured Query Language (SQL) is the set of statements with which all programs and users access data in an Oracle database.
The language, Structured English Query Language ("SEQUEL") was developed by IBM Corporation, Inc. SEQUEL later became SQL (still pronounced "sequel").
All major relational database management systems support SQL,so you can transfer all skills you have gained with SQL from one database to another. In addition, all programs written in SQL are portable. They can often be moved from one database to another with very little modification.

SQL has the following advantages:

- Efficient
- Easy to learn and use
- With SQL, you can define, retrieve, and manipulate data in the tables

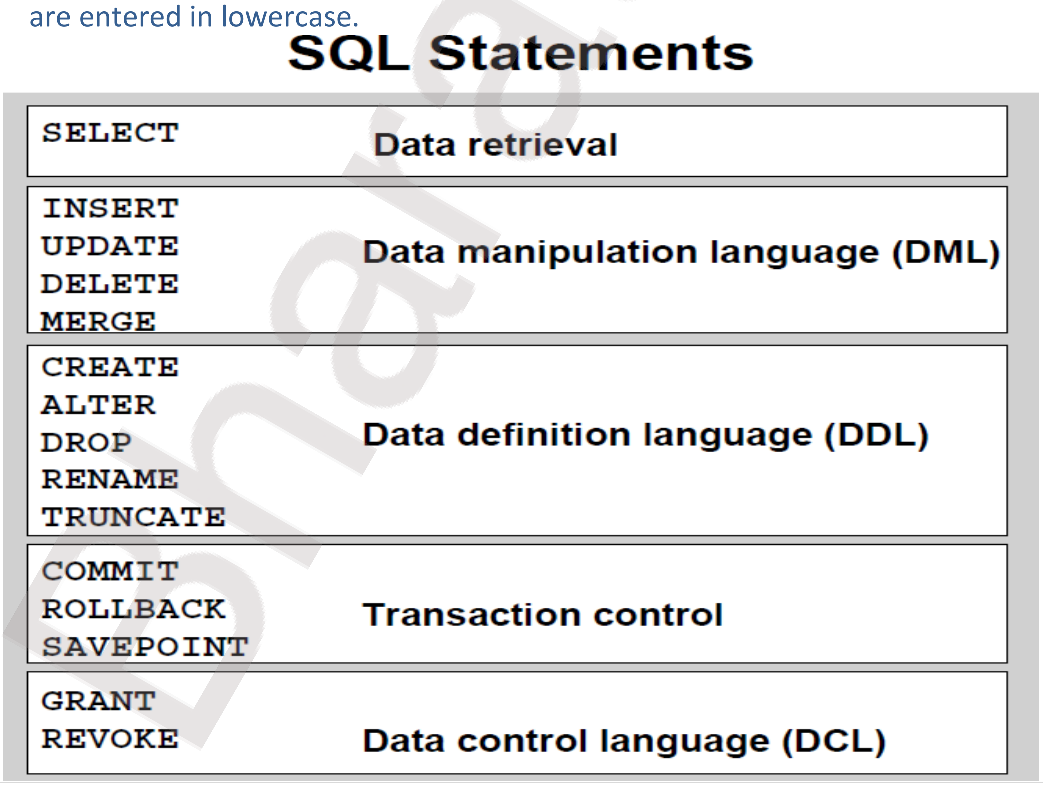
SQL Standards:-

Oracle SQL complies with industry-accepted standards.. Industry-accepted committees are the American National Standards Institute (ANSI) and the International Standards Organization (ISO). Both ANSI and ISO have accepted SQL as the standard language for relational databases.

Writing SQL Statements

Using the following simple rules and guidelines, you can construct valid statements that are both easy to read and easy to edit:

- SQL statements are not case sensitive.
- SQL statements can be entered on one or many lines.
- Keywords cannot be split across lines or abbreviated.
- Clauses are usually placed on separate lines for readability and ease of editing.
- Indents should be used to make code more readable.
- Keywords typically are entered in uppercase; all other words, such as table names and columns, are entered in lowercase.



SQL Statements

SELECT	Data retrieval

INSERT
UPDATE
DELETE
MERGE
	Data manipulation language (DML)

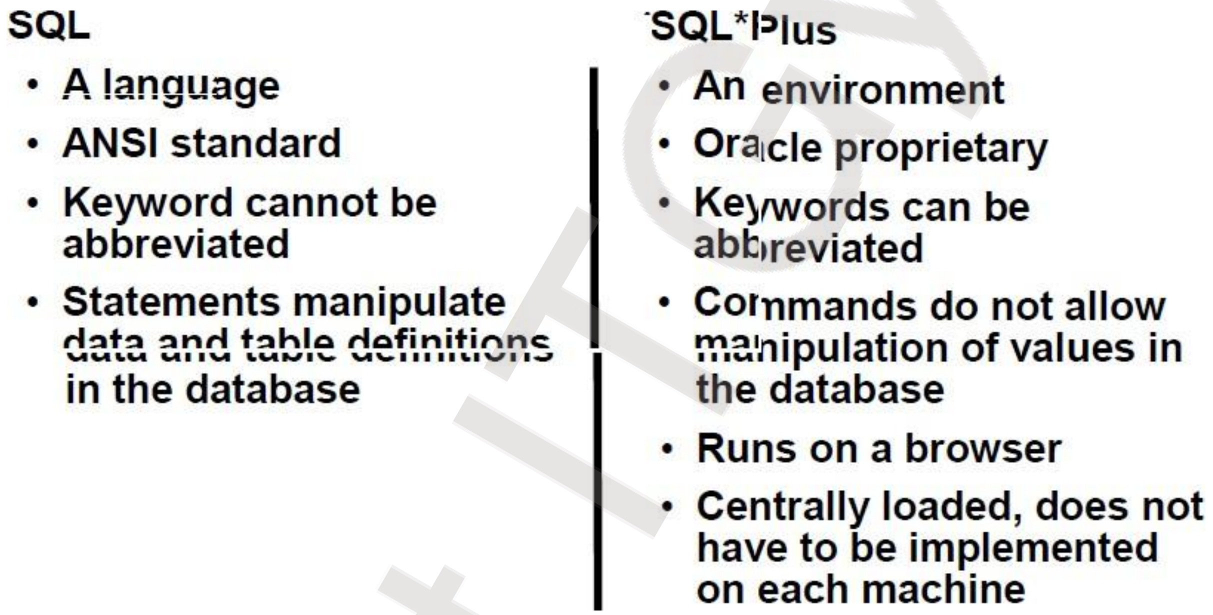
CREATE ALTER DROP RENAME TRUNCATE	Data definition language (DDL)
COMMIT ROLLBACK SAVEPOINT	Transaction control
GRANT REVOKE	Data control language (DCL)

SQL*Plus :-

SQL*Plus is an Oracle tool that recognizes and submits SQL statements to the Oracle server for execution and contains its own command language.
SQL*Plus is an environment in which you can do the following :-

- Execute SQL statements to retrieve, modify, add, and remove data from the database
- Format, perform calculations on, store, and print query results in the form of reports
- Create script files to store SQL statements for repetitive use in the future

SQL vs SQL*PLUS :-

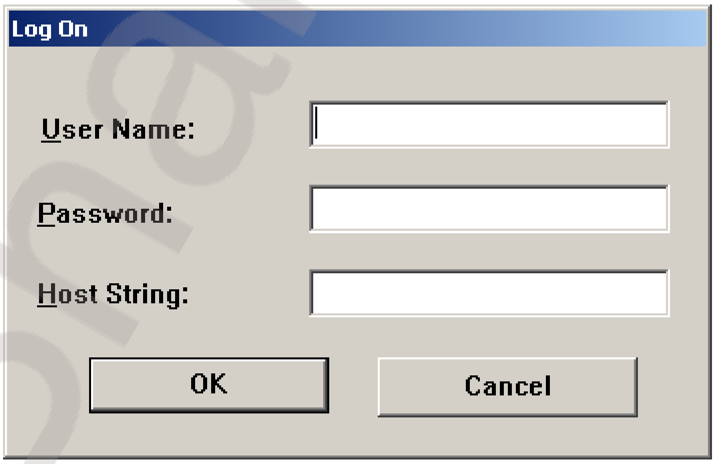


SQL	SQL*Plus
• A language	• An environment
• ANSI standard	• Oracle proprietary
• Keyword cannot be abbreviated	• Keywords can be abbreviated
• Statements manipulate data and table definitions in the database	• Commands do not allow manipulation of values in the database
	• Runs on a browser
	• Centrally loaded, does not have to be implemented on each machine

Schema :- a user in oracle db is called schema and objects created by user are called schema objects

Logging In to sql*plus :-

➔ Open sql*plus
➔ Enter username & password
➔ Enter host string



Datatypes :-

When you create a table or cluster, you must specify a datatype for each of its columns. When you create a procedure or stored function, you must specify a datatype for each of its arguments. These datatypes define the domain of values that each column can contain or each argument can have. For example, DATE columns cannot accept the value February 29 (except for a leap year) or the values 2 or 'SHOE'. Each value subsequently placed in a column assumes the column's datatype. For example, if you insert '01-JAN-98' into a DATE column, Oracle treats the '01-JAN-98' character string as a DATE value after verifying that it translates to a valid date.

CHAR(size) :-

Fixed-length character data of length size bytes.Maximum size is 2000 bytes. Default and minimum size is 1 byte.

VARCHAR2(size) :-

Variable-length character string having maximum length size bytes or characters. Maximum size is 4000 bytes, and minimum is 1 byte or 1 character. You must specify size for VARCHAR2.

NCHAR(size) :-

Fixed-length character data of length size characters or bytes, depending on the choice of national character set. Maximum size is determined by the number of bytes required to store each character, with an upper limit of 2000 bytes. Default and minimum size is 1 character or 1 byte, depending on the character set.

NVARCHAR2(size) :-

Variable-length character string having maximum length size characters or bytes, depending on the choice of national character set. Maximum size is determined by the number of bytes required to store each character, with an upper limit of 4000 bytes. You must specify size for NVARCHAR2.

NUMBER(p,s) :-

Number having precision p and scale s. The precision p can range from 1 to 38. The scale s can range from -84 to 127

LONG :-

Character data of variable length up to 2 gigabytes, or 231 -1 bytes.

DATE :-

Allows date & time but Time is optional if not entered by user then oracle inserts 12:00AM.Valid date range from January 1, 4712 BC to December 31, 9999 AD.a Date field occupies 7 bytes of memory

TIMESTAMP (fractional_seconds_precision) :-

Year, month, and day values of date, as well as hour, minute, and second values of time, where fractional_seconds_precision is the number of digits in the fractional part of the SECOND datetime field. Accepted values of fractional_seconds_precision are 0 to 9. The default is 6.

TIMESTAMP(fractional_seconds_precision) WITH TIME ZONE:-

All values of TIMESTAMP as well as time zone displacement value, where fractional_seconds_precision is the number of digits in the fractional part of the SECOND datetime field.Accepted values are 0 to 9. The default is 6.a The codes listed for the datatypes are used internally by Oracle. The datatype code of a column or object attribute is returned by the DUMP function.

TIMESTAMP(fractional_seconds_precision) WITH LOCAL TIME ZONE:-

All values of TIMESTAMP WITH TIME ZONE, with the following exceptions and Data is normalized to the database time zone When it is stored in the database. When the data is retrieved, users see the data in the session time zone.

INTERVAL YEAR(year_precision) TO MONTH

Stores a period of time in years and months, whereyear_precision is the number of digits in the YEAR datetime field. Accepted values are 0 to 9. The default is 2.

INTERVAL DAY (day_precision) TO SECOND (fractional_seconds_precision)

Stores a period of time in days, hours, minutes, and seconds, where day_precision is the maximum number of digits in the DAY datetime field. Accepted values are 0 to 9. The default is 2.and fractional_seconds_precision is the number of digits in the fractional part of the SECOND field. Accepted values are 0 to 9. The default is 6

RAW(size) :-

Raw binary data of length size bytes. Maximum size is 2000 bytes. You must specify size for a RAW value.

LONG RAW :-

Raw binary data of variable length up to 2 gigabytes.

ROWID :-

Hexadecimal string representing the unique address of a row in its table. This datatype is primarily for values returned by the ROWID pseudocolumn.

UROWID [(size)] :-

Hexadecimal string representing the logical address of a row of an index-organized table. The optional size is the size of a column of type UROWID. The maximum size and default is 4000 bytes.

CLOB :-

A character large object containing single-byte characters. Both fixed-width and variable-width character sets are supported, both using the CHAR database character set. Maximum size is 4 gigabytes.

NCLOB :-

A character large object containing unicode characters. Both fixed-width and variable-width character sets are supported, both using the NCHAR database character set. Maximum size is 4 gigabytes. Stores national character set data.

BLOB :-

A binary large object. Maximum size is 4 gigabytes.

BFILE :-

Contains a locator to a large binary file stored outside the database. Enables byte stream I/O access to external LOBs residing on the database server. Maximum size is 4 gigabytes.

BINARY_FLOAT :-

32-bit single precision floating point number datatype. Binary float equires 5 bytes including a length byte.

BINARY_DOUBLE:-

64-bit double precision floating point number datatype. Binary double requires 9 bytes including a length byte.

OCA question :-

1 Which three statements are true regarding the data types in Oracle Database 10g/11g?

A. Only one LONG column can be used per table.
B. A TIMESTAMP data type column stores only time values with fractional seconds.
C. The BLOB data type column is used to store binary data in an operating system file.
D. The minimum column width that can be specified for a VARCHAR2 data type column is one.
E. The value for a CHAR data type column is blank-padded to the maximum defined column width.

2 You need to create a table for a banking application. One of the columns in the table has the following requirements:

1) You want a column in the table to store the duration of the credit period.
2) The data in the column should be stored in a format such that it can be easily added and subtracted with DATE data type without using conversion functions.
3) The maximum period of the credit provision in the application is 30 days.
4) The interest has to be calculated for the number of days an individual has taken a credit for.

Which data type would you use for such a column in the table?

A. DATE B. NUMBER C. TIMESTAMP
D. INTERVAL DAY TO SECOND E. INTERVAL YEAR TO MONTH

OPERATORS IN ORACLE :-

Operators in ORACLE categorized into following categories

ARTHMETIC OPERATORS :-

+ - * /

Operator precedence:-

- ➔ Operators * , / having higher precedence than operators + , -
- ➔ Operators of the same priority are evaluated from left to right.
- ➔ Use parenthesis to control the precedence.

RELATIONAL OPERATORS :-

Used for comparision , different relational operators supported by oracle

Operator	Description
>	greater than
>=	greater than or equal
<	less than
<=	less than or equals

=	equal
< >	not equal

LOGICAL OPERATORS :-

AND	used to combine two conditions
OR	used to combine two conditions
NOT	negate condition

SPECIAL OPERATORS :-

||
BETWEEN
IN
LIKE
IS
ANY
ALL
EXISTS
PIVOT

Creating table :-

Different types of tables can be created in ORACLE.

- ➔ Standard tables
- ➔ Partitioned tables
- ➔ Clustered tables
- ➔ Index organized tables
- ➔ External tables
- ➔ Global temporary tables

Standard Table :-

Syntax:-

```
SQL> CREATE TABLE  <Table Name>
                   (Colname  datatype(size) ,
                    Colname  datatype(size) ,
                    - ------------------ ) ;
```

Rules for creating a table :-

- tablename should start with alphabet
- tablename should not contain spaces or special symbols , but allows _ , $, #
- tablename should not be a oracle reserved word
- tablename can contain max 30 chars
- a table can contain max of 1000 columns

Example:-

```
SQL> CREATE TABLE emp
        (empno NUMBER(4), ename VARCHAR2(20),
         job     VARCHAR2(10),  hiredate DATE,
        sal     NUMBER(6,2), comm NUMBER(6,2) ,
        deptno  NUMBER(2));
```

OCA Question :-

Which is the valid CREATE TABLE statement?

A. CREATE TABLE emp9$# (emp_no NUMBER (4));
B. CREATE TABLE 9emp$# (emp_no NUMBER(4));
C. CREATE TABLE emp*123 (emp_no NUMBER(4));
D. CREATE TABLE emp9$# (emp_no NUMBER(4), date DATE);

Inserting Data into a Table:-

INSERT command is used to insert record into a table.

Syntax:-

INSERT INTO <table name> VALUES(list of values)

Note :- Strings and Dates must be enclosed in single quotes.

Example :-

```
SQL>INSERT INTO emp  VALUES(1000,'BLAKE','MANAGER',  '10-JAN-10',5000,500,10) ;
```

NOTE:-

Order of values in the INSERT command should match with order of columns declared in table.to insert values in different order then we need to specify the order.

Inserting NULL values :-

→NULL values are inserted when value is

- Absent
- Unknown
- Not Applicable

→NULL is not equal to 0 and not equal to space

→NULL values can be inserted in two ways.

- ➔ EXPLICITLY
- ➔ IMPLICITLY

Inserting NULL values EXPLICITLY:

- ➢ to insert Null values into Numeric columns use NULL keyword.
- ➢ To insert Null values into character & date columns use '' .

Example :-

```
SQL>INSERT INTO  emp   VALUES(1002,'JAMES','',5000,NULL,10);
```

Inserting NULL values IMPLICITLY :-

Example :-

```
SQL>  INSERT INTO emp(EMPNO,ENAME,SAL,DEPTNO)
                         VALUES(1005,'SMITH',2000,10);
```

Remaining columns are automatically filled with NULL values.

Inserting MULTIPLE records :-

The same INSERT command can be executed number of times with different values by using substitution variables. Substitution variables can be declared by using

Single ampersand (&)
Double ampersand (&&)

These variables stores data temporarily

Using Single ampersand :-

These variables are prefixed with &. Values assigned to these variables exists upto the command , once command execution is completed values assigned to these variables are erased.

Example:-

```
SQL>INSERT INTO emp VALUES(&empno,'&ename','&job',
'&hiredate',&sal,&comm,&deptno);
```

Using Double Ampersand :-

These variables are prefixed with &&. Values assigned to these variables even after execution of INSERT command upto the end of session.

Example :-

```
SQL>INSERT INTO emp VALUES (&empno,'&ename','&&job',&&sal,'&&hiredate',&deptno);
```

Inserting data into TIMESTAMP column :-

```
SQL>CREATE TABLE transactions
        (trid    NUMBER(5),
        ttype   CHAR(1),
        ttime   TIMESTAMP,
        tamt    NUMBER(11,2),
        accno   NUMBER(4));
```

```
SQL>INSERT INTO transactions VALUES(1,'W','11-JAN-2012 10:00:00.123456',1001);
```

To insert current timestamp

```
SQL>INSERT INTO transactions VALUES(1,'W',SYSTIMESTAMP,1001);
```

Inserting data into TIMESTAMP WITH TIMZONE column :-

```
SQL>CREATE TABLE transactions
        (trid    NUMBER(5),
        ttype   CHAR(1),
        ttime   TIMESTAMP WITH TIME ZONE ,
        tamt    NUMBER(11,2),
        accno   NUMBER(4));
```

```
SQL>INSERT INTO transactions VALUES(1,'W','11-JAN-2012 10:00:00.123456 AM +5:30',1001);
```

Instead of inserting TZH (time zone hour), we can also insert TZR (time zone region) as follows

```
SQL>INSERT INTO transactions VALUES(1,’W’,’11-JAN-2012 10:00:00.123456 AM US/Pacific’,1001);
```

Inserting data into INTERVAL YEAR TO MONTH column :-

```
SQL>CREATE TABLE course
              (cid            NUMBER(2),
              cname           VARCHAR2(20),
              duration        INTERVAL YEAR TO MONTH);
```

to insert a 3 years course

```
SQL>INSERT INTO course VALUES(1,’MCA’,INTERVAL ‘3’ YEAR);
```

to insert a 1 year 6 months course

```
SQL>INSERT INTO course VALUES(2,’PGDBM’,INTERVAL ‘1-6’ YEAR TO MONTH);
```

to insert 6 months course

```
SQL>INSERT INTO course VALUES(3,’Diploma in Hotel Mgmt’,INTERVAL ‘6’ MONTH);
```

Form of Interval Literal	Interpretation
INTERVAL '123-2' YEAR(3) TO MONTH	An interval of 123 years, 2 months.
INTERVAL '123' YEAR(3)	An interval of 123 years 0 months.
INTERVAL '300' MONTH(3)	An interval of 300 months.
INTERVAL '4' YEAR	indicates 4 years.
INTERVAL '50' MONTH	indicates 50 months or 4 years 2 months.
INTERVAL '123' YEAR	Returns an error, because the default precision is 2

You can add or subtract one INTERVAL YEAR TO MONTH literal to or from another to yield another INTERVAL YEAR TO MONTH literal.

For example:-

INTERVAL '5-3' YEAR TO MONTH + INTERVAL'20' MONTH = INTERVAL '6-11' YEAR TO MONTH

Inserting data into INTERVAL DAY TO SECOND:-

INTERVAL '4 5:12:10.222' DAY TO SECOND(3)	4 days, 5 hours, 12 minutes, 10.222 seconds
INTERVAL '4 5:12' DAY TO MINUTE	4 days, 5 hours and 12 minutes.
INTERVAL '400 5' DAY(3) TO HOUR	400 days 5 hours.
INTERVAL '400' DAY(3)	400 days.
INTERVAL '11:12:10.22' HOUR TO SECOND(7)	11 hours, 12 minutes, and 10.2222222 seconds.
INTERVAL '11:20' HOUR TO MINUTE	11 hours and 20 minutes.
INTERVAL '10' HOUR	10 hours.
INTERVAL '10:22' MINUTE TO SECOND	10 minutes 22 seconds.
INTERVAL '10' MINUTE	10 minutes.
INTERVAL '4' DAY	4 days.
INTERVAL '25' HOUR	25 hours.
INTERVAL '30.12345' SECOND(2,4)	30.1235 seconds.

The fractional second '12345' is rounded to '1235' because the precision is 4.

You can add or subtract one DAY TO SECOND interval literal from another DAY TO SECOND literal. For example.

INTERVAL'20' DAY - INTERVAL'240' HOUR = INTERVAL'10-0' DAY TO SECOND

Virtual Columns :-

Feature introduced in **ORACLE 11g.**

When queried, virtual columns appear to be normal table columns, but their values are derived rather than being stored on disc. The syntax for defining a virtual column is listed below.

```
  column_name [datatype] [GENERATED ALWAYS] AS (expression) [VIRTUAL]
```

If the datatype is omitted, it is determined based on the result of the expression. The GENERATED ALWAYS and VIRTUAL keywords are provided for clarity only.

Example :-

```
SQL>CREATE TABLE employees (
  id         NUMBER,
  first_name  VARCHAR2(10),
  last_name   VARCHAR2(10),
  salary      NUMBER(9,2),
  comm        NUMBER(3),
  hra     NUMBER GENERATED ALWAYS AS (SAL*0.3) VIRTUAL,
  CONSTRAINT employees_pk PRIMARY KEY (id)
          );
```

```
SQL>INSERT INTO employees (id, first_name, last_name, salary, comm)
                      VALUES (1, 'JOHN', 'DOE', 1000, 500) ;
```

Querying the table shows the inserted data plus the derived hra

```
SQL>SELECT * FROM employees;
      ID FIRST_NAME            LAST_NAME     SALARY     COMM       HRA
------------ ---------------------- ---------------------- ---------------------- ---------------------
       1          JOHN             DOE             1000          500        300
```

Note:-

→Indexes defined against virtual columns are equivalent to function-based indexes.
→Virtual columns can be referenced in the WHERE clause of updates and deletes, but they cannot be manipulated by DML.

Limitations :-

→A virtual column can only be of scalar datatype . It can't be a user defined type, LOB or RAW.
→All columns mentioned as part of the virtual column expression should belong to the same table.
→No DMLs are allowed on the virtual columns.
→The virtual column expression can't reference any other virtual column.
→Virtual columns can only be created on ordinary tables. They can't be created on index-organized,

external, object, cluster or temporary tables.

User-Specified Quote Character Assignment :-

SQL statements may contain literal single quotes in them, such as when a possessive form of a noun is used (e.g., 'Robert's Bike'). Prior to Oracle 10g, the literal quotes had to be double quoted to make it clear to the SQL or PL/SQL engine that they were literal (e.g., Robert''s Bike). This can make the code much less readable and can cause errors that be difficult to find.

Oracle 10g introduces a solution to this problem in the form of user-specified quote character assignment. With this new functionality, the ' symbol can be replaced by just about any single- or multibyte delimiter or the character pairs [ ], { }, ( ), or < >.

The delimiter is defined by using the quote operator, q, followed by a quote and then the assigned replacement quote delimiter to be used. Here is an example that uses the bracket pair ([ ]) as quote delimiters:

```
SQL> INSERT INTO record VALUES  (q'[Robert's book is good isn't it?]');
SQL> SELECT * FROM record
       WHERE the_value=q'XRobert's book is good isn't it?X';
 THE_VALUE
-------------------------------------------------
Robert's book is good isn't it?
```

In this example, a record was inserted into the RECORD table. A bracket set was used as the delimiter. Then, the same record was queried, this time using the letter X as the delimiter. In both cases, there are single quotes at the beginning after the q operator, and at the very end after the final delimiter.

Data Retrieval :-

SELECT statement can be used to retrieve data from database.

Capabilities of SELECT Statement:-

Using a **SELECT** statement, you can do the following :

- **Projection** :- You can use the projection capability in SQL to choose the columns in a table that you want .You can choose as few or as many columns of the table as you require.
- **Selection**:- You can use the selection capability in SQL to choose the rows in a table that you . You can use various criteria to restrict the rows that you see.
- **Joining:** You can use the join capability in SQL to bring together data that is stored in different tables by creating a link between them.

Syntax :-

SELECT ***/ {*****column*****|*****expression*** **[*****alias*****],...} FROM** ***table;***

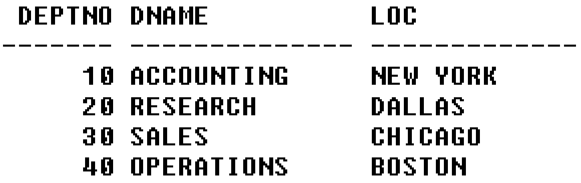
In the syntax :-

- A SELECT clause, which specifies the columns to be displayed

- A FROM clause, which specifies the table containing the columns listed in the SELECT clause

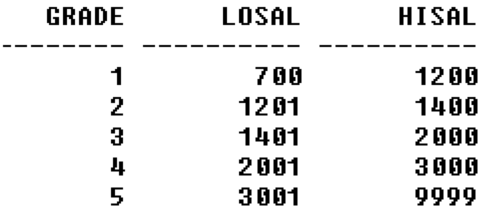
Selecting All Columns :-

SQL>SELECT * FROM dept;



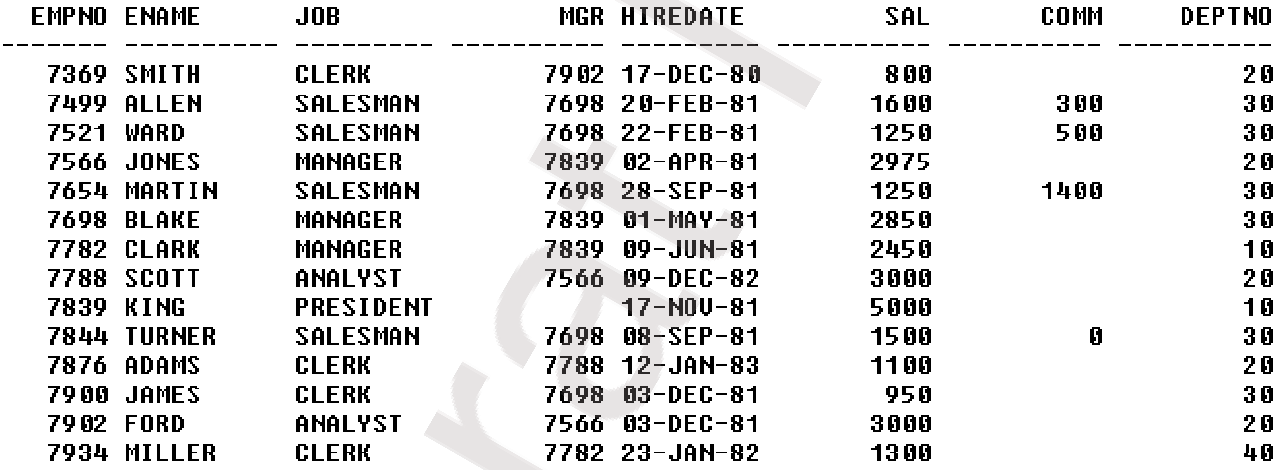
```
 DEPTNO DNAME          LOC
------- -------------- -------------
     10 ACCOUNTING     NEW YORK
     20 RESEARCH       DALLAS
     30 SALES          CHICAGO
     40 OPERATIONS     BOSTON
```

SQL>SELECT * FROM salgrade;



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

SQL>SELECT * FROM emp ;



```
  EMPNO ENAME      JOB              MGR HIREDATE         SAL       COMM     DEPTNO
------- ---------- --------- ---------- --------- ---------- ---------- ----------
   7369 SMITH      CLERK           7902 17-DEC-80        800                    20
   7499 ALLEN      SALESMAN        7698 20-FEB-81       1600        300         30
   7521 WARD       SALESMAN        7698 22-FEB-81       1250        500         30
   7566 JONES      MANAGER         7839 02-APR-81       2975                    20
   7654 MARTIN     SALESMAN        7698 28-SEP-81       1250       1400         30
   7698 BLAKE      MANAGER         7839 01-MAY-81       2850                    30
   7782 CLARK      MANAGER         7839 09-JUN-81       2450                    10
   7788 SCOTT      ANALYST         7566 09-DEC-82       3000                    20
   7839 KING       PRESIDENT            17-NOV-81       5000                    10
   7844 TURNER     SALESMAN        7698 08-SEP-81       1500          0         30
   7876 ADAMS      CLERK           7788 12-JAN-83       1100                    20
   7900 JAMES      CLERK           7698 03-DEC-81        950                    30
   7902 FORD       ANALYST         7566 03-DEC-81       3000                    20
   7934 MILLER     CLERK           7782 23-JAN-82       1300                    40
```

Selecting Specific Columns :-

Display only empno,ename,job,sal from emp table ?

SQL>SELECT empno, ename, job, sal, FROM emp;

NOTE:-

→Date and Character data aligned to LEFT

→Numeric data aligned to RIGHT

Arthmetic Expressions :-

an arithmetic expression contain column names,constant numeric values and athmetic operator.

Example :-

Display ename ,sal, annual salaries ?

SQL>SELECT ename, sal, sal*12 FROM emp;

Operator precedence :-

- Multiplication and division take priority over addition and subtraction.
- Operators of the same priority are evaluated from left to right.
- Parentheses are used to force prioritized evaluation and to clarify statements

Example :-

SQL>SELECT ename, sal, 12*sal+100 FROM emp;

The above example displays the ename, sal, and annual sal of employees. It calculates the annual sal as 12 multiplied by the monthly salary, plus a one-time bonus of 100. Notice that multiplication is performed before addition.

You can override the rules of precedence by using parentheses.

SQL>SELECT ename, sal, 12*(sal+100) FROM emp;

Concatenation Operator:-

This operator concatenates two strings represented by two vertical bars || .

Example :-

SQL>SELECT ename||' working as '||job FROM emp;

SQL>SELECT ename||' joined on '||hiredate FROM emp;

Literals in ORACLE:-

A Literal is a Constant

Types of Literals :-

- String constant
- Numeric constant
- Date constant

NOTE :- String constant and Date constants must be enclosed in ' '.

Example :-

SQL>SELECT ename || ' EARNS '|| sal*12 ||' PER YEAR' FROM emp;

OCA question :-

Evaluate the following query:

SQL>SELECT INTERVAL '300' MONTH,
INTERVAL '54-2' YEAR TO MONTH,
INTERVAL '11:12:10.1234567' HOUR TO SECOND FROM dual;

What is the correct output of the above query?

A. +25-00 , +54-02, +00 11:12:10.123457
B. +00-300, +54-02, +00 11:12:10.123457
C. +25-00 , +00-650, +00 11:12:10.123457
D. +00-300 , +00-650, +00 11:12:10.123457

Declaring Alias:-

An Alias is an another name or alternative name, aliases in Oracle are of two types.

- Column Alias
- Table Alias

Column Alias :-

Alias declared for column is called column alias.

Syntax :-

COLNAME / EXPR [AS] ALIAS

→If alias contains spaces or special characters then alias must be enclosed in " "
→The scope of the alias is upto that query.

Example :-

Display ename,sal,comm and in report display sal as basic and comm as bonus ?

SQL>SELECT ename,sal AS basic,comm AS bonus FROM emp;

Display ename , annual salary ?

SQL>SELECT ename,sal*12 AS "ANNUAL SALARY" FROM emp;

Display ename,sal,hra,da,tax,totsal ?

```
SQL>SELECT ename,sal,sal*0.3 AS hra,sal*0.2 AS da, sal*0.1 AS tax ,
              sal+(sal*0.3)+(sal*0.2)-(sal*0.1) AS totsal    FROM emp;
```

OCA question :-

. You need to produce a report where each customer's credit limit has been incremented by $1000. In the output, the customer's last name should have the heading Name and the incremented credit limit should be labeled New Credit Limit. The column headings should have only the first letter of each word in uppercase .

Which statement would accomplish this requirement?

A. SELECT cust_last_name Name, cust_credit_limit + 1000
"New Credit Limit"
FROM customers;

B. SELECT cust_last_name AS Name, cust_credit_limit + 1000
AS New Credit Limit
FROM customers;

C. SELECT cust_last_name AS "Name", cust_credit_limit + 1000
AS "New Credit Limit"
FROM customers;

D. SELECT INITCAP(cust_last_name) "Name", cust_credit_limit + 1000 INITCAP("NEW CREDIT LIMIT")
FROM customers;

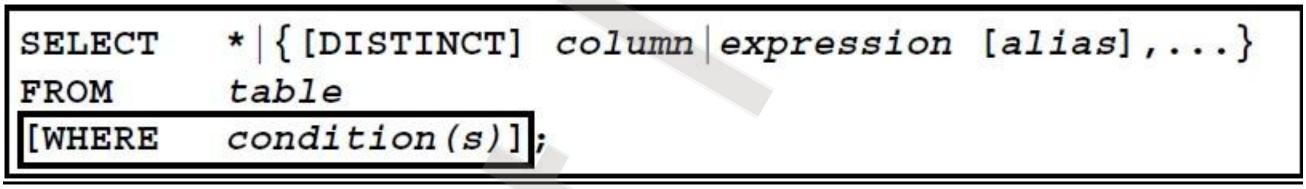
Clauses in ORACLE :-

- WHERE
- ORDER BY
- DISTINCT
- GROUP BY
- HAVING
- ON
- USING
- START WITH
- CONNECT BY
- WITH
- RETURNING
- FOLLOWS
- MODEL

Data Filtering using WHERE clause:-

You can restrict the rows returned from the query by using the WHERE clause. A WHERE clause contains a condition that must be met, and it directly follows the FROM clause. If the condition is true, the row meeting the condition is returned.

syntax:



```
SELECT   *|{[DISTINCT] column|expression [alias],...}
FROM     table
[WHERE   condition(s)];
```

WHERE restricts the rows that meet a condition.

condition is composed of column names, expressions,constants, and a comparison operator

It consists of three elements:

- Column name
- Comparison operator
- Column name, constant, or list of values

Examples :-

Display employee record whose empno=7844 ?

SQL>SELECT * FROM emp WHERE empno=7844 ;

Display employee records whose job='CLERK' ?

SQL>SELECT * FROM emp WHERE job='CLERK' ;

Display employee records working for 10 dept and working as CLERK ?

SQL>SELECT * FROM emp WHERE deptno=10 AND job='CLERK';

Display employee records working as CLERK OR MANAGER ?

SQL>SELECT * FROM emp WHERE job='CLERK' OR job='MANAGER' ;

Display employee records earning between 2000 and 5000 ?

SQL>SELECT * FROM emp WHERE sal>=2000 AND sal<=5000;

Display employee records joined after 1981 ?

SQL>SELECT * FROM emp WHERE hiredate > '31-DEC-1981' ;

Expect the output of the following Query ?

```
SQL>SELECT * FROM emp
    WHERE  job='CLERK' OR  job='MANAGER' AND   sal>2000 ;
```

BETWEEN operator:-

You can display rows based on a range of values using the BETWEEN operator. The range that you specify contains a lowerlimit and an upperlimit . Values specified with the BETWEEN condition are inclusive. You must specify the lower limit first.

Syntax:- **BETWEEN value1 and value2**

Example :-

Display employee records earning between 2000 and 5000 ?

SQL>SELECT * FROM emp WHERE sal BETWEEN 2000 AND 5000;

Note:-

BETWEEN … AND … is actually translated by Oracle server to a pair of AND conditions: (a >=lower limit) AND (a <= higher limit). So using BETWEEN … AND … has no performance benefits, and it is used for logical simplicity.

Example :-

Display employee records who are joined between 1981 year?

```
SQL>SELECT * FROM emp
            WHERE hiredate BETWEEN  '01-JAN-1981' AND '31-DEC-1981' ;
```

Display employee records who are not joined in 2000 year ?

```
SQL>SELECT * FROM emp
        WHERE hiredate NOT BETWEEN    '01-JAN-2000' AND '31-DEC-2000' ;
```

OCA question :-

Expect the output of the following query ?

SQL>SELECT * FROM emp WHERE sal BETWEEN 5000 AND 2000 ;

A error B returns records C returns no rows D none

IN operator :-

To test for values in a specified list of values, use IN operator. The IN operator can be used with any data type. If characters or dates are used in the list, they must be enclosed in single quotation marks ('').

Syntax:-

IN (V1,V2,V3 -----------);

Example :-

Display employee records working as CLERK OR MANAGER ?

SQL>SELECT * FROM emp WHERE job IN ('CLERK','MANAGER') ;

Display employee records not working for dept 10 or 20 ?

SQL>SELECT * FROM emp WHERE deptno NOT IN (10,20)

Note :-

IN (...) is actually translated by Oracle server to a set of OR conditions: a =value1 OR a = value2 OR a =

value3. So using IN (...) has no performance benefits, and it is used for logical simplicity.

LIKE operator:-

You may not always know the exact value to search for. You can select rows that match a character pattern by using the LIKE operator. The character pattern-matching operation is referred as *wildcard* search.

Syntax:-

```
LIKE            'pattern'
NOT LIKE        'pattern'
```

Pattern consists of alphabets,digits and metacharacters. The different meta characters in ORACLE

```
%       denotes zero or many characters.
_       denotes one character.
```

Display employee records whose name starts with S ?

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

Display employee records whose name ends with S ?

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

Display employee records whose name doesn't contain S ?

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

Display employee records where A is the second char in their name ?

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

Display employee records who are joined in JAN month ?

```
SQL>SELECT * FROM emp WHERE hiredate LIKE '%JAN%' ;
```

Display employee records who are joined in 1981 year ?

```
SQL>SELECT * FROM emp WHERE hiredate LIKE '%81' ;
```

Display employee records who are joined in 1st 9 days ?

```
SQL>SELECT * FROM emp WHERE hiredate LIKE '0%' ;
```

Display employee records who are earning 5 digits salary ?

```
SQL>SELECT * FROM emp WHERE sal LIKE '_____' ;
```

Display employee records whose name contains _ ?

```
SQL>SELECT * FROM EMP WHERE ENAME LIKE '%\_%'      ESCAPE '\' ;
```

Expect the output of the following query

```
SQL>SELECT * FROM EMP   WHERE JOB IN ('CLERK','%MAN%');
```

OCA question :-

You need to extract details of those products in the SALES table where the PROD_ID column contains the string '_D123'. ?

Which WHERE clause could be used in the SELECT statement to get the required output?

A. WHERE prod_id LIKE '%_D123%' ESCAPE '_'
B. WHERE prod_id LIKE '%_D123%' ESCAPE '\'
C. WHERE prod_id LIKE '%_D123%' ESCAPE '%_'
D. WHERE prod_id LIKE '%_D123%' ESCAPE '_'

IS operator :

The IS operator tests for nulls. A null value means the value is unavailable, unassigned,unknown, or inapplicable. Therefore, you cannot test with = because a null cannot be equal or unequal to any value.

Syntax :-

IS NULL
IS NOT NULL

Example:-

Display employee records whose comm. Is null ?
SQL>SELECT * FROM emp WHERE comm IS NULL ;
Display employee records whose comm. Is not null ?
SQL>SELECT * FROM emp WHERE comm IS NOT NULL ;

Operator Precedence :-

Order Evaluated	Operator
1	Arithmetic Operator
2	Concatenation Operator
3	Comparison Operator
4	IS [NOT] NULL ,LIKE , [NOT] IN
5	[NOT] BETWEEN
6	NOT logical condition
7	AND logical condition
8	OR logical condition

NOTE:- we can override rules of precedence by using parentheses.

ORDER BY Clause :-

The order of rows returned in a query result is undefined. The ORDER BY clause can be used to sort the rows. If you use the ORDER BY clause, it must be the last clause of the SQL statement. You can specify an expression, or an alias, or column position in ORDER BY clause.

Syntax:-

SELECT *expr* FROM *table*
[WHERE *condition(s)*]
[ORDER BY {*column*, *expr*} [ASC|DESC]];

In the syntax:

ORDER BY	specifies the order in which the retrieved rows are displayed.
ASC	orders the rows in ascending order (default order)
DESC	orders the rows in descending order

Default Ordering of Data :-

- Numeric values are displayed with the lowest values firs for example 1–999.
- Date values are displayed with the earliest value first for example 01-JAN-92 before 01-JAN-95.
- Character values are displayed in alphabetical order—for example, A first and *Z* last.
- Null values are displayed last for ascending sequences and first for descending sequences.

Examples :-

Arrange employee records in ascending order of their sal ?

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

Arrange employee records in descending order of their sal ?

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

Display employee records working for 10th dept and arrange the result in ascending order of their sal ?

```
SQL>SELECT * FROM emp WHERE deptno=10 ORDER BY sal ;
```

Arrange employee records in ascending of their deptno and with in dept arrange records in descending order of their sal ?

```
SQL>SELECT * FROM emp ORDER BY deptno,sal DESC ;
```

In ORDER BY clause we can use column name or column position , for example

```
SQL>SELECT * FROM emp ORDER BY 5 DESC ;
```

In the above example records are sorted based on the fifth column in emp table.

Arrange employee records in descending order of their comm. If comm. Is null then arrange those records last ?

```
SQL>SELECT * FROM emp ORDER BY comm DESC NULLS LAST ;
```

OCA :-

1 Which two statements are true regarding the ORDER BY clause? (Choose two.)

A. It is executed first in the query execution.
B. It must be the last clause in the SELECT statement.
C. It cannot be used in a SELECT statement containing a HAVING clause.
D. You cannot specify a column name followed by an expression in this clause.
E. You can specify a combination of numeric positions and column names in this clause.

2 Which statement is true regarding the default behavior of the ORDER BY clause?

A. In a character sort, the values are case- sensitive.
B. NULL values are not considered at all by the sort operation.
C. Only those columns that are specified in the SELECT list can be used in the ORDER BY clause.
D. Numeric values are displayed from the maximum to the minimum value if they have decimal positions

DISTINCT clause :-

DISTINCT clause is used to eliminate duplicate values.

Syntax :-

```
        DISTINCT collist / *
```

```
SQL>SELECT DISTINCT job FROM emp;
SQL>SELECT DISTINCT deptno,job FROM EMP;
```

OCA question :-

Using the CUSTOMERS table, you need to generate a report that shows 50% of each credit amount in each income level. The report should NOT show any repeated credit amounts in each income level. Which query would give the required result?

A. SELECT cust_income_level, DISTINCT cust_credit_limit * 0.50 AS "50% Credit Limit"
FROM customers;

B. SELECT DISTINCT cust_income_level, DISTINCT cust_credit_limit * 0.50 AS "50% Credit Limit"
FROM customers;

C. SELECT DISTINCT cust_income_level || ' ' || cust_credit_limit * 0.50
AS "50% Credit Limit" FROM customers;

D. SELECT cust_income_level ||' '|| cust_credit_limit * 0.50 AS "50% Credit Limit" FROM customers;

DML commands :-

- ➔ INSERT
- ➔ UPDATE
- ➔ DELETE
- ➔ INSERT ALL
- ➔ MERGE

Copying Data from one table to another table :-

Syntax:-

INSERT INTO <TARGETTABLE>
SELECT <COLLIST> FROM <SOURCE TABLE>

Example :-

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

In the above example first SELECT statement gets data from EMP table and inserts data into EMP_TEMP table and command will be successful only if both tables structure is same.

UPDATE command :-

Update command is used to modify data in a table.

Syntax:-

UPDATE table   SET column = value[, column = value,……]   [WHERE   condition];

Examples :-

Update all employees commission to 500 ?

SQL>UPDATE EMP SET comm=500 ;

Update employee comm to 500 whose comm. Is null ?

SQL>UPDATE EMP SET comm=500 WHERE comm IS NULL ;

Increment employee salary by 10% and comm. By 20% Those who are working as SALESMAN ?

SQL>UPDATE EMP SET sal=sal*1.1 , comm=comm*1.2 WHERE job='SALESMAN' ;

Update different employees comm. With different values ?

SQL>UPDATE EMP SET comm = &comm WHERE empno=&empno;

Update the column value with DEFAULT value ?

SQL>UPDATE EMP SET hiredate=DEFAULT WHERE empno=7844;

Returning Clause:-

→returning clause is used to return values into variables after update .
→To use returning clause declare bind variable (session-level variables)
→Bind variables are declared at SQL prompt , and accessed using : operator.

```
SQL>variable sumsal number ;
SQL>UPDATE emp  SET sal=sal*1.2 Where deptno=10
                                   RETURNING SUM(sal) INTO :sumsal;
_SQL> print :sumsal
```

DELETE command :-

DELETE command is used to delete record or records from a table.

Syntax:-

```
DELETE FROM <TABNAME>   [WHERE <cond> ---- ] ;
```

Delete all employee records ?

```
SQL>DELETE FROM emp ;
```

Delete employee records whose empno=7844 ?

```
SQL>DELETE FROM emp WHERE empno=7844 ;
```

Delete employee records having more than 30 yrs expr ?

```
SQL>DELETE FROM emp WHERE (SYSDATE-hiredate)/365 >= 3 ;
```

Multi Table Insert or INSERT ALL command :-

INSERT ALL command used to insert data into multiple tables.Using INSERT ALL command we can extract data from one or more tables and insert data into multiple tables.

There are two types of INSERT ALL command

1 UNCONDITIONAL INSERT ALL
2 CONDITIONAL INSERT ALL

Unconditional INSERT ALL :-

Syntax:-

```
INSERT ALL
INTO <TAB1> VALUES (VALUE LIST)
INTO <TAB2> VALUES(VALUE LIST)
SELECT STATEMENT ;
```

Example :-

Create table two tables as follows

Emp1(empno,ename,sal)
Emp2(empno,ename,sal)

Copy data from emp to emp1,emp2

```
  SQL>INSERT ALL
           INTO emp1 VALUES(empno,ename,sal)
         INTO  emp2  VALUES(empno,ename,sal)
         SELECT empno,ename,sal FROM emp;
```

scenario:-

Suppose we have sales table with the following structure.
Sales

Prodid	Prodname	Mon_Amt	Tue_Amt	Wed_Amt	Thu_Amt	Fri_Amt	Sat_Amt
101	AIWA	2000	2500	2230	2900	3000	2100
102	AKAI	1900	2100	2130	3100	2800	2120

Now we want to add the rows from SALES table Weekly_Sales Table in the following Structure.

Prodid	Prodname	WeekDay	Amount
101	AIWA	Mon	2000
101	AIWA	Tue	2500
101	AIWA	Wed	2230
101	AIWA	Thu	2900
101	AIWA	Fri	3000
101	AIWA	Sat	2100
102	AKAI	Mon	1900
102	AKAI	Tue	2100
102	AKAI	Wed	2130
102	AKAI	Thu	3100
102	AKAI	Fri	2800
102	AKAI	Sat	2120

To achieve the above we can give a multi table INSERT statement given below

```
SQL>Insert all
    Into week_sales(prodid,prodname,weekday,amount)
    Values (prodid,prodname,’Mon’,mon_amt)
    Into week_sales(prodid,prodname,weekday,amount)
    Values (prodid,prodname,’Tue’,tue_amt)
    Into week_sales(prodid,prodname,weekday,amount)
    Values (prodid,prodname,’Wed’,wed_amt)
    Into week_sales(prodid,prodname,weekday,amount)
    Values (prodid,prodname,’Thu’,thu_amt)
    Into week_sales(prodid,prodname,weekday,amount)
    Values (prodid,prodname,’Fri’,fri_amt)
    Into week_sales(prodid,prodname,weekday,amount)
    Values (prodid,prodname,’Sat’,sat_amt)
   Select prodid,prodname,mon_amt,tue_amt,wed_amt,thu_amt,Fri_amt,sat_amt from sales;
```

Conditional INSERT ALL :-

Syntax :-

```
INSERT ALL
WHEN COND1 THEN
      INTO <TAB1> VALUES (VALUE LIST)
WHEN COND2 THEN
      INTO <TAB2> VALUES  (VALUE LIST)
  SELECT STATEMENT ;
```

Example :-

Copy data from emp to emp1 , emp2 based on the following conditions
If job='CLERK' then copy data to emp1
If job='MANAGER' then copy data to emp2 ?

```
SQL>INSERT ALL
 WHEN job='CLERK' THEN
      INTO emp1 VALUES(empno,ename,sal)
  WHEN job='MANAGER' THEN
       INTO emp2 VALUES(empno,ename,sal)
 SELECT empno,ename,sal FROM emp;
```

MERGE Statement:-

the new MERGE SQL command (sometimes referred to as "UPSERT") is a DML command that enables us to optionally update or insert data into a target table, depending on whether matching records already exist. In versions prior to 9i, we would have to code this scenario either in separate bulk SQL statements or in PL/SQL.

Syntax :-

```
MERGE INTO <TARGETTABLE> <ALIAS>
USING <SOURCE TABLE>/QUERY  <ALIAS>
ON (CONDITION)
WHEN MATCHED THEN
          UPDATE COMMAND
WHEN NOT MATCHED THEN
          INSERT COMMAND
```

records that satisfy the ON condition are updated , if condition is not matched then record is inserted.

Example :-

EMPS			EMPT		
EMPNO	ENAME	SAL	EMPNO	ENAME	SAL
1	A	5000-→UPDATE	1	A	2000
2	B	3000	2	B	3000
3	C	4000	3	C	4000
4	D	6000-→INSERT			

```
SQL>MERGE INTO empt  t
          USING emps  s
          ON (s.empno=t.empno)
          WHEN MATCHED THEN
                    UPDATE SET t.ename = s.ename , t.sal = s.sal
          WHEN NOT MATCHED THEN
                    INSERT VALUES (s.empno , s.ename,s.sal)
```

OCA question :-

You need to load information about new customers from the NEW_CUST table into the tables CUST and CUST_SPECIAL. If a new customer has a credit limit greater than 10,000, then the details have to be inserted into CUST_SPECIAL. All new customer details have to be inserted into the CUST table. Which technique should be used to load the data most efficiently?

A. external table
B. the MERGE command
C the multitable INSERT command
D INSERT using WITH CHECK OPTION

DDL commands :-

- ➔ CREATE
- ➔ ALTER
- ➔ DROP
- ➔ TRUNCATE
- ➔ RENAME

Creating a table from another table:-

Syntax :-

```
CREATE TABLE <TABNAME>
AS   SELECT STATEMENT [WHERE <cond>];
```

Example :-

Create table emp11 from table emp ?

```
SQL>CREATE TABLE emp11
         AS
       SELECT * FROM emp;
```

After executing above command a new table is created called emp11 from the result of SELECT Statement

Copying only structure :-

Create new table emp12 from emp and into the new table copy only structure but do not copy data?

```
SQL>CREATE TABLE emp12 AS
 SELECT * FROM emp WHERE 1=2;
```

Because no record in emp table satisfies condition 1=2 , so no record is copied to EMP12 only the structure is copied.

ALTER command :-

ALTER command is used to modify data definition of a table.ALTER command is used to do following operations.

- ➔ ADD A COLUMN(S)
- ➔ DROP A COLUMN(S)
- ➔ TO RENAME A COLUMN
- ➔ MODIFY A COLUMN
  - INCR/DECR FIELD SIZE
  - CHANGING DATATYPE
  - CHANGING FROM NULL TO NOT NULL
  - CHANGING FROM NOT NULL TO NULL.
- ➔ TO MAKE TABLE READ ONLY

Adding a Column:-

Syntax :-

```
ALTER TABLE <tabname> ADD (colname DATATYPE(SIZE) [ , colname --------])
```

Example:-

```
SQL>ALTER TABLE emp ADD (dob DATE);
```

Droping a Column:-

Syntax :-

```
ALTER TABLE <TABNAME> DROP COLUMN COLNAME ;
```

Example :-

```
SQL>ALTER TABLE emp DROP COLUMN dob;
SQL>ALTER TABLE emp DROP (ename,sal);
```

NOTE :- all columns in a table cannot be dropped , because the table should contain atleast one column.

Renaming a Column :-

Syntax:-

```
ALTER TABLE <tabname> RENAME COLUMN <oldname> to <newname> ;
```

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

Modifying a Column:-

Syntax :-

```
ALTER TABLE <TABNAME>
          MODIFY(COLNAME  DATATYPE(SIZE) , ---------- )
```

Increasing / Decreasing Field Size:-

Increase size of ENAME field to 20 ?

```
SQL> ALTER TABLE emp MODIFY (ename VARCHAR2(20)) ;
```

NOTE :- 1 char field size can be decremented upto max length.
2 to decrement precision or scale of a numeric field , field must be empty.

Changing Datatype:-

```
SQL>ALTER TABLE emp MODIFY (ename CHAR(20)) ;
```

NOTE :-

To change datatype of a column the column should be empty.

Changing Column from NULL to NOT NULL

```
  SQL>ALTER TABLE emp MODIFY (ename  NOT NULL) ;
```

Changing column from NOT NULL to NULL:-

```
  SQL>ALTER TABLE emp  MODIFY(ename NULL) ;
```

Read only Tables :-

From **ORACLE 11g** we can make the table as read only , prior to ORACLE 11g we can do this through view.
A read only table doesn't allow DML operations.

```
SQL>ALTER TABLE emp READ ONLY ;
```

To make table read , write

```
SQL>ALTER TABLE emp READ WRITE ;
```

DROP command :-

DROP command drops a table from database.

Syntax :-

```
 DROP TABLE <TABNAME> ;
```

Example :-

```
SQL>DROP TABLE customer;
```

TRUNCATE command :-

- TRUNCATE command releases memory allocated for a table.
- TRUNCATE deletes all the data from a table.
- TRUNCATE command empties the table.

Syntax :-

```
TRUNCATE TABLE <TABNAME>
```

Example :-

```
SQL>TRUNCATE TABLE EMP ;
```

Difference between DELETE and TRUNCATE :-

DELETE TRUNCATE

DML command
Deletes all or particular records Data can be restored
Deletes row by row Used by developer Triggers can be created

DDL command
deletes only all records Data cannot be restored
doesn't read record before deleting used by DBA
triggers cannot be created

Note :- TRUNCATE is faster than DELETE

RENAME command :-

Used to change name of the table.
Syntax :-

RENAME <OLDNAME> TO <NEWNAME> ;
Example :-

```
SQL>RENAME  emp TO employee;
```

OCA question :-

```
SQL> DROP TABLE products;
```

What is the implication of this command? (Choose all that apply.)

A. All data along with the table structure is deleted.
B. The pending transaction in the session is committed.
C. All indexes on the table will remain but they are invalidated.
D. All views and synonyms will remain but they are invalidated.
E. All data in the table are deleted but the table structure will remain.

Integrity Constraints

Integrity constraints are the rules in real life, which are to be imposed on the data. If the data is not satisfying the constraints then it is considered as inconsistent. These rules are to be enforced on data because of the presence of these rules in real life. These rules are called integrity constraints. Every DBMS software must enforce integrity constraints, otherwise inconsistent data is generated.

You can use constraints to do the following:

- to prevent invalid data entry into tables.
- Enforce rules on the data in a table whenever a row is inserted, updated, or deleted from that table. The constraint must be satisfied for the operation to succeed.
- Prevent the deletion of a record from a table if there are dependencies.

Example for Integrity Constraints :-

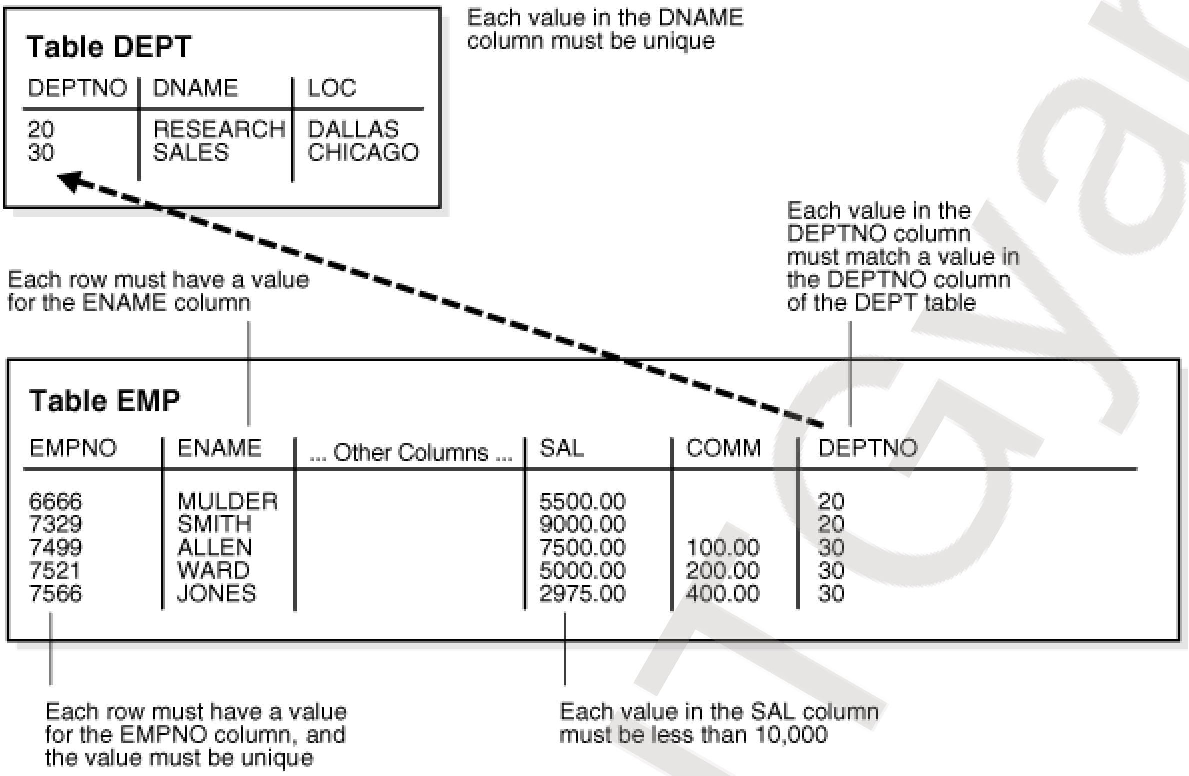


Table DEPT

DEPTNO	DNAME	LOC
20	RESEARCH	DALLAS
30	SALES	CHICAGO

Each value in the DNAME column must be unique

Each value in the DEPTNO column must match a value in the DEPTNO column of the DEPT table

Each row must have a value for the ENAME column

Table EMP

EMPNO	ENAME	... Other Columns ...	SAL	COMM	DEPTNO
6666	MULDER		5500.00		20
7329	SMITH		9000.00		20
7499	ALLEN		7500.00	100.00	30
7521	WARD		5000.00	200.00	30
7566	JONES		2975.00	400.00	30

Each row must have a value for the EMPNO column, and the value must be unique

Each value in the SAL column must be less than 10,000

Types of Integrity Constraints :-

Entity Integrity :-

Entity Integrity constraints are two types

→**Unique Constraint**
→**Primary Constraint**

Refrential Integrity :-

→A refrential integrity constraint states that the values of the foreign key value should match with values of primary key/unique Column of another or same table. Foreign key constraint establishes relationship between tables.

→ The table holding primary key is called parent /master table.

→ The table holding foreign key is called child /detail table.

Self Refrential Integrity :-

If a foreign key in one table refers primary key/unique column of the same table then it is called self refrential Integrity.

Domain constraints:-

A domain means a set of values assigned to a column. Domain constraints are handled by

→defining proper data type
→specifying not null constraint
→specifying check constraint.

Types of Constraints in ORACLE:-

The above said constraints are implemented in oracle with the help of

- →NOT NULL
- →UNIQUE
- →PRIMARY KEY
- →CHECK
- →FOREIGN KEY

The above constraints can be declared at

- →Column level
- → Table level

Column level :-

- → Constraint is declared immediately declaring column.
- → Use column level to declare constraint for single column.

Table level :-

- → use table level to declare constraint for combination of columns.
- → constraint is declared after declaring all columns.

NOT NULL constraint :-

- → It ensures that a table column cannot be left empty.
- → Column declared with NOT NULL is a mandatory column.
- → The NOT NULL constraint can only be applied at column level.

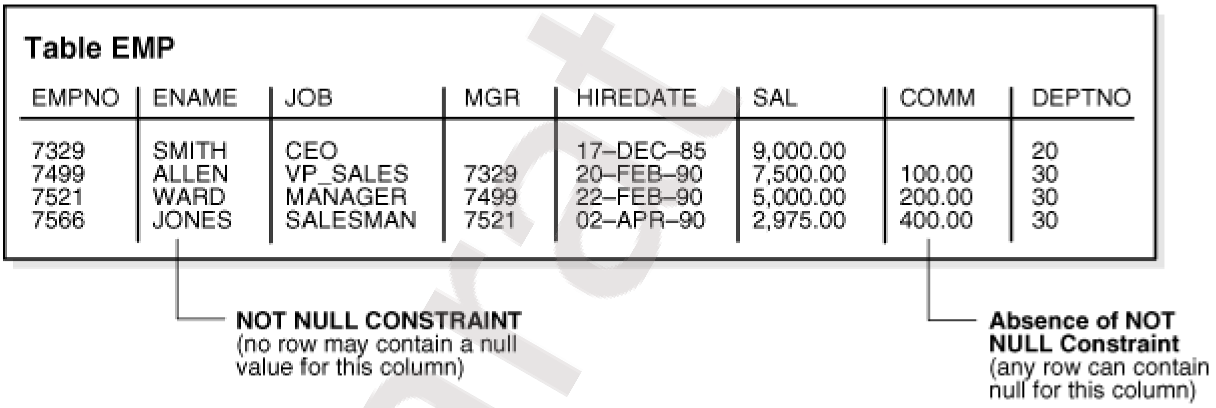


Table EMP

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7329	SMITH	CEO		17–DEC–85	9,000.00		20
7499	ALLEN	VP_SALES	7329	20–FEB–90	7,500.00	100.00	30
7521	WARD	MANAGER	7499	22–FEB–90	5,000.00	200.00	30
7566	JONES	SALESMAN	7521	02–APR–90	2,975.00	400.00	30

NOT NULL CONSTRAINT
(no row may contain a null
value for this column)

**Absence of NOT
NULL Constraint**
(any row can contain
null for this column)

Syntax :-

Columnname Datatype(size) NOT NULL

Example :-

```
SQL> CREATE TABLE emp(
                    Empno      NUMBER(4),
                    Ename   VARCHAR2(20)   NOT   NULL,
                    Job           VARCHAR2(20) ,
                    Mgr           NUMBER(4),
                    Hiredate    DATE,
                    Sal            NUMBER(7,2),
                    Comm        NUMBER(7,2),
                    Deptno     NUMBER(2) );
```

SQL>INSERT INTO emp VALUES(7329,'SMITH','CEO',NULL,'17-DEC-85',9000,NULL,20);
1 row created

SQL>INSERT INTO emp VALUES(7499,'', 'VP_SALES',7329,'20-FEB-90',7,500,100,30);

ERROR ORA-1400 :- cannot insert null into (scott.dept.dname)

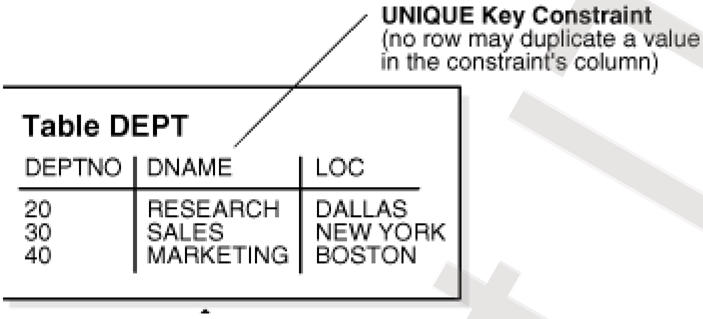
UNIQUE constraint :-

→Columns declared with UNIQUE constraint does not accept duplicate values.
→One table can have a number of unique keys.
→By default UNIQUE columns accept null values unless declared with NOT NULL constraint
→Oracle automatically creates UNIQUE index on the column declared with UNIQUE constraint
→UNIQUE constraint can be declared at column level and table level.

Declaring UNIQUE constraint at Column Level :-

Syntax :-

Columnname Datatype(size) UNIQUE



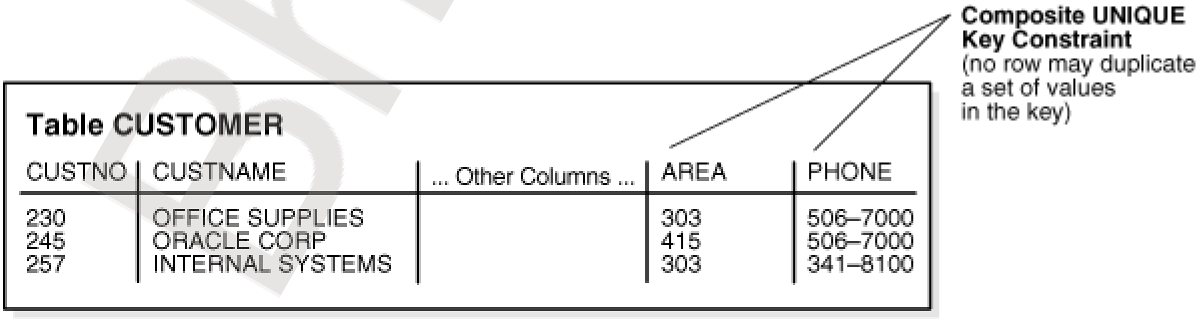
DEPTNO	DNAME	LOC
20	RESEARCH	DALLAS
30	SALES	NEW YORK
40	MARKETING	BOSTON

Example :-

SQL> CREATE TABLE dept
(deptno NUMBER(4)
dname VARCHAR2(20) CONSTRAINT uq_dname_dept UNIQUE ,
loc VARCHAR2(20)) ;

SQL>INSERT INTO dept VALUES(10,'ACCOUNTING','HYDERABAD');
1 row created
SQL>INSERT INTO dept VALUES(20,'ACCOUNTING','MUMBAI');
ERROR ORA-00001 :- unique constraint (uq_dname_dept) violated

Declaring UNIQUE constraint Table Level :-



CUSTNO	CUSTNAME	... Other Columns ...	AREA	PHONE
230	OFFICE SUPPLIES		303	506–7000
245	ORACLE CORP		415	506–7000
257	INTERNAL SYSTEMS		303	341–8100

```
SQL>CREATE TABLE customer(custno        NUMBER(4),
                          custname    VARCHAR2(20),
                          area           NUMBER(3),
                          phone          VARCHAR2(8) ,
                          CONSTRAINT uq_area_ph_cust  UNIQUE(area,phone));
```

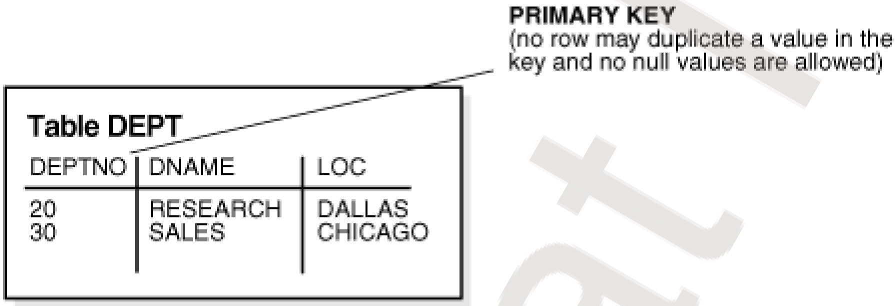
PRIMARY KEY constraint :-

PRIMARY KEY is the candidate key which uniquely identifies a record in a table.

characterstics of PRIMARY KEY :-

- There should be at the most one PK per table.
- PK column do not accept null values.
- PK coumn do not accept duplicate values.
- RAW,LONG RAW,VARRAY,NESTED TABLE,BFILE columns cannot be declared with PK
- If PK is composite then uniqueness is determined by the combination of columns.
- A composite primary key cannot have more than 32 columns
- It is recommended that PK column should be short and numeric.
- Oracle automatically creates Unique Index on PK column

Declaring PRIMARY KEY at Column Level :-



PRIMARY KEY
(no row may duplicate a value in the key and no null values are allowed)

Table DEPT

DEPTNO	DNAME	LOC
20	RESEARCH	DALLAS
30	SALES	CHICAGO

Syntax :-

Colname Datatype(size) PRIMARY KEY

Example :-

```
SQL> CREATE TABLE dept(deptno    NUMBER(4)  CONSTRAINT pk_dept PRIMARY KEY,
                       dname     VARCHAR2(20)  ,
                       loc         VARCHAR2(20) );
```

Declaring PRIMARY KEY at Table Level :-

Example :-

consider the following ORDER_DETAILS table

OrderId	ProdId	Quantity
1000	10	100
1000	11	50
1001	10	20

1001	11	50

In the above example values of **OrderId** are repeated ,so it cannot be taken as primary key. And the values of **ProdId** are also repeated , so it cannot be taken as primary key . when it is not possible with single column to uniquely identify the records then take combination of columns. In the above example combination or **OrdId & ProdId** is not repeated so this combination can be taken as **PRIMARY KEY.**if combination uniquely identifies the records then it is called **composite primary key.**

```
SQL>CREATE TABLE order_details
        (ordid   NUMBER(4)  ,
         prodid  NUMBER(4)  ,
         qty     NUMBER(2) ,
         CONSTRAINT pk_ordid_prodid  PRIMARY KEY(ordid,prodid)) ;
```

CHECK Constraint :-

- Check constraint validates data based on a condition .
- Value entered in the column should not violate the condition.
- Check constraint allows null values.
- Check constraint can be declared at table level or column level.

Limitations :-

- Conditions should not contain pseudo columns like ROWNUM,SYSDATE etc.
- Condition should not access columns of another table

Declaring Check Constraint Column level :-

Syntax :-

```
COLNAME DATATYPE(SIZE)      [CONSTRAINT <NAME>] CHECK(CONDITION)
```

Example :-

```
SQL>CREATE TABLE accounts_master(
                accno NUMBER(4) PRIMARY KEY,
                acname VARCHAR2(20) NOT NULL ,
                balance NUMER(11,2)  CONSTRAINT
                                         ck_bal_accts  CHECK(bal>1000)) ;

SQL>INSERT INTO accounts_master VALUES(1,'A',500);
ERROR  ORA-02293 :- cannot validate  (SCOTT.CK_BAL_ACCTS) check constraint violated
```

Declaring CHECK constraint at Table level :-

Table :- Managers

Mgrno	Mgrname	Start_date	End_date

Rule :- End_date should be greater than Start_date

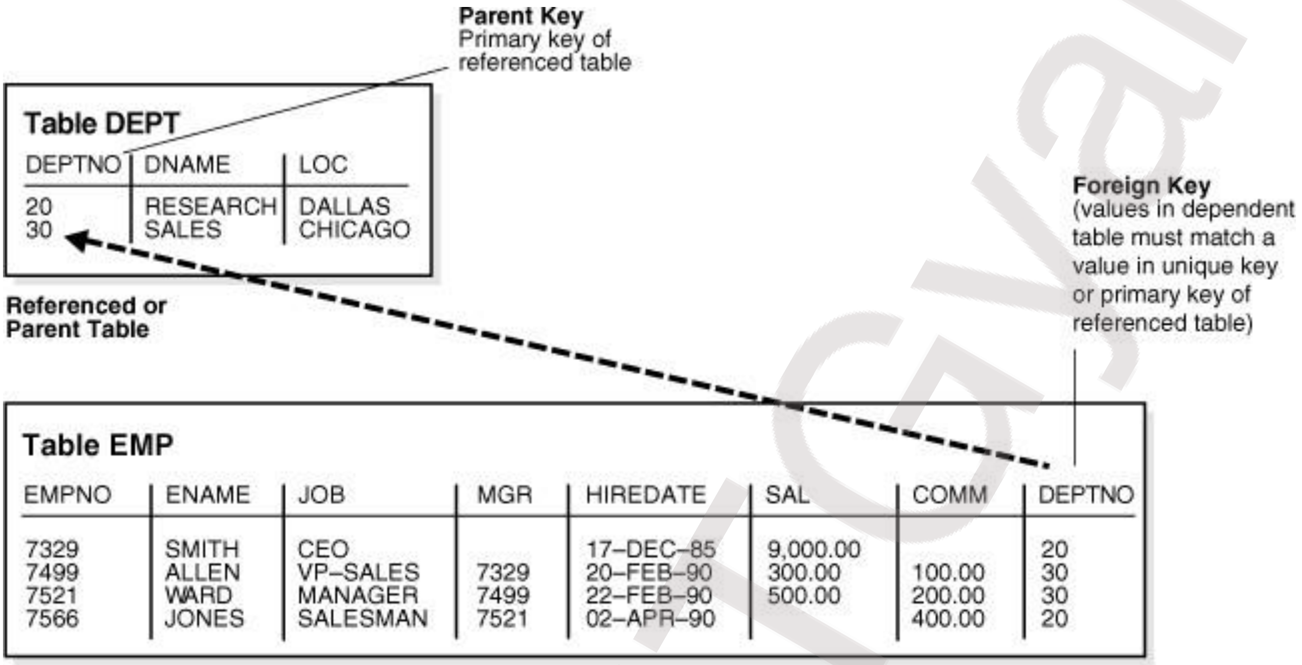
```
SQL>CREATE TABLE managers
             (mgrno        NUMBER(4) PRIMARY KEY,
              mname        VARCHAR2(20) NOT NULL,
             start_date   DATE,
             end_date     DATE ,
            CONSTRAINT ck_mgr  CHECK(end_date > start_date));


SQL>INSERT INTO manager VALUES(1,'A','01-JAN-2011','01-JAN-2010');

ERROR :-  ORA-02290 :- check constraint violated
```

FOREIGN KEY Constraint:-

- Foreign key is used to establish relationship between tables.
- Foreign key is a column in one table that refers primary key/unique columns of another or same table.
- Values of foreign key should match with values of primary key/unique or foreign key can be null.
- Foreign key column allows null values unless it is declared with NOT NULL.
- Foregin key column allows duplicates unless it is declared with UNIQUE
- By default oracle establish 1:M relationship between two tables.
- To establish 1:1 relationship between two tables declare foreign key with unique constraint
- Foreign key can be declared at column level or table level.

- Composite foreign key must refer composite primary key or Composite unique key.

Declaring foreign key at column level :-



Parent Key
Primary key of
referenced table

Table DEPT

DEPTNO	DNAME	LOC
20	RESEARCH	DALLAS
30	SALES	CHICAGO

Referenced or
Parent Table

Foreign Key
(values in dependent
table must match a
value in unique key
or primary key of
referenced table)

Table EMP

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7329	SMITH	CEO		17–DEC–85	9,000.00		20
7499	ALLEN	VP–SALES	7329	20–FEB–90	300.00	100.00	30
7521	WARD	MANAGER	7499	22–FEB–90	500.00	200.00	30
7566	JONES	SALESMAN	7521	02–APR–90		400.00	20

Syntax :-

Colname datatype(size) [constraint <name>] REFERENCES tabname(colname)

Example :-

Creating Parent table :-

```
SQL> CREATE TABLE dept
              (deptno NUMBER(2)    CONSTRAINT pk_dept PRIMARY KEY,
               dname VARCHAR2(20) ,
               loc  VARCHAR2(20)) ;
```

insert records into DEPT table as follows

Deptno	Dname	Loc
10	Accounting	Hyderabad
20	Research	Mumbai

Creating child table :-

```
SQL> CREATE TABLE emp
   (empno      NUMBER(4)      CONSTRAINT pk_emp PRIMARY KEY,
    ename      VARCHAR2(20)  NOT NULL ,
     sal         NUMBER(7,2)    CONSTRAINT ck_sal_emp CHECK(sal>3000),
     deptno     NUMBER(2)       CONSTRAINT fk_deptno_emp    REFERENCES dept(deptno));
```

insert records into EMP table as follows

Empno	Ename	Salary	Deptno	Result
1	Smith	5000	10	**Record is inserted because fk value is matching with pk value**
2	Allen	4000	Null	**Record is inserted because fk allows NULL values**
3	Blake	6000	90	**Oracle returns error because fk value is not matching with pk value**
4	King	7000	10	**Record is inserted because fk allows duplicates**

Declaring Foreign Key constraint at Table Level :-

```
SQL>CREATE TABLE stud_course
                          (sid NUMBER(2) ,
                           cid NUMBER(2) ,
                           doc DATE ,
                           CONSTRAINT pk_stud_course  PRIMARY KEY(sid,cid));

SQL>CREATE TABLE certificates
                           (certno NUMBER(4) PRIMARY KEY,
                            doi  DATE ,
                            sid NUMBER(2),
                            cid NUMBER(2) ,
                            CONSTRAINT fk_sid_cid  FOREIGN KEY(sid,cid)
                                                 REFERENCES  stud_course(sid,cid));
```

DEFAULT Option :-

→If column Declared with DEFAULT option then oracle inserts DEFAULT value when value is not provided.
→DEFAULT option prevents entering NULL values into the column.

Example :-

```
SQL>CREATE TABLE emp
      (empno NUMBER(4) ,
       ename VARCHAR2(20),
        hiredate DATE DEFAULT SYSDATE);

 SQL> INSERT INTO emp(empno,ename)  VALUES(1,'x') ;
```

After executing the above command oracle inserts sysdate into Hiredate column.

Adding constraints to an existing table :-

Constraints can be also be added to an existing table with the help of ALTER command

Syntax :-

```
   ALTER TABLE <TABNAME>  ADD [CONSTRAINT <NAME>]
                               CONSTRAINT_TYPE(COL1 [,COL2])
```

Example :-

Create a table without constraints later add constraints

```
SQL>CREATE TABLE emp55
           (empno NUMBER(4),
            ename VARCHAR2(20),
            sal  NUMBER(7,2),
            dno NUMBER(2));
```

Adding PRIMARY KEY :-

```
SQL>ALTER TABLE emp55
                  ADD CONSTRAINT  pk_emp55  PRIMARY KEY(empno);
```

Note:- primary key constraint cannot be added to a column that already contains duplicates or NULL values.

Adding FOREIGN KEY :-

```
SQL>ALTER TABLE emp55
                ADD CONSTRAINT fk_dno_emp55
                    FOREIGN KEY(dno) REFERENCES  dept(deptno);
```

Adding CHECK constraint :-

```
SQL> ALTER TABLE emp55
         ADD  CONSTRAINT ck_sal_emp55 CHECK(sal>3000)   NOVALIDATE ;
```

NOVALIDATE option :-

If constraint added with NOVALIDATE option then oracle doesn't validate existing data and validates only future DML operations.

Droping Constraints :-

Syntax :-

ALTER TABLE <TABNAME> DROP CONSTRAINT <NAME>

Example :-

```
SQL>ALTER TABLE emp55 DROP CONSTRAINT  pk_emp55;
SQL>ALTER TABLE emp55 DROP CONSTRAINT ck_sal_emp55
```

Note :-

→ PRIMARY KEY cannot be dropped if it referenced by any FOREIGN KEY constraint.
→If PRIMARY KEY is dropped with CASCADE option then along with PRIMARY KEY referencing FOREING KEY is also dropped.
→PRIMARY KEY column cannot be dropped if it is referenced by some FOREIGN KEY.
→PRIMARY KEY table cannot be dropped if it is referenced by some FOREIGN KEY.
→PRIMARY KEY table cannot be truncated if it is referenced by some FOREIGN KEY.

Enabling/Disabling a Constraint:

If the constraints are present, then for each DML operation constraints are checked by executing certain codes internally. It may slow down the DML operation marginally. For massive DML operations, such as transferring data from one table to another because of the presence of constraint, the speed will be considered slower. To improve the speed in such cases, the following methods are adopted:

→Disable constraint
→Performing the DML operation DML operation
→Enable constraint

Disabling Constraint:-

Syntax :-

```
ALTER TABLE <tabname>    DISABLE CONSTRAINT <constraint_name> ;
```

Example :-

```
SQL>ALTER TABLE emp DISABLE CONSTRAINT  ck_sal_emp ;
SQL>ALTER TABLE dept DISABLE PRIMARY KEY CASCADE;
```

NOTE:-

If constraint is disabled with CASCADE then PK is disabled with FK.

Enabling Constraint :-

Syntax :-

```
ALTER TABLE <TABNAME> ENABLE CONSTRAINT <NAME>
```

Example :-

```
SQL>ALTER TABLE emp ENABLE CONSTRAINT ck_sal_emp;
```

Reporting Constraint Exceptions:-

If exceptions exist when a constraint is validated, an error is returned and the integrity constraint remains novalidated. When a statement is not successfully executed because integrity constraint exceptions exist, the statement is rolled back.

To determine which rows violate the integrity constraint, issue the ALTER TABLE statement with the EXCEPTIONS option in the ENABLE clause. The EXCEPTIONS option places the rowid, table owner, table name, and constraint name of all exception rows into a specified table.

Example :-

```
SQL>ALTER TABLE dept ENABLE PRIMARY KEY
                    EXCEPTIONS INTO EXCEPTIONS;
```

If duplicate primary key values exist in the dept table and the name of the PRIMARY KEY constraint on dept is sys_c00610, then the following query will display those exceptions:

```
SQL>SELECT * FROM EXCEPTIONS;
fROWID            OWNER     TABLE_NAME     CONSTRAINT
----------------- --------- ------------- -----------   -----------------

AAAAZ9AABAAABvqAAB  SCOTT     DEPT          SYS_C00610
AAAAZ9AABAAABvqAAG  SCOTT     DEPT          SYS_C00610
```

A more informative query would be to join the rows in an exception report table and the master table to list the actual rows that violate a specific constraint.

```
SQL>SELECT deptno, dname, loc FROM dept, EXCEPTIONS
   WHERE EXCEPTIONS.constraint = 'SYS_C00610'
                            AND
                   dept.rowid = EXCEPTIONS.row_id;
DEPTNO    DNAME          LOC
---------- -------------    -----------

10      ACCOUNTING      NEW YORK
10      RESEARCH        DALLAS
```

Initially Immediate Deferrable:--

Suppose the constraint uses the deferrable clause the enforcement of constraint is deferred, untill commit is issued by the user.

Example :-

```
SQL>Create Table T2  (A number(2), B number(2), c number(2),
CONSTRAINT ck_A CHECK (A>1),
CONSTRAINT ck_B CHECK (B>1),
CONSTRAINT ck_c CHECK (C>1) INITIALLY DEFERRED DEFERRABLE);
```

Suppose insert following data into T2

A	B	C
2	3	4
2	2	0
2	6	0

the clause deferrable causes postponing of the enforcement of constraint checking. Therefore, no error is issued even though second and third row violates the constraint. If you issue commit, the whole transaction is rolled back and you will not get all the rows that are inserted prior to it. If initially deferred is specified then that constraint is checked at the end of the transaction.If the rows that are manipulated by the transaction do not satisfy the constraint, then the whole transaction is rolled back. When initially immediate is specified, which is also the default setting then the constraints are enforced at the end of every DML statement.

When the clause deferrable is used, the behavior of constraint can be changed in the transaction by using the set constraint command.

Enforcing the constraint IMMEDIATE:-

```
SQL>SET CONSTRAINT ck_C IMMEDIATE;
```

Now the user can issue the following command for deferring it:

```
SQL>SET CONSTRAINT ck_c DEFERRED;
```

Changing all constraints to Immediate

```
SQL>SET CONSTRAINT ALL IMMEDIATE;
```

If the constraint is not deferrable, then it cannot be manipulated using the set constraint command. These facilities are provided to tackle various situations in the practical application where data is made according to the constraint at the end of the transaction.

DELETE RULES :-

Delete rules specifies how child record gets affected if parent record is deleted and these rules are declared with foreign key declaration. Oracle supports following Delete Rules :-

→ON DELETE NO ACTION (DEFAULT)
→ON DELETE CASCADE
→ON DELETE SET NULL

ON DELETE NO ACTION :-

If foreign key declared with ON DELETE NO ACTION then parent record cannot be deleted if any child records exists.

ON DELETE CASCADE :-

If foreign key declared with ON DELETE CASCADE then if any parent record is deleted then dependent child records also deleted automatically.

```
SQL>CREATE TABLE dept
          (deptno NUMBER(2)  PRIMARY KEY,
           dname  VARCHAR2(20) NOT NULL ,
           loc  VARCHAR2(20) );

SQL>CREATE TABLE emp (
       empno NUMBER(4) PRIMARY KEY,
       ename VARCHAR2(20) NOT NULL,
       sal NUMBER(7,2) CHECK(sal>3000) ,
       dno NUMBER(2) REFERENCES  dept(deptno) ON DELETE CASCADE);
```

ON DELETE SET NULL :-

if foreign key declared with ON DELETE SET NULL then foreign key value in child table is set to NULL if user deletes record from parent table.

```
SQL>CREATE TABLE dept
          (deptno NUMBER(2)  PRIMARY KEY,
           dname  VARCHAR2(20) NOT NULL ,
           loc  VARCHAR2(20) );



SQL>CREATE TABLE emp (
       empno NUMBER(4) PRIMARY KEY,
       ename VARCHAR2(20) NOT NULL,
       sal NUMBER(7,2) CHECK(sal>3000) ,
       dno NUMBER(2) REFERENCES  dept(deptno) ON DELETE SET NULL);
```

UPDATE RULE :-

Update rules specifies that value of PK cannot be updated if it referenced by any FK.

Retrieving Constraint Information :-

USER_CONSTRAINTS
USER_CONS_COLUMNS
ALL_CONSTRAINTS
DBA_CONSTRAINTS

Example :-

Display list of constraints declared in EMP table ?

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

CONSTRAINT_NAME	CONSTRAINT_TYPE
PK_EMP	P
SYS_C004455	C
CK_SAL_EMP	C
FK_DNO_EMP	R

Oracle gives same code for CHECK and NOT NULL constraint , to know whether constraint is CHECK or NOT NULL use SEARCH_CONDITION as given below.

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

Display which columns are declared with what constraints in EMP table ?

```
SQL>SELECT constraint_name , column_name FROM user_constraints  WHERE table_name='EMP';
```

OCA question :-

1 Which statements are true regarding constraints ?

A a foreign key cannot contain NULL value

B a column with UNIQUE constraint can contain NULL value

C a constraint is enforced only for the INSERT operation on a table

D a constraint can be disabled even if the column contains data

E all constraints can be defined at column level and table level.

2 Evaluate the following CREATE TABLE commands:

```
CREATE TABLE orders
(ord_no NUMBER(2) CONSTRAINT ord_pk PRIMARY KEY,
ord_date DATE,
cust_id NUMBER(4));

CREATE TABLE ord_items
(ord_no NUMBER(2),
item_no NUMBER(3),
qty NUMBER(3) CHECK (qty BETWEEN 100 AND 200),
expiry_date date CHECK (expiry_date > SYSDATE),
CONSTRAINT it_pk PRIMARY KEY (ord_no,item_no),
CONSTRAINT ord_fk FOREIGN KEY(ord_no) REFERENCES orders(ord_no));
```

The above command fails when executed. What could be the reason?

A. SYSDATE cannot be used with the CHECK constraint.

B. The BETWEEN clause cannot be used for the CHECK constraint.

C. The CHECK constraint cannot be placed on columns having the DATE data type.

D. ORD_NO and ITEM_NO cannot be used as a composite primary key because ORD_NO is also the FOREIGN KEY.

3 Which CREATE TABLE statement is valid?

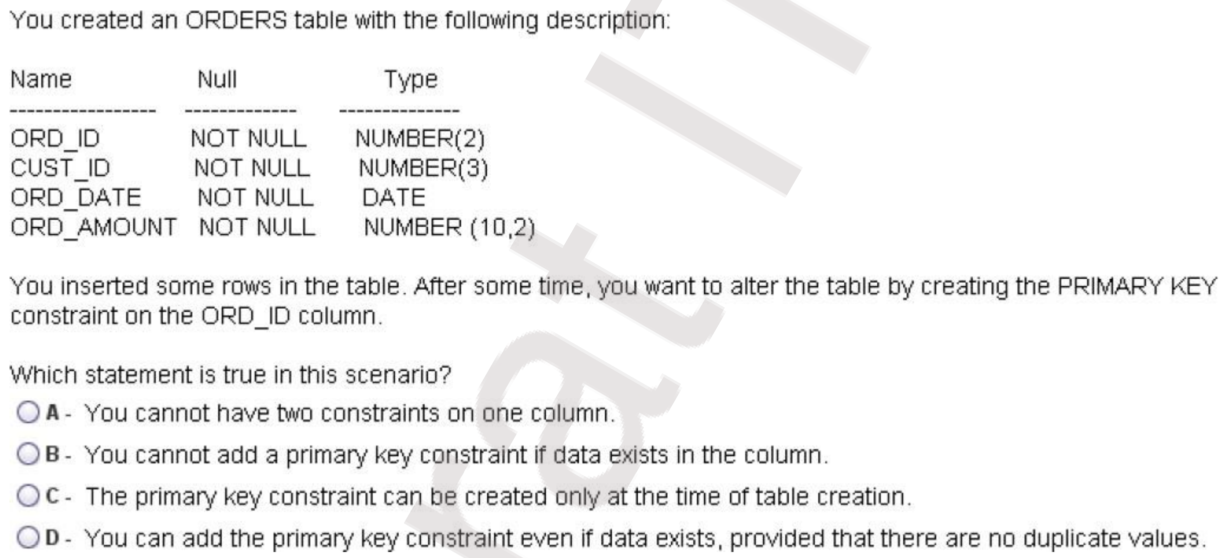
A. CREATE TABLE ord_details
```
(ord_no NUMBER(2) PRIMARY KEY,
item_no NUMBER(3) PRIMARY KEY,
ord_date DATE NOT NULL);
```

```
B. CREATE TABLE ord_details
(ord_no NUMBER(2) UNIQUE, NOT NULL,
item_no NUMBER(3),
ord_date DATE DEFAULT SYSDATE NOT NULL);
```

```
C. CREATE TABLE ord_details
(ord_no NUMBER(2) ,
item_no NUMBER(3),
ord_date DATE DEFAULT NOT NULL,
CONSTRAINT ord_uq UNIQUE (ord_no),
CONSTRAINT ord_pk PRIMARY KEY (ord_no));
```

```
D. CREATE TABLE ord_details
(ord_no NUMBER(2),
item_no NUMBER(3),
ord_date DATE DEFAULT SYSDATE NOT NULL,
CONSTRAINT ord_pk PRIMARY KEY (ord_no, item_no));
```

4



You created an ORDERS table with the following description:

```
Name              Null            Type
-----------------  -------------   --------------
ORD_ID            NOT NULL        NUMBER(2)
CUST_ID           NOT NULL        NUMBER(3)
ORD_DATE          NOT NULL        DATE
ORD_AMOUNT        NOT NULL        NUMBER (10,2)
```

You inserted some rows in the table. After some time, you want to alter the table by creating the PRIMARY KEY constraint on the ORD_ID column.

Which statement is true in this scenario?

○ **A** - You cannot have two constraints on one column.

○ **B** - You cannot add a primary key constraint if data exists in the column.

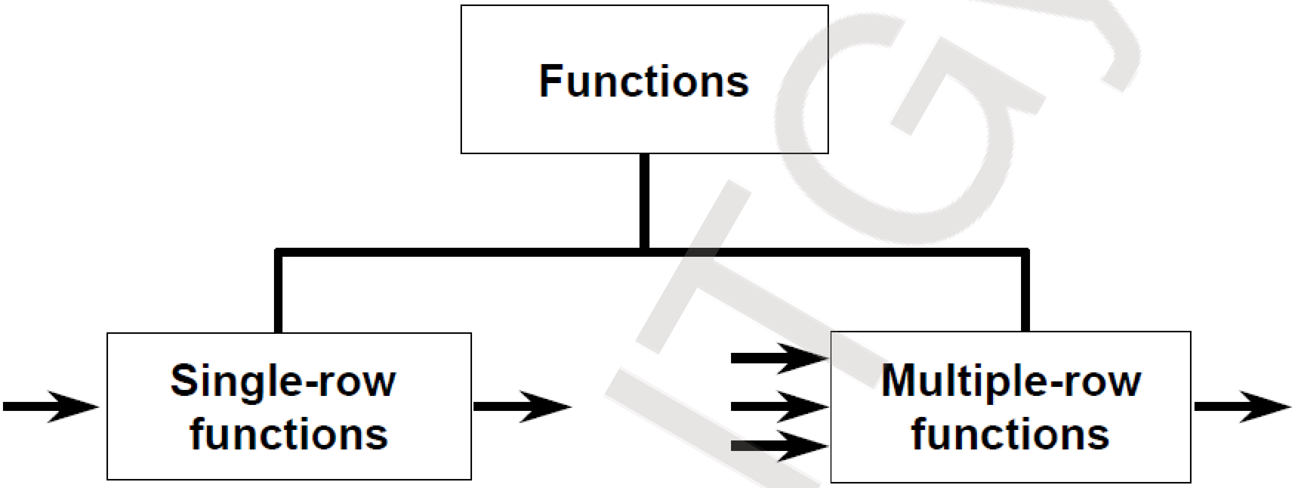
○ **C** - The primary key constraint can be created only at the time of table creation.

○ **D** - You can add the primary key constraint even if data exists, provided that there are no duplicate values.

Built-in Functions

functions are a very powerful feature of SQL and can be used to do the following:-

- Perform calculations on data
- Modify individual data items
- Manipulate output for groups of rows
- Format dates and numbers for display
- Convert column data types



SINGLE ROW FUNCTIONS :-

These functions operate on single rows only and return one result per row. The single row functions are categorized as follows.

- **Character functions**
- **Date functions**
- **Mathematical functions**
- **Conversion functions**
- **Special functions**
- **OLAP functions**

Character functions:-

These functions mainly operate on character data.

UPPER :- converts string to uppercase

Syntax:- UPPER(string)

Example:-

SQL>SELECT UPPER('hello') FROM DUAL;

HELLO

LOWER :- converts string to lower case

Syntax:- LOWER(string)

Example:-

```
SQL>SELECT LOWER('HELLO') FROM DUAL;
hello
```

Display ename,salaries and display names in lower case ?

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

Convert all ename from uppercase to lowercase in table ?

```
SQL>UPDATE emp SET ename=LOWER(ename);
```

INITCAP:- first character is capitalized

Syntax:- **INITCAP(string)**

Example :-

```
SQL>SELECT INITCAP('hello welcome') FROM DUAL ;
Hello Welcome
```

LENGTH :- returns string length

Syntax :- **LENGTH(string)**

Example :-

```
SQL> SELECT LENGTH('hello') FROM DUAL;
5
```

Display employee records whose name contains 5 characters ?

```
SQL>SELECT * FROM emp WHERE LENGTH(ename)=5;
```

SUBSTR:- used to extract part of the string

Syntax:- **SUBSTR(string1,start [, length])**

Example:-

```
SQL>SELECT SUBSTR('hello',2,4) FROM DUAL;
ello
SQL>SELECT SUBSTR('hello welcome',-5,4) FROM DUAL;
lcom
```

Display employee records whose name starts with and ends with same character ?

```
SQL>SELECT * FROM emp WHERE SUBSTR(ename,1,1)=SUBSTR(ename,-1,1);
```

Display employee records whose name starts between 'A' AND 'P' ?

```
SQL>SELECT * FROM emp WHERE SUBSTR(ename,1,1) BETWEEN  'A' AND 'P' ;
```

INSTR :- returns occurrence of one string in another string

Syntax:- **INSTR(str1,str2 [,start , occurrence])**

If str2 exists in str1 returns position
If not exists returns 0.

Example:-

```
SQL> SELECT INSTR('HELLO WELCOME','O') FROM DUAL:
5
SQL> SELECT INSTR('HELLO WELCOME','O',1,2) FROM DUAL:
11
SQL>SELECT INSTR('HELLO WELCOME','O',-1,2) FROM DUAL ;
5
```

Display employee records whose name contains 'S' ?

SQL>SELECT * FROM EMP WHERE INSTR(ENAME,'S') <> 0 ;

Scenario :-

1 CUSTOMER TABLE :-

email

sachin@gmail.com

sourav@gmail.com

from the above email addresses display only the first part ?

SQL>SELECT SUBSTR(EMAIL,1,INSTR(EMAIL,'@')-1) FROM customer;

2 CUSTOMER_TABLE :-

CNAME

Rahuld dravid
Virendra sehwag
Sachin ramesh tendulkar
Sourav ganguly
Mahindra singh dhoni

From the above customer names display only those names that contains 3 parts ?

SQL>SELECT * FROM customer WHERE INSTR(cname,' ',1,2) > 0;

LTRIM :- trims white spaces and unwanted characters on left side

Syntax:- **LTRIM(string1 [, string2])**

Example:-

```
SQL>SELECT LTRIM('             HELLO')  FROM DUAL;
HELLO
SQL>SELECT LTRIM('XXXXXHELLO','X')  FROM DUAL;
HELLO
```

RTRIM :- trims whitespaces and unwanted characters on right side.

Syntax:- **RTRIM(string1 [,string2])**

Example:-

```
SQL>SELECT RTRIM('HELLO       ') FROM DUAL;
HELLO
SQL>SELECT RTRIM('HELLOXXXX','X') FROM DUAL;
HELLO
```

TRIM :- **trims whitespaces and unwanted characters on both left and right side**

```
SQL>SELECT TRIM('     HELLO   ') FROM DUAL;
HELLO
SQL>SELECT TRIM(LEADING 'X' FROM 'XXXXHELLO') FROM DUAL;
HELLO
SQL>SELECT TRIM(TRAILING 'X' FROM 'HELLOXXXXX') FROM DUAL;
HELLO
SQL>SELECT TRIM(BOTH 'X' FROM 'XXXXHELLOXXXX') FROM DUAL;
HELLO
```

LPAD :- one string is filled with another string on left side

Syntax :- **LPAD(string1,length,string2)**

```
SQL>SELECT LPAD(‘hello’,10,’*’) FROM DUAL;
```

```
*****hello
```

RPAD :- fills one string with another string on right side

Syntax:- **RPAD(string1,length,string2)**

Example :-

```
SQL>SELECT RPAD(‘HELLO’,10,’*’)  FROM DUAL;
```

Display ename , salaries and in salary column display **** instead of actual values , for example if salary is 4000 display **** ?

```
SQL> SELECT ename,RPAD(‘*’,sal/1000,’*’) as salary FROM emp ;
```

REPLACE :- to replace one string with another string

Syntax:- **REPLACE(string1,string2,string3)**

Example:-

```
SQL>SELECT REPLACE(‘UTI BANK’,’UTI’,’AXIS’)  FROM DUAL;
```

```
AXIS BANK
```

Display employee records whose name contains exactly one ‘A’ ?

```
SQL> SELECT * FROM emp
          WHERE  LENGTH(ename) – LENGTH(REPLACE(ename,’A’,’’))=1 ;
```

Scenario :-

Examine the data in the ENAME and HIREDATE columns of the EMPLOYEES table:

ENAME	HIREDATE
SMITH	17-DEC-80
ALLEN	20-FEB-81
WARD	22-FEB-81

You want to generate a list of user IDs as follows:

```
USERID
Smi17DEC80
All20FEB81
War22FEB81
```

You issue the following query:

```
SQL>SELECT SUBSTR(INITCAP(ename),1,3) || REPLACE(hiredate,'-',’’) "USERID"  FROM emp;
```

TRANSLATE:- translates one char to another character

Syntax:- **TRANSLATE(string1,string2,string3)**

Example:-

```
SQL> SELECT TRANSLATE(‘HELLO’,’ELL’,’ABC’) FROM DUAL;
```

```
HABBO
```

SQL>SELECT ename, TRANSLATE(sal,'0123456789','$qT*K#PjH@') FROM emp;

CONCAT :- concatenates two strings

Syntax :- CONCAT(str1,str2)

Example :-

```
SQL> SELECT CONCAT('HELLO ', ' WELCOME') FROM DUAL;
HELLO WELCOME
```

SOUNDEX:- A character value representing the sound Of a word, using this we can find strings that sounds same.

Syntax:- SOUNDEX(string)

Example :-

```
SQL>SELECT * FROM EMP
        WHERE   SOUNDEX('SMITH')=SOUNDEX('SMYTH') ;
```

ASCII :- returns ASCII value of first character

Syntax: ASCII(string)

Example :-

```
SQL>SELECT ASCII('A') FROM DUAL ;
65
```

CHR :- returns character for a given ASCII value

Syntax :- CHR(ascii value)

Example :-

```
SQL>SELECT CHR(65) FROM DUAL ;
A
```

Date Functions:-

EXTRACT :- used to extract part of the date.

Syntax:- EXTRACT(FMT FROM DATE)

Extracting year from date:-

```
SQL>SELECT EXTRACT(YEAR FROM SYSDATE) FROM DUAL;
2012
```

Extracting month from date:-

```
SQL>SELECT EXTRACT(MONTH FROM SYSDATE) FROM DUAL;
5
```

Extracting day from date :-

```
SQL> SELECT EXTRACT(DAY FROM SYSDATE) FROM DUAL;
23
```

Display employee records joined in first 15 days in the month APR,DEC in the year between 1980 and 1987 ?

```
 SQL> SELECT * FROM emp
          WHERE  EXTRACT(DAY FROM hiredate) BETWEEN 1 AND 15
                           AND
                    EXTRACT(MONTH FROM hiredate) IN (4,12)
                    AND
                EXTRACT(YEAR FROM hiredate) BETWEEN 1980 AND 1987;
```

ADD_MONTHS:- adds no of months to a date.

Syntax:- ADD_MONTHS(DATE,MONTHS)

Example:-

```
SQL>SELECT ADD_MONTHS(SYSDATE,2)  FROM DUAL;
23-JUN-12
SQL>SELECT ADD_MONTHS(SYSDATE,-2) FROM DUAL;
23-MAR-12
```

Display ename,sal,hiredate and date of retirement , assume that date of retiment is 30 years after date of join ?

```
SQL>SELECT ename,sal,hiredate,ADD_MONTHS(hiredate,30*12) AS DOR FROM emp ;
```

LAST_DAY:- returns last day of the month

Example:-

```
SQL>SELECT LAST_DAY(sysdate) FROM DUAL;

31-MAY-12
```

Display first day of the current month ?

```
SQL>SELECT  ADD_MONTHS(LAST_DAY(SYSDATE)+1,-1) FROM DUAL ?
```

MONTHS_BETWEEN :- returns no of months between two dates.

Syntax:- MONTHS_BETWEEN(date1,date2)

Example:-

```
SQL>SELECT MONTHS_BETWEEN(Sysdate,'20-APR-11')  FROM DUAL

12
```

NEXT_DAY :- returns next specified day starting from given date.

Syntax:- NEXT_DAY(DATE ,DAY)

Example :-

```
SQL>SELECT NEXT_DAY(SYSDATE,'SUNDAY') FROM DUAL;
27-MAY-12
```

Mathematical Functions:-

ABS:- returns absolute value

Syntax:- ABS(number)

Example:-

SQL>SELECT ABS(-10) FROM DUAL;

10

SIGN :-

Syntax :- **SIGN(expr)**

If expr >0 then returns 1
If expr <0 then returns -1
If expr=0 then returns 0

Example :-

SQL>SELECT SIGN(100) FROM DUAL ;
1
POWER:- returns power

Syntax :- **POWER(M,N)**

Example :-

SQL>SELECT POWER(3,2) FROM DUAL;
9
SQRT:- returns square root.

Syntax :- **SQRT(N)**

Example:-

SQL>SELECT SQRT(25) FROM DUAL ;
5
MOD:- returns remainder

Syntax:- **MOD(m,n)**

Example:-

SQL>SELECT MOD(10,2) FROM DUAL;
0
Display employee records earning multiple of 50.
SQL>SELECT * FROM emp WHERE MOD(sal,50)=0;
CEIL:- returns integer greater than or equal to given number.

Syntax:- **CEIL (number)**

Example:-

SQL>SELECT CEIL(9.5) FROM DUAL

10

FLOOR:- returns integer less than or equal to given number.

Syntax:- **FLOOR(number)**

Example:-

SQL>SELECT FLOOR(9.5) FROM DUAL

9

ROUND:- rounds number to given number of decimal places.

Syntax:- ROUND(number [,decimal places])

Example:-

SQL>SELECT ROUND(3.456,2) FROM DUAL ;
3.46

SQL> SELECT ROUND(3.453,2) FROM DUAL;
3.45
SQL>SELECT ROUND(3.456) FROM DUAL ;
3
SQL>SELECT ROUND(3.65) FROM DUAL ;
4
SQL>SELECT ROUND(383.456,-2) FROM DUAL ;
400
SQL>SELECT ROUND(383.456,-1) FROM DUAL
380
SQL>SELECT ROUND(383.456,-3) FROM DUAL ;
0

Note :- ROUND function can also be used to round dates. Date can be rounded to YEAR / MONTH/DAY part.

Assume SYSDATE = 20-apr-2012
SQL>SELECT ROUND(SYSDATE,'YEAR') FROM DUAL;
01-JAN-2012
SQL>SELECT ROUND(SYSDATE,'MONTH') FROM DUAL;
01-may-2012
SQL>SELECT ROUND(SYSDATE,'DAY') FROM DUAL;
22-APR-2012

TRUNC :- truncated the number to specified number of decimal places

Syntax:- TRUN(m,n)

Example :-

SQL>SELECT TRUNC(3.456,2) FROM DUAL ;
3.45
SQL>SELECT TRUN(SYSDATE,'YEAR') FROM DUAL;
01-JAN-2012

Conversion Functions :-

These functions are used to convert from one datatype to another datatype
Conversion of two types :-

→**implicit conversion**
→**explicit conversion**

Implicit Conversion:-

if conversion is performed by ORACLE then it is called implicit conversion.
For assignments, the oracle server can automatically convert the following.

FROM	TO
VARCHAR2	NUMBER
VARCHAR2	DATE
NUMBER	VARCHAR2
DATE	VARCHAR2

For expression evaluation , the oracle server can automatically convert the following .

FROM	TO
VARCHAR2	NUMBER
VARCHAR2	DATE

Example for implicit conversion :-

SQL>SELECT 1000 + '1000' FROM DUAL ;
2000

Explicit Conversion:-

if conversion is performed by user then it is called explicit conversion.The following functions are used to do explicit conversion

1. TO_CHAR
2. TO_DATE
3. TO_NUMBER

TO_CHAR :-

This function is used to convert DATE / NUMBER to CHAR type

Converting DATE to CHAR type :-

DATEs are converted to CHAR type to display DATEs in different format.

Syntax:- **TO_CHAR(DATE [,FORMAT])**

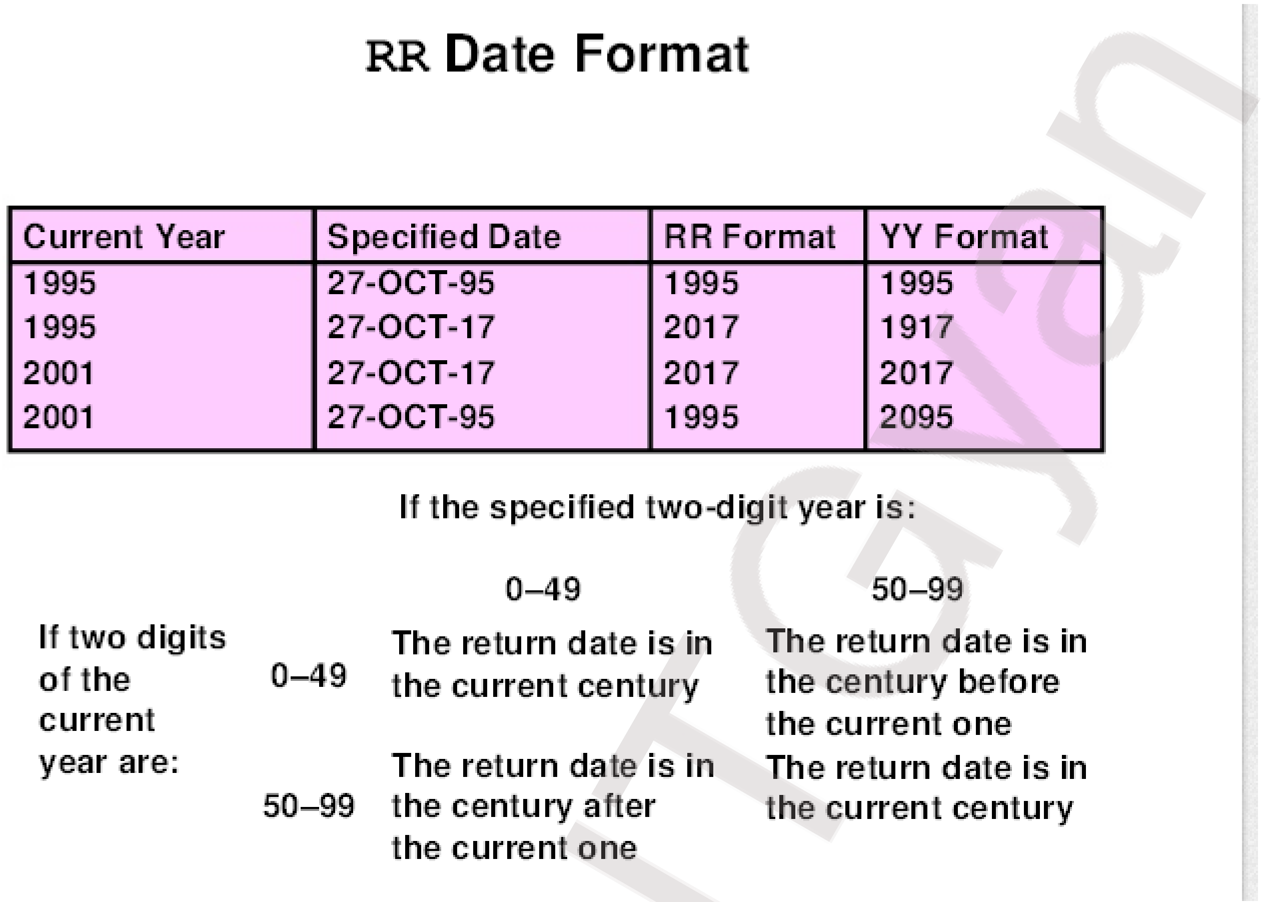
The different formats supported by ORACLE listed below

Century formats :-

CC	Two Digits Century	21
Scc	Two Digits Century with a negative sign for Bc	-10

Year Formats :-

YYYY	All four Digits of the Year	2012
IYYY	All four Digits of the ISO year	2012
SYYYY	All four Digits of the Year with a negative sign for Bc	-1001
YY	Last Two Digits of the Year	12
YEAR	Name of the Year	Two Thousand Twelve



RR Date Format

Current Year	Specified Date	RR Format	YY Format
1995	27-OCT-95	1995	1995
1995	27-OCT-17	2017	1917
2001	27-OCT-17	2017	2017
2001	27-OCT-95	1995	2095

		If the specified two-digit year is:	
		0–49	50–99
If two digits of the current year are:	0–49	The return date is in the current century	The return date is in the century before the current one
	50–99	The return date is in the century after the current one	The return date is in the current century

Example :-

Display employee records joined in 2012 year ?

SQL>SELECT * FROM emp WHERE TO_CHAR(hiredate , 'YYYY') = 2012 ;

SQL>SELECT ename || ' JOINED IN ' || TO_CHAR (hiredate,'YEAR') FROM emp ;

Quarter :-

Q	One Digit Quarter of the Year	2

Example :-

SQL>SELECT TO_CHAR(sysdate , 'Q') FROM emp ;

Month :-

MM	Month Number	1
MON	First Three letter from month	JAN
MONTH	Full name of the month	JANUARY
RM	Roman Numeral month	IV

Example :-

Display employee records joined between JANUARY and APRIL ?

```
SQL>SELECT * FROM emp  WHERE TO_CHAR(hiredate,’mm’) BETWEEN 1 AND 4 ;
```

Day :-

DD	Day of the Month	26
DDD	Day of the Year	103
DAY	Name of the Week Day	SATURDAY
DY	First Three letter from Week Day	SAT
D	Day of the Week	7

Example :-

Display employee records joined on SUNDAY ?

```
SQL>SELECT *  FROM emp
      WHERE TRIM(TO_CHAR(hiredate,’DAY’)) = ‘SUNDAY’;
```

Display on which day employee joined ?

```
SQL>SELECT  ENAME || ‘ joined on ‘ || TO_CHAR(hiredate,’DAY’)  FROM emp;
```

Week :-

WW	week of the year	24
W	week of the month	4

Time :-

HH	hour in 12-format	12
HH24	hour in 24-format	23
MI	minute	20
SS	second	30
AM/PM	AM/PM as appropriate	

Example :-

Display sysdate as follows ?

25 january 2012 , Monday 10:00:00 AM

```
SQL>SELECT  TO_CHAR(SYSDATE,’DD month YYYY , Day  HH:MI:SS PM’)  FROM DUAL;
```

Other Formats :-

AD/BC	AD/BC date as appropriate
TH	th,rd,nd,st
SP	Number is spelled out.
J	Date is displayed in Julian format

Example :-

```
SQL>SELECT TO_CHAR(SYSDATE,’J’) FROM DUAL;
```

2439892

The above number representes number of days passed since 01 JAN 4712BC to SYSDATE.

```
SQL>SELECT TO_CHAR(SYSDATE,’DDSPTH  MON YYYY’) FROM DUAL;
```

To change default DATE format during the session execute following command

```
SQL>ALTER SESSION SET NLS_DATE_FORMAT=’MM/DD/YY’ ;
```

then execute the following command

```
SQL>SELECT ENAME , HIREDATE FROM EMP ;
```

When above query is executed then HIREDATEs are displayed in MM/DD/YY format.

OCA question :-

You need to display the date 11-oct-2007 in words as 'Eleventh of October, Two Thousand Seven'. Which SQL statement would give the required result?

A. SELECT TO_CHAR('11-oct-2007', 'fmDdspth "of" Month, Year') FROM DUAL;

B. SELECT TO_CHAR(TO_DATE('11-oct-2007'), 'fmDdspth of month, year') FROM DUAL;

C. SELECT TO_CHAR(TO_DATE('11-oct-2007'), 'fmDdthsp "of" Month, Year') FROM DUAL;

D. SELECT TO_DATE(TO_CHAR('11-oct-2007','fmDdspth ''of'' Month, Year')) FROM DUAL;

Converting number to character type :-

Syntax :- TO_CHAR(NUMBER [,FORMAT])

Format	Description
S999	Returns Digit with a leading - sign for negative number
0999	returns number with a leading zeros
9900	returns number with trailing zeros.
999.99	returns decimal point in the specified position.
9,999	returns comma in the specified position.
$999	returns a leading Dollar Sign.
C999	returns ISO currency symbol in the specified position
9.99EE	returns number in scientific notation.
RN	returns number in roman format.
L999	retuns number with local currency symbol.

Example :-

```
SQL>SELECT  ename, TO_CHAR(sal,’L9,999’) AS sal  FROM emp ;
```

To set local currency symbol execute the following command.

```
SQL>ALTER SESSION SET NLS_TERRITORY=America;
SQL>ALTER SESSION SET NLS_TERRITORY=Germany;
```

TO_DATE :-

Used to convert string to datetime. You can provide an optional format to indicate the format of string.if you omit format , the date must be in the default format usually (DD-MON-YYYY ,DD-MON-YY).

Syntax:- TO_DATE(string [,format])

Example :-

```
SQL>SELECT  ’26-AUG-2012’ + 10  FROM DUAL ;
```

The above statement returns ORACLE error INVALID NUMBER , because 26-AUG-2012 is treated as string , so to do the calculation conversion is required.

SQL>SELECT TO_DATE('26-AUG-2012') + 10 FROM DUAL ;

SQL>SELECT TO_DATE('08/26/12','MM/DD/YY') + 10 FROM DUAL;

Display on which day india has got independenc ?

SQL>SELECT TO_CHAR(TO_DATE('15-AUG-1947'),'DAY') FROM DUAL;

Display employee names , salaries and display salaries in words ?

SQL>SELECT ename , TO_CHAR(TO_DATE(sal,'J'),'JSP') AS SAL FROM emp;

Example :-

SQL>CREATE TABLE emp (empno NUMBER(4), dob DATE) ;

You need to insert date & time into dob column , but by default DATE datatype accepts only DATE but not time. To insert date along with time conversion is required.

SQL>INSERT INTO emp VALUES (1, TO_DATE('26-AUG-2012 10:20:30','DD-MON-YYYY HH:MI:SS')) ;

But TIMESTAMP datatype allows both date and time without conversion.

SQL>CREATE TABLE emp (empno NUMBER(4), dob TIMESTAMP);

SQL>INSERT INTO emp VALUES(1,'26-AUG-2012 10:20:30') ;

Difference between DATE & TIMESTAMP :-

To insert time into DATE column conversion is required , but to insert time into TIMESTAMP column conversion is not required.

Difference between two DATES returns days , but difference between two TIMESTAMPS returns days,hours,min,sec,milli secs.

TO_NUMBER() :-

Used to convert string to number type.

Syntax :- **TO_NUMBER(string [,format])**

SQL>SELECT TO_NUMBER('$1,000','L9,999') + 1000 FROM DUAL;

2000

Special Functions :-

DECODE Function :-

Decode functions works like if-then-else

Syntax :-

```
DECODE(expr,value1,return expr1,
              Value2, return expr2,
               -------------- ,
              [default expr])
```

- If EXPR is equal to VALUE1 then DECODE returns EXPR1
- If EXPR is equal to VALUE2 then DECODE returns EXPR2
- Returns DEFAULT EXPR if EXPR is not matching with any of the VALUE.
- DEFAULT EXPR is optional , if it is not provided then DECODE returns NULL value.

Example :-

Display ename , job,sal and display job as follows

If job='CLERK' then display 'WORKER'

If job='MANAGER' then display 'BOSS'
If job='PRESIDENT' then display 'BIG BOSS'

```
SQL>SELECT ename,sal,
                DECODE(job,    'CLERK','WORKER',
                                'MANAGER','BOSS',
                                'PRESIDENT','BIG BOSS',
                                'EMPLOYEE') AS JOB  FROM emp;
```

Increment employee salaries as follows ?
If job='CLERK' then increment sal by 10%
If job='SALESMAN' then increment sal by 15%
If job='MANAGER' then increment sal by 20%
Otherwise increment sal by 5%

```
SQL>UPDATE emp SET
                  sal = DECODE( job,    'CLERK',sal*1.1,
                                         'SALESMAN',sal*1.15,
                                        'MANAGER',sal*1.2,
                                        SAL*1.05) ;
```

NOTE :- a decode function can be nested in another decode function

Example :-

If deptno=10 , if job='CLERK' then increment sal by 10%
If job='MANAGER' then increment sal by 15%
Others increment sal by 5%
If deptno=20 , if job='CLERK' then increment sal by 15%
If job='MANAGER' then increment sal by 20%
Others increment sal by 10%
Other departments set to same value ?

```
SQL>UPDATE emp
     SET sal = DECODE(deptno,10,   DECODE(job,'CLERK',SAL*1.1,
                                              'MANAGER',SAL*1.15,
                                              SAL*1.05) ,
                         20,  DECODE(JOB,'CLERK',SAL*1.15,
                                              'MANAGER',SAL*1.2,
                                              SAL*1.05) ,
                              SAL) ;
```

NVL:-

NVL function converts NULL values

Syntax :-

NVL(expr1,expr2)

If expr1 is NULL then NVL function returns expr2 otherwise returns expr1 only.

Example :-

Display employee ename,sal,totsal (sal+comm) ?

SQL>SELECT ename,sal,sal+nvl(comm,0) as totsal FROM emp;

Display ename , sal ,comm. If comm. Is NULL then display "N/A" ?

SQL>SELECT ename , sal , NVL(TO_CHAR(COMM),'N/A') AS comm. FROM emp ;

NVL2 :-

NVL2 function converts NULL values and NOT NULL values.

Syntax :-

```
NVL2(expr1,expr2,expr3)
```

If expr1 is NOT NULL returns expr2 otherwise returns expr3

Example :-

Update employee comm. As follows
If comm. is null then update it to 500 otherwise increment comm. by 200 ?

```
SQL>UPDATE emp SET  comm = NVL2(comm,comm+200,500) ;
```

GREATEST :-

Returns GREATEST number among given numbers

Syntax :- **GREATEST(expr1,expr2,expr4, ----)**

Example :-

```
SQL>SELECT GREATEST(10,20,30)  FROM DUAL ;
SQL>SELECT  GREATEST('A','B','C') FROM  DUAL;
```

LEAST :-

Returns LEAST number among given numbers

Syntax: **LEAST(expr1,expr2,expr3)**

Example:-

```
SQL>SELECT LEAST(10,20,30) FROM DUAL;
```

NULLIF :-

Syntax:- **NULLIF(expr1,expr2)**

Returns NULL if given expressions are equal otherwise returns expr1.

```
SQL>SELECT NULLIF(100,200) FROM DUAL;
```

COALESCE:-

The Oracle COALESCE function returns the first non-NULL expression in the list. If all expressions in the list evaluate to NULL, then the COALESCE function will return NULL. The database evaluates each expression's value and determines whether it is NULL, rather than evaluating all of the expressions before determining if any of them are NULL.

Syntax:-

```
COALESCE( expression_1, expression_2, ... expression_n )
```

The following example uses the sample product_information table to organize a clearance sale of products. It gives a 10% discount to all products with a list price. If there is no list price, then the sale price is the minimum price. If there is no minimum price, then the sale price is "5":

```
SQL>SELECT product_id, list_price, min_price,
   COALESCE(0.9*list_price, min_price, 5) "Sale"
   FROM product_information
   WHERE supplier_id = 102050
   ORDER BY product_id, list_price, min_price, "Sale";
```

OCA question :-

1 Which two statements are true regarding single row functions? (Choose two.)

A. They accept only a single argument.
B. They can be nested only to two levels.
C. Arguments can only be column values or constants.
D. They always return a single result row for every row of a queried table.
E. They can return a data type value different from the one that is referenced.

2 Generate a report showing the total compensation paid to each employee to till date.

```
SQL>SELECT ename ||' joined on ' hiredate || ', the total compensation paid is '||
ROUND(ROUND(SYSDATE-hiredate)/365) *12* sal + NVL(comm,0)) “Until Date “
FROM  emp;
```

3 Which tasks can be performed using SQL functions built into Oracle Database ? (Choose three.)

A. displaying a date in a nondefault format
B. finding the number of characters in an expression
C. substituting a character string in a text expression with a specified string
D. combining more than two columns or expressions into a single column in the output

4 The following data exists in the PRODUCTS table:

PROD_ID	PROD_LIST_PRICE
123456	152525.99

You issue the following query:

```
SQL> SELECT RPAD(( ROUND(prod_list_price)), 10,'*')  FROM products
WHERE prod_id = 123456;
```

What would be the outcome?

A. 152526 ****
B. **152525.99
C. 152525** **
D. an error message

5 Examine the data in the CUST_NAME column of the CUSTOMERS table.

<u>**CUST_NAME**</u>
Renske Ladwig
Jason Mallin
Samuel McCain
Allan MCEwen
Irene Mikkilineni

You need to display customers' second names where the second name starts with "Mc" or "MC."
Which query gives the required output?

```
A. SELECT SUBSTR(cust_name, INSTR(cust_name,' ')+1) FROM customers
          WHERE INITCAP(SUBSTR(cust_name, INSTR(cust_name,' ')+1))='Mc';

B. SELECT SUBSTR(cust_name, INSTR(cust_name,' ')+1)  FROM customers
```

WHERE INITCAP(SUBSTR(cust_name, INSTR(cust_name,' ')+1)) LIKE 'Mc%';

C. SELECT SUBSTR(cust_name, INSTR(cust_name,' ')+1) FROM customers
WHERE SUBSTR(cust_name, INSTR(cust_name,' ')+1) LIKE INITCAP('MC%');

D. SELECT SUBSTR(cust_name, INSTR(cust_name,' ')+1) FROM customers
WHERE INITCAP(SUBSTR(cust_name, INSTR(cust_name,' ')+1)) = INITCAP('MC%');

6 Which SQL statements would display the value 1890.55 as $1,890.55? (Choose three .)

A. SELECT TO_CHAR(1890.55,'$0G000D00') FROM DUAL;
B. SELECT TO_CHAR(1890.55,'$9,999V99') FROM DUAL;
C. SELECT TO_CHAR(1890.55,'$99,999D99') FROM DUAL;
D. SELECT TO_CHAR(1890.55,'$99G999D00') FROM DUAL;
E. SELECT TO_CHAR(1890.55,'$99G999D99') FROM DUAL;

7 In the CUSTOMERS table, the CUST_CITY column contains the value 'Paris' for the CUST_FIRST_NAME 'ABIGAIL'.

Evaluate the following query:

SQL> SELECT INITCAP(cust_first_name)||UPPER(SUBSTR(cust_city,-LENGTH(cust_city),2)))
FROM customers
WHERE cust_first_name = 'ABIGAIL';

What would be the outcome?

A. Abigail PA
B. Abigail Pa
C. Abigail IS
D. an error message
7 Evaluate the following query:

SQL> SELECT TRUNC(ROUND(156.00,-1),-1) FROM DUAL;

What would be the outcome?

A. 16 B. 100 C. 160 D. 200 E. 150

8 Which statements are true regarding data type conversion in expressions used in queries? (Choose all that apply.)

A. inv_amt ='0255982' : requires explicit conversion
B. inv_date > '01-02-2008' : uses implicit conversion
C. CONCAT(inv_amt,inv_date) : requires explicit conversion
D. inv_date = '15-february-2008' : uses implicit conversion
E. inv_no BETWEEN '101' AND '110' : uses implicit conversion

9 You need to calculate the number of days from 1st January 2007 till date. Dates are stored in the default format of dd-mon-rr.

Which SQL statements would give the required output? (Choose two .)

A. SELECT SYSDATE - '01-JAN-2007' FROM DUAL;
B. SELECT SYSDATE - TO_DATE('01/JANUARY/2007') FROM DUAL;
C. SELECT SYSDATE - TO_DATE('01-JANUARY-2007') FROM DUAL;
D. SELECT TO_CHAR(SYSDATE, 'DD-MON-YYYY') - '01-JAN-2007' FROM DUAL;
E. SELECT TO_DATE(SYSDATE, 'DD/MONTH/YYYY') - '01/JANUARY/2007' FROM

Multi-Row functions:-

→These functions will process group of rows and Returns one value from that group.
→These functions are also called AGGREGATE functions or GROUP functions

MAX :-

Returns maximum value of a given expression

Syntax:- MAX(expr)

Example :-

SQL>SELECT MAX(sal) FROM emp;

Display maximum salary of 30th DEPT ?

SQL>SELECT MAX(sal) FROM EMP WHERE deptno=20;

MIN:-

Returns minimum value of a given expression.

Syntax :- MIN(EXPR)

Example:-

SQL>SELECT MIN(sal) FROM emp;

SUM :-

→Returns sum of a given expression.
→This function cannot be applied on strings and dates.

Syntax:-   SUM(expr)

Example:-

SQL>SELECT SUM(sal) FROM emp;

Display total salary paid to MANAGERS ?

SQL>SELECT SUM(sal) FROM emp WHERE job = ‘MANAGER’ ;

Scenario :-

Calculate total salaries paid to each dept as follows ?

DEPT_10	DEPT_20	DEPT_30
?	?	?

```
SQL>SELECT  SUM(DECODE(deptno,10,sal)) as DEPT_10 ,
                    SUM(DECODE(deptno,20,sal)) as DEPT_20,
                   SUM(DECODE(deptno,30,sal)) as DEPT_30
```

```
   FROM emp;
```

AVG :-

Returns avg value of a given expression.

Syntax:- AVG(expr)

Example:-

```
SQL>SELECT AVG(sal) FROM emp;
```

COUNT :-

→Returns no of values present in a column.
→COUNT function ignores NULL values.

Syntax :- COUNT(expr)

Example:-

```
SQL>SELECT COUNT(empno) FROM emp;
SQL>SELECT COUNT(DISTINCT deptno) FROM emp;
```

COUNT(*):-

Returns no of records

Example :-

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

Display number of employees joined in 1981 year ?

```
SQL>SELECT COUNT(*) FROM emp WHER TO_CHAR(hiredate,'yyyy')=1981;
```

Display number of employees joined as follows ?

```
1981    1982    1983
?       ?       ?
```

```
SQL>SELECT  COUNT(DECODE(TO_CHAR(hiredate,'YYYY'),1981,empno)) AS Y1981 ,
            COUNT(DECODE(TO_CHAR(hiredate,'YYYY'),1982,empno)) AS Y1982,
            COUNT(DECODE(TO_CHAR(hiredate,'YYYY'),1983,empno)) AS Y1983  FROM emp;
```

OCA question :-

Which two statements are true regarding the COUNT function? (Choose two.)

A. The COUNT function can be used only for CHAR, VARCHAR2, and NUMBER data types.

B. COUNT(*) returns the number of rows including duplicate rows and rows containing NULL value in any of the columns.

C. COUNT(cust_id) returns the number of rows including rows with duplicate customer IDs and NULL value in the CUST_ID column

D. COUNT(DISTINCT inv_amt)returns the number of rows excluding rows containing duplicates and NULL values in the INV_AMT column.

E. A SELECT statement using the COUNT function with a DISTINCT keyword cannot have a WHERE clause.

CASE Statement :-

→The CASE expression performs if-then –else logic .
→introduced in ORACLE 9i
→The CASE expression works in a similar manner to DECODE, but use CASE because it is ANSI-compliant .
→the CASE expression is easier to read.
There are two types of CASE Statements

→**Simple case.**
→**Searched case** .

Simple CASE Statement :-

Simple CASE expressions use expressions to determine the value to return.

Syntax :-

```
CASE search_expression
WHEN expression1 THEN result1
WHEN expression2 THEN result2
……………………
WHEN expression THEN result
ELSE default_result
END ;
```

→Search_expression is the expression to be evaluated.
→expression1, expression2, …………,expression are the expressions to be evaluated against search_expression.
→result1, result2,…………. , result are the returned results(one for each possible expression). If expression1 evaluates to search_expression, results is returned, and similarly for the other expressions.
→default_result is returned when no matching expression is found.

Example :-

```
SQL>SELECT ename,sal,        CASE job
                               WHEN 'CLERK' THEN 'WORKER'
                             WHEN 'MANAGER' THEN 'BOSS'
                              WHEN 'PRESIDENT' THEN 'BIG BOSS'
                              ELSE
                                   'EMPLOYEE'
                              END  AS JOB
        FROM emp ;
```

Searched CASE Statement :

Searched CASE expressions use conditions to determine the returned value.

```
Syntax :- CASE
WHEN condition1 THEN result1 WHEN condition2 THEN result2
……………….
WHEN condition THEN result

END;
ELSE
default_result
```

Where,
→condition1, condition2,.........conditionN are expressions to be evaluated.
→result1, result2,...........resultN are the returned results(one for each possible condition). If condition is true, result1 is returned, and similarly for the other expressions.
→default_result is returned when there is no condition returns true

Example :-

```
SQL>SELECT ename,sal,        CASE
                               WHEN sal>3000 THEN 'HISAL'
                              WHEN sal<3000 THEN 'LOSAL'
                              ELSE
                                  'MODERATE SAL'
                              END AS SALRANGE
FROM emp ;
```

OCA question :-

1 Examine the data in the PROMO_BEGIN_DATE column of the PROMOTIONS table:

PROMO_BEGIN _DATE
04-jan-00
10-jan-00
15-dec-99
18-oct-98

You want to display the number of promotions started in 1999 and 2000.

Which query gives the correct output?

```
A. SELECT       SUM(DECODE(SUBSTR(promo_begin_date,8),'00',1,0)) "2000",
               SUM(DECODE(SUBSTR (promo_begin_date,8),'99',1,0)) "1999"
   FROM promotions;

B. SELECT SUM(CASE TO_CHAR(promo_begin_date,'yyyy')
                  WHEN '99' THEN 1 ELSE 0 END) "1999",
            SUM(CASE TO_CHAR(promo_begin_date,'yyyy')
                  WHEN '00' THEN 1 ELSE 0 END) "2000"
  FROM promotions;

 C  SELECT COUNT(CASE TO_CHAR(promo_begin_date,'yyyy')
                    WHEN '99' THEN 1 ELSE 0 END)  "1999",
          COUNT(CASE TO_CHAR(promo_begin_date,'yyyy')
                    WHEN '00' THEN 1 ELSE 0 END) "2000"
  FROM promotions;

D   SELECT
        COUNT(DECODE(SUBSTR(TO_CHAR(promo_begin_date,'yyyy'), 8), '1999', 1, 0)) "1999",
        COUNT(DECODE(SUBSTR(TO_CHAR(promo_begin_date,'yyyy'), 8),'2000', 1,0)) "2000"
   FROM promotions;
```

GROUP BY clause:-

You can use GROUP BY clause to divide the rows in a table into smaller groups. You can then use the group functions to return summary information for each group.

Syntax :-

SELECT *column*, *group_function(column)*
FROM *table*
[WHERE *condition*]
[GROUP BY *group_by_expression*]
[HAVING condition] [ORDER BY *column*];

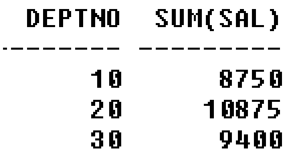
Guidelines :-

- only GROUP BY columns and AGGREGATE functions should appear in SELECT list other than these two if any column appears then oracle returns error.
- Using WHERE clause, you can exclude rows before dividing them into groups.
- You cannot use a column alias in the GROUP BY clause.
- By default, rows are sorted by ascending order of the columns included in the GROUP BY list. You can override this by using the ORDER BY clause.

Examples :-

Display total salaries paid to each department ?

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



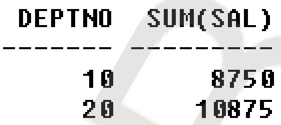
```
  DEPTNO   SUM(SAL)
-------- ---------
      10       8750
      20      10875
      30       9400
```

Display no of employees joined each year ?

```
SQL>SELECT
        EXTRACT(YEAR FROM hiredate) AS YEAR,  COUNT(*) AS EMPS
        FROM emp
        GROUP BY EXTRACT(YEAR FROM hiredate);
```

Display total salaries paid to each department where deptno in (10,20) ?

```
SQL>SELECT deptno,SUM(sal) FROM emp
      WHERE deptno IN (10,20)
        GROUP BY deptno ;
```



```
 DEPTNO   SUM(SAL)
------- ---------
     10       8750
     20      10875
```

HAVING clause :-

In the same way that you use the WHERE clause to restrict the rows that you select, you can use the HAVING clause to restrict groups.

find the maximum salary of each department, but show only the depts. that have a maximum salary more than 10,000, you need to do the following:

1 Find the maximum salary for each department by grouping by deptno

2. Restrict the groups to those departments with a maximum salary greater than 10,000.

The Oracle server performs the following steps when you use the HAVING clause:

1. Rows are grouped.
2. The group function is applied to the group.
3. The groups that match the criteria in the HAVING clause are displayed.

Example :-

```
SQL>SELECT deptno, SUM(sal)
 FROM emp
 GROUP BY deptno
HAVING SUM(sal)>10000;
```

```
Deptno          SUM(SAL)
20              10875
```

WHERE Vs HAVING :-

WHERE	HAVING
Filter rows	filter groups
Filter data before group by	filter data after group by

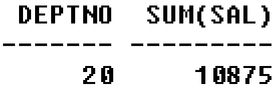
NOTE:- in condition if there is no group function then use WHERE clause , if condition contains group function use HAVING clause.

Using WHERE , GROUP BY ,HAVING clauses Together :-

You can use WHERE,GROUP BY, and HAVING clauses together in the same query. When you do this the WHERE clause first filters the rows, the GROUP BY clause then groups the remaining rows and finally HAVING clause filters the groups.

Example :-

```
SQL>SELECT   deptno,sum(sal)  FROM emp
        WHERE deptno IN (10,20)
         GROUP BY deptno
         HAVING SUM(sal) > 10000 ;
```



```
 DEPTNO   SUM(SAL)
------- ----------
     20      10875
```

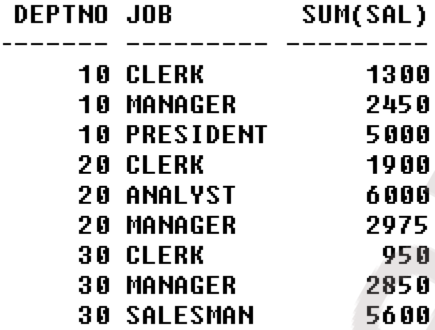
Grouping Rows Based on more than one Column :-

You can GROUP rows based on more than one column.

Calcuate total salaries department wise and within department job wise ?

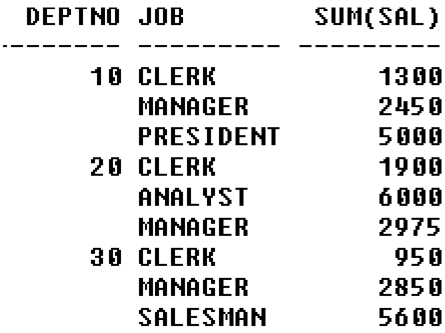
Example :-

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



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

```
SQL>BREAK ON deptno
SQL> /
```



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

```
SQL>SELECT  TO_CHAR(hiredate,’YYYY’) AS Year ,
                        TO_CHAR(hiredate,’Mon’) AS Month,
                      TO_CHAR(hiredate,’Dy’) AS Day , COUNT(*) AS Emps
     FROM emp
     GROUP BY TO_CHAR(hiredate,’YYYY’)  ,
                        TO_CHAR(hiredate,’Mon’) ,
                      TO_CHAR(hiredate,’Dy’)
    ORDER BY Year,Month,Day ;
```

Cross Tabulation:-

An example of cross tabulation shown below :-

DEPTNO	CLERK	MANAGER	SALESMAN
10	1300	2450	
20	1900	2975	
30	95	2850	5600

To produce the above result the following query should be run

```
SQL>SELECT    deptno,          SUM( DECODE(job,’CLERK’,sal)) AS CLERK ,
                                SUM(DECODE(job,’MANAGER’,sal)) AS MANAGER,
                            SUM(DECODE(job,’SALESMAN’,sal)) AS SALESMAN
          FROM emp
          GROUP BY deptno;
```

Using PIVOT operator :-

Cross tabulation is simplified in ORACLE 11g with the help of PIVOT operator.

```
SQL>SELECT * FROM
                          (SELECT DEPTNO,SAL,JOB FROM EMP)
                        PIVOT
                         (
                          SUM(SAL) FOR JOB IN (‘CLERK’,’MANAGER’,’SALESMAN’)
                          )
                        ORDER BY DEPTNO;
```

UNPIVOT operator :-

The UNPIVOT operator converts column-based data into separate rows. To see the UNPIVOT operator in action we need to create a test table.

```
SQL>CREATE TABLE unpivot_test (
                              id           NUMBER,
                              customer_id    NUMBER,
                              product_code_a  NUMBER,
                              product_code_b  NUMBER,
                              product_code_c  NUMBER,
                              product_code_d  NUMBER);

SQL>INSERT INTO unpivot_test VALUES (1, 101, 10, 20, 30, NULL);
SQL>INSERT INTO unpivot_test VALUES (2, 102, 40, NULL, 50, NULL);
SQL>INSERT INTO unpivot_test VALUES (3, 103, 60, 70, 80, 90);
SQL>INSERT INTO unpivot_test VALUES (4, 104, 100, NULL, NULL, NULL);
SQL>COMMIT;
```

So our test data starts off looking like this.

```
SQL>SELECT * FROM unpivot_test;
  ID    CUSTOMER_ID PRODUCT_CODE_A PRODUCT_CODE_B PRODUCT_CODE_C PRODUCT_CODE_D

     1       101                 10                 20                 30
     2       102                 40                 50
     3       103                 60                 70                 80                 90
     4       104                 100
```

The UNPIVOT operator converts this column-based data into individual rows.

```
SQL>SELECT *
FROM   unpivot_test
UNPIVOT  (quantity  FOR  product_code  IN  (product_code_a  AS  'A',  product_code_b  AS  'B',
product_code_c AS 'C', product_code_d AS 'D'));

   ID CUSTOMER_ID   P   QUANTITY
---------- -----------       - ----------
       1       101        A      10
       1       101        B      20
       1       101        C      30
       2       102        A      40
       2       102        C      50
       3       103        A      60
       3       103        B      70
       3       103        C      80
       3       103        D      90
       4       104        A     100
```

Convert rows to columns :-

```
SQL> desc t1
 Name            Null?           Type

 NAME                            VARCHAR2(10)
 YEAR                            NUMBER(4)
 VALUE                           NUMBER(4)

 SQL> select * from t1;
NAME            YEAR            VALUE

john            1991            1000
john            1992            2000
john            1993            3000
jack            1991            1500
jack            1992            1200
jack            1993            1340
mary            1991            1250
mary            1992            2323
mary            1993            8700
```

perform a sql query to return results like this:

```
year, john, Jack, mary
1991, 1000, 1500 1250
1992, 2000, 1200, 2323
1993, 3000, 1340, 8700
```

```
SQL> SELECT year,
  MAX( DECODE( name, 'john', value, null ) ) "JOHN",
  MAX( DECODE( name, 'jack', value, null ) ) "JACK",
  MAX( DECODE( name, 'mary', value, null ) ) "MARY"
  FROM abc
  GROUP BY year;
```

Extending Group By using ROLLUP & CUBE :-

GROUP BY clause can be extended by using two operators.

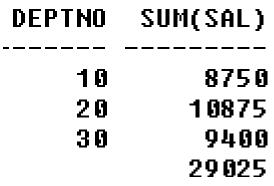
- ROLLUP
- CUBE

ROLLUP :-

ROLLUP returns a row containing a subtotal for each group ,plus a row containing a grand total for all the groups.

Passing single column to ROLLUP :-

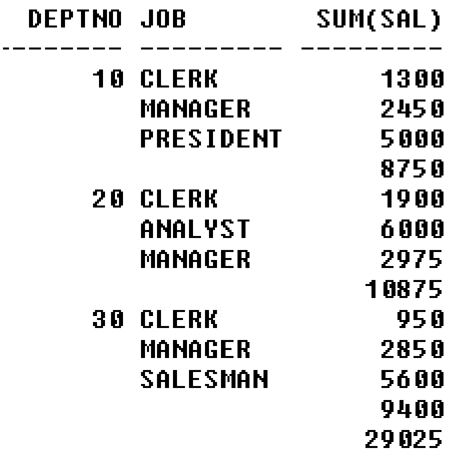
```
SQL>SELECT deptno,SUM(sal) FROM emp GROUP BY  ROLLUP(deptno) ;
```



```
 DEPTNO   SUM(SAL)
------- ---------
     10      8750
     20     10875
     30      9400
            29025
```

Passing multiple columns to ROLLUP :-

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



```
  DEPTNO JOB         SUM(SAL)
-------- --------- ---------
      10 CLERK          1300
         MANAGER        2450
         PRESIDENT      5000
                        8750
      20 CLERK          1900
         ANALYST        6000
         MANAGER        2975
                       10875
      30 CLERK           950
         MANAGER        2850
         SALESMAN       5600
                        9400
                       29025
```

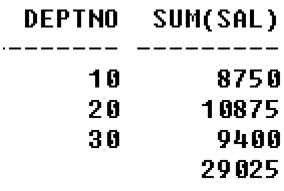
CUBE :-

Cube returns rows containing a subtotal for all combinations of columns, plus a row containing the grand total.

Example : -

Passing single column to CUBE :-

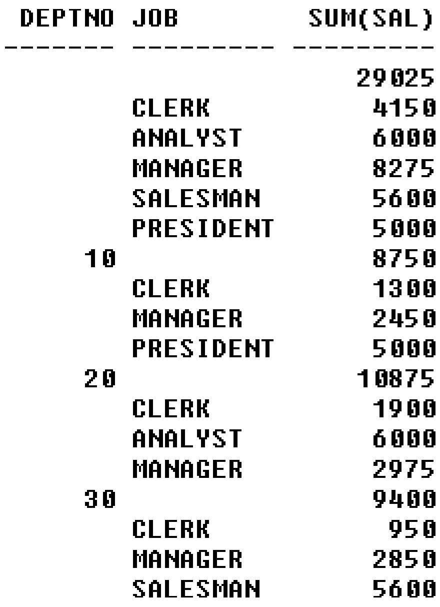
```
SQL>SELECT deptno,SUM(sal) FROM emp GROUP BY CUBE(deptno) ;
```



```
 DEPTNO   SUM(SAL)
------- ---------
     10      8750
     20     10875
     30      9400
            29025
```

Passing multiple columns to CUBE :-

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



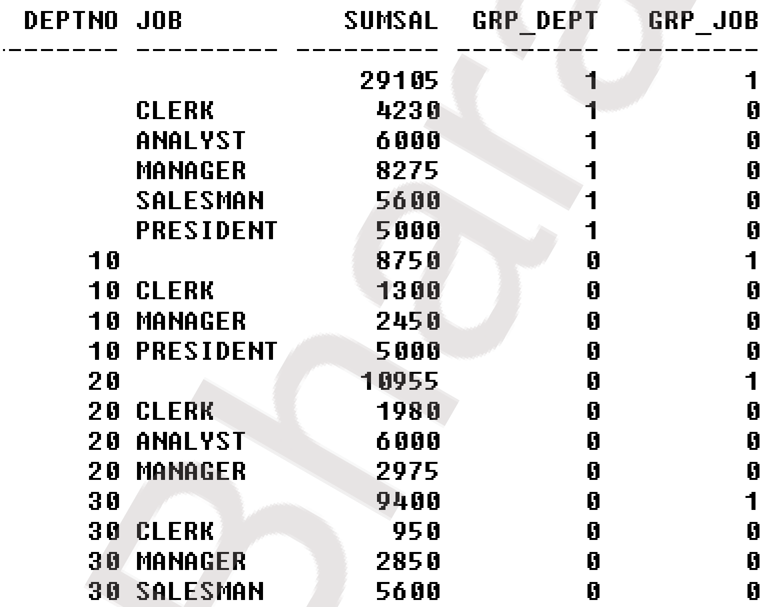
```
 DEPTNO JOB         SUM(SAL)
------- --------- ----------
                       29025
        CLERK           4150
        ANALYST         6000
        MANAGER         8275
        SALESMAN        5600
        PRESIDENT       5000
     10                 8750
        CLERK           1300
        MANAGER         2450
        PRESIDENT       5000
     20                10875
        CLERK           1900
        ANALYST         6000
        MANAGER         2975
     30                 9400
        CLERK            950
        MANAGER         2850
        SALESMAN        5600
```

GROUPING function:-

It can be quite easy to visually identify subtotals generated by rollups and cubes, but to do it programatically you really need something more accurate than the presence of null values in the grouping columns. This is where the GROUPING function comes in. It accepts a single column as a parameter and returns "1" if the column contains a null value generated as part of a subtotal by a ROLLUP or CUBE operation or "0" for any other value The following query demonstrates the usage of GROUPING function.

Example :-

```
SQL> SELECT deptno,job,SUM(sal) AS SUMSAL ,
        GROUPING(deptno) AS GRP_DEPT ,GROUPING(job)  AS GRP_JOB
       FROM emp
       GROUP BY CUBE(deptno,job);
```

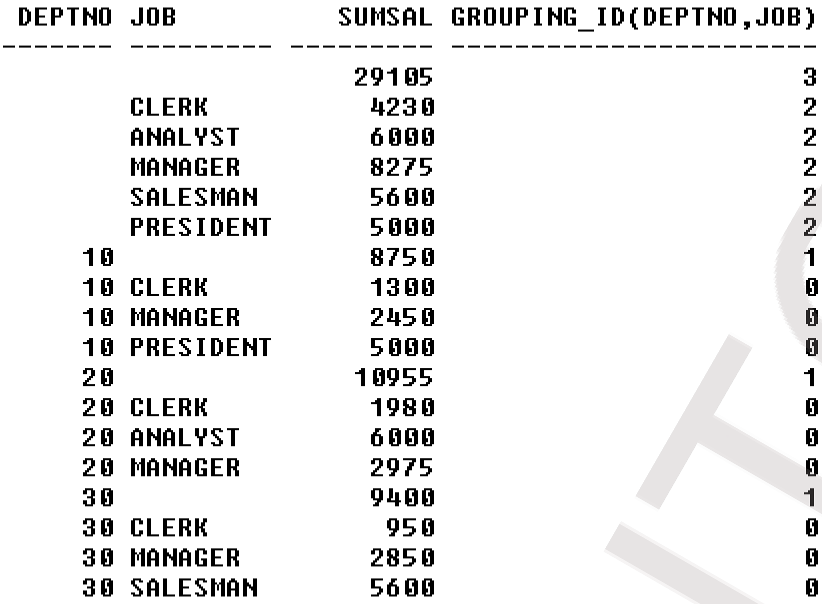


```
 DEPTNO JOB           SUMSAL   GRP_DEPT    GRP_JOB
------- --------- ---------- ---------- ----------
                       29105          1          1
        CLERK           4230          1          0
        ANALYST         6000          1          0
        MANAGER         8275          1          0
        SALESMAN        5600          1          0
        PRESIDENT       5000          1          0
     10                 8750          0          1
     10 CLERK           1300          0          0
     10 MANAGER         2450          0          0
     10 PRESIDENT       5000          0          0
     20                10955          0          1
     20 CLERK           1980          0          0
     20 ANALYST         6000          0          0
     20 MANAGER         2975          0          0
     30                 9400          0          1
     30 CLERK            950          0          0
     30 MANAGER         2850          0          0
     30 SALESMAN        5600          0          0
```

GROUPING_ID function :-

The GROUPING_ID function provides an alternate and more compact way to identify subtotal rows. Passing the GROUP BY columns as arguments, it returns a number indicating the GROUP BY level.

SQL>SELECT deptno,job,SUM(sal) AS SUMSAL ,GROUPING_ID(deptno,job)
FROM emp
GROUP BY CUBE(deptno,job) ;



```
 DEPTNO JOB           SUMSAL GROUPING_ID(DEPTNO,JOB)
------- --------- ---------- -----------------------
                       29105                       3
        CLERK           4230                       2
        ANALYST         6000                       2
        MANAGER         8275                       2
        SALESMAN        5600                       2
        PRESIDENT       5000                       2
     10                 8750                       1
     10 CLERK           1300                       0
     10 MANAGER         2450                       0
     10 PRESIDENT       5000                       0
     20                10955                       1
     20 CLERK           1980                       0
     20 ANALYST         6000                       0
     20 MANAGER         2975                       0
     30                 9400                       1
     30 CLERK            950                       0
     30 MANAGER         2850                       0
     30 SALESMAN        5600                       0
```

GROUPING SETS :-

Calculating all possible subtotals in a cube, especially those with many columns, can be quite an intensive process. If you don't need all the subtotals, this can represent a considerable amount of wasted effort. if cube applied on three columns then it gives 8 levels of subtotals.

If we only need a few of these levels of subtotaling we can use the GROUPING SETS expression and specify exactly which ones we need, saving us having to calculate the whole cube.

```
SQL>SELECT to_char(hiredate,'yyyy') as year,
    to_char(hiredate,'mon') as month,
    to_char(hiredate,'day') as day,
    count(*) as emps,
    GROUPING_ID(to_char(hiredate,'yyyy'),to_char(hiredate,'mon'),to_char(hiredate,'day'))AS
grouping_id
FROM   emp
GROUP BY GROUPING SETS((to_char(hiredate,'yyyy'),to_char(hiredate,'mon')), (to_char(hiredate,'yyyy'),
to_char(hiredate,'day')))
ORDER BY year,month,day
```

OCA questions :-

1 Evaluate the following SQL statement:

SQL>SELECT promo_category, AVG(promo_cost) Avg_Cost, AVG(promo_cost)*.25 Avg_Overhead
FROM promotions
WHERE UPPER(promo_category) IN ('TV', 'INTERNET','POST')
GROUP BY Avg_Cost
ORDER BY Avg_Overhead;

The above query generates an error on execution. Which clause in the above SQL statement causes the error?

A. WHERE
B. SELECT
C. GROUP BY
D. ORDER B

2 Which statement would display the highest credit limit available in each income level in each city in the CUSTOMERS table?

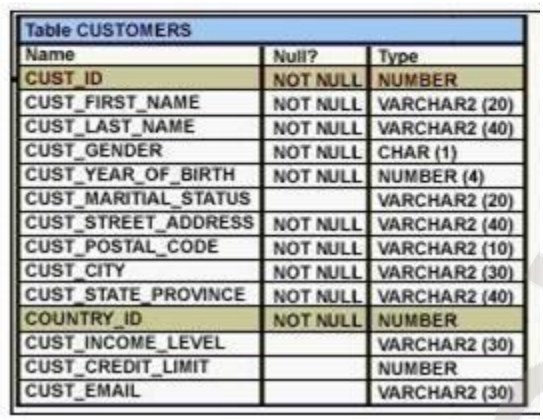


Table CUSTOMERS

Name	Null?	Type
CUST_ID	NOT NULL	NUMBER
CUST_FIRST_NAME	NOT NULL	VARCHAR2 (20)
CUST_LAST_NAME	NOT NULL	VARCHAR2 (40)
CUST_GENDER	NOT NULL	CHAR (1)
CUST_YEAR_OF_BIRTH	NOT NULL	NUMBER (4)
CUST_MARITIAL_STATUS		VARCHAR2 (20)
CUST_STREET_ADDRESS	NOT NULL	VARCHAR2 (40)
CUST_POSTAL_CODE	NOT NULL	VARCHAR2 (10)
CUST_CITY	NOT NULL	VARCHAR2 (30)
CUST_STATE_PROVINCE	NOT NULL	VARCHAR2 (40)
COUNTRY_ID	NOT NULL	NUMBER
CUST_INCOME_LEVEL		VARCHAR2 (30)
CUST_CREDIT_LIMIT		NUMBER
CUST_EMAIL		VARCHAR2 (30)

A. SELECT cust_city, cust_income_level, MAX(cust_credit_limit) FROM customers
GROUP BY cust_city, cust_income_level, cust_credit_limit;

B. SELECT cust_city, cust_income_level, MAX(cust_credit_limit) FROM customers
GROUP BY cust_city, cust_income_level;

C. SELECT cust_city, cust_income_level, MAX(cust_credit_limit) FROM customers
GROUP BY cust_credit_limit, cust_income_level, cust_city;

D. SELECT cust_city, cust_income_level, MAX(cust_credit_limit) FROM customers
GROUP BY cust_city, cust_income_level, MAX (cust_credit_limit);

Joins

In OLTP db tables are normalized and data organized in more than one table. For example sales DB is organized in customer,product,supplier tables etc. JOIN is an operation that combines rows from two or more tables or view.

ORACLE performs JOIN operation when more than one table is listed in FROM clause.

Tables participated in JOIN operation must share a meaningful relationship.

Types of JOINS :-

- Inner join or Equi Join
- Non-Equi Join
- Self Join
- Outer Join
- Cross Join

Inner Join :-

- In INNER JOIN join operation is performed based on common columns.
- To perform INNER JOIN there should be a common column in joining tables and name of the common column need not to be same.
- To perform INNER JOIN parent/child relationship between the tables is not mandatory.
- INNER join is most commonly used join in realtime.

Syntax :-

```
SQL>  SELECT <collist> FROM  <tab11> , <tab2>
        WHERE <join cond
         [AND <join cond> AND <cond> ------ ]
```

Join Condition :-

Child.fk = parent.pk (if relationship exists)

Tab1.commoncolumn = Tab2.commoncolumn (if there is no relationship)

- Oracle performs INNER JOIN by comparing fk value with pk value by using = operator.
- INNER JOIN is also called EQUI JOIN because join cond is based on = operator.
- INNER JOIN returns all rows from both tables that satisfies the JOIN CONDITION.
- No of JOIN CONDS depends on number of tables to be joined .
- To join N tables , min N-1 JOIN CONDS are required.

Guidelines:-

When writing a SELECT statement that joins tables, precede the column name with the table name or table alias for faster access and to avoid ambiguity.

Example:-

Display EMPNO,ENAME,DEPTNO,DNAME,LOC ?

```
SQL> SELECT    e.empno,  e. ename , e.sal, d. deptno ,  d. dname , d. loc
          FROM emp e,dept  d
          WHERE e.deptno = d.deptno;
```

Display ENAME of the employees working at NEW YORK location ?

```
SQL>SELECT e.ename
       FROM  emp e, dept d
      WHERE      e.deptno = d.deptno
                  AND
                  d.loc='NEW YORK' ;
```

Display ENAME of the employees working at NEW YORK location and earning more than 2000 ?

```
SQL>SELECT e.ename
       FROM emp e,dept d
       WHERE   e.deptno=d.deptno
                 AND
                 d.loc='NEW YORK'  and e.sal > 2000;
```

Joining more than 2 tables :-

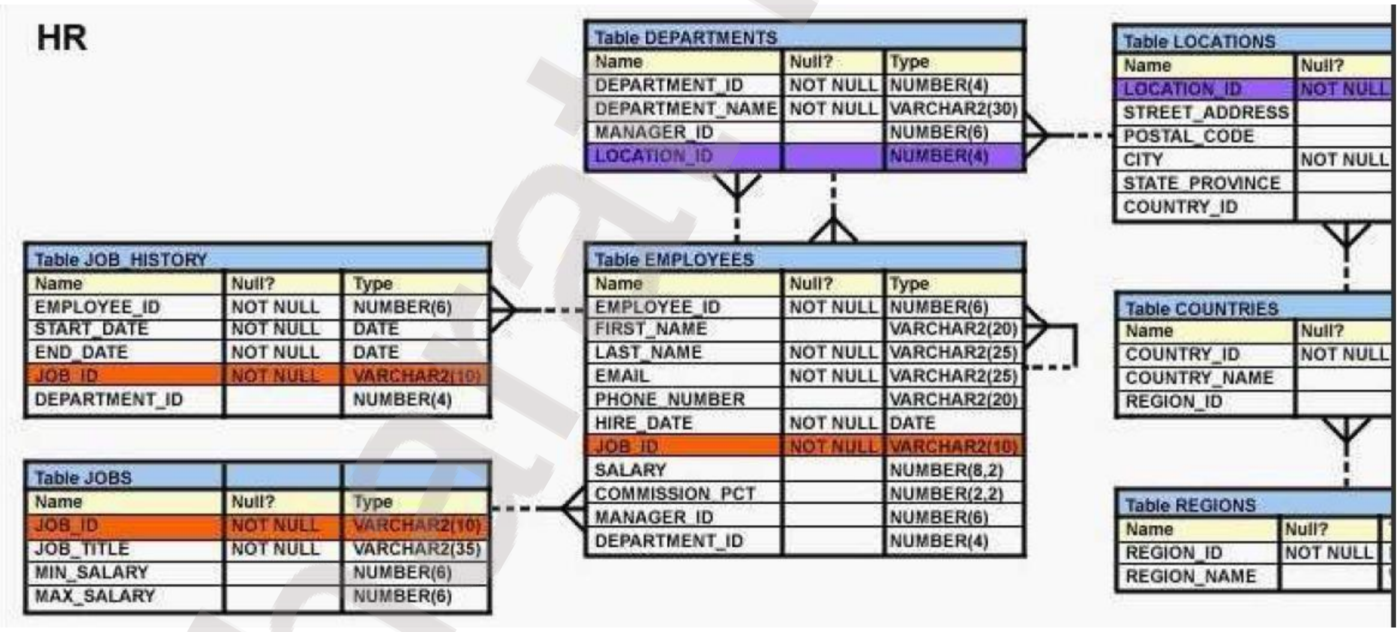


Table DEPARTMENTS		
Name	Null?	Type
DEPARTMENT_ID	NOT NULL	NUMBER(4)
DEPARTMENT_NAME	NOT NULL	VARCHAR2(30)
MANAGER_ID		NUMBER(6)
LOCATION_ID		NUMBER(4)

Table LOCATIONS	
Name	Null?
LOCATION_ID	NOT NULL
STREET_ADDRESS	
POSTAL_CODE	
CITY	NOT NULL
STATE_PROVINCE	
COUNTRY_ID	

Table JOB_HISTORY		
Name	Null?	Type
EMPLOYEE_ID	NOT NULL	NUMBER(6)
START_DATE	NOT NULL	DATE
END_DATE	NOT NULL	DATE
JOB_ID	NOT NULL	VARCHAR2(10)
DEPARTMENT_ID		NUMBER(4)

Table EMPLOYEES		
Name	Null?	Type
EMPLOYEE_ID	NOT NULL	NUMBER(6)
FIRST_NAME		VARCHAR2(20)
LAST_NAME	NOT NULL	VARCHAR2(25)
EMAIL	NOT NULL	VARCHAR2(25)
PHONE_NUMBER		VARCHAR2(20)
HIRE_DATE	NOT NULL	DATE
JOB_ID	NOT NULL	VARCHAR2(10)
SALARY		NUMBER(8,2)
COMMISSION_PCT		NUMBER(2,2)
MANAGER_ID		NUMBER(6)
DEPARTMENT_ID		NUMBER(4)

Table COUNTRIES	
Name	Null?
COUNTRY_ID	NOT NULL
COUNTRY_NAME	
REGION_ID	

Table JOBS		
Name	Null?	Type
JOB_ID	NOT NULL	VARCHAR2(10)
JOB_TITLE	NOT NULL	VARCHAR2(35)
MIN_SALARY		NUMBER(6)
MAX_SALARY		NUMBER(6)

Table REGIONS	
Name	Null?
REGION_ID	NOT NULL
REGION_NAME	

Display employee first_name , salary , job_title , department_name, city , street , state ,country_name,region_name ?

```
SQL>SELECT
          e.first_name,e.salary,j.job_title,d.department_name,l.city,l.street_address,l.state_province,
          c.country_name,r.region_name
 FROM           employees e,
                jobs j,
                departments d,
                locations l,
                countries c,
                regions r
 WHERE           e.job_id = j.job_id
                AND
                e.department_id = d.department_id
                AND
                d.location_id = l.location_id
                AND
                l.country_id = c.country_id
                AND
                c.region_id = r.region_id ;
```

ANSI Style :-

Oracle 9i now supports the ANSI/ISO SQL: 1999 standards. This allows easier product migration, but there is no performance increase compared to the existing syntax.

- ORACLE 9i supports SQL/99 standard joins.
- In SQL/99 style JOIN COND is specified by using ON clause or USING clause.
- in SQL/99 style to perform EQUI JOIN use keyword JOIN or INNER JOIN.
- JOIN COND is specified by using ON clause or USING clause.
- Use USING clause if common column name is same.

Example :-

Using ON clause :-

```
SQL>SELECT e.empno,e.ename,e.sal,d.dname,d.loc
        FROM emp e JOIN dept d
        ON (e.deptno = d.deptno) ;
```

Using USING clause :-

```
SQL>SELECT e.empno,e.ename,e.sal,d.dname,d.loc
        FROM emp e JOIN dept d
        USING (DEPTNO) ;
```

HINT :- In USING clause common column name should not be prefixed with table alias.

NOTE :- A join order is the order in which tables are accessed and joined together. For example, in a join order of table1, table2, and table3, table table1 is accessed first. Next, table2 is accessed, and its data is joined to table1. Finally, table3 is accessed, and its data is joined to the result of the join between table1 and table2.

Non Equi Join :-

When the Join Cond is based on equality operator, the join is said to be an equi join. When the join condition based on otherthan equality operator , the join is said to be a non-equi join.

Syntax:-

```
Select col1,col2,…….
From <table 1>,<table 2>
Where <join cond>   [AND <join cond> AND <cond>---- ]
```

→In NON-EQUI JOIN JOIN COND is not based on = operator. It is based on other than = operator usually BETWEEN or > or < operators.

Example:-

Display EMPNO,ENAME,SAL,GRADE ?

```
SQL> SELECT e.empno,e.ename,e.sal,s,grade
         FROM emp e, salgrade s
         WHERE e.sal BETWEEN s.losal AND s.hisal;
```

Display EMPNO,ENAME,SAL,DNAME,LOC,GRADE ?

```
SQL>SELECT e.empno,e.ename,e.sal,d.dname,d.loc,s.grade
         FROM emp e,dept d,salgrade s
         WHERE e.deptno = d.deptno
                         AND
                 e.sal between g.losal AND g.hisal ;
```

ANSI Style :-

Display EMPNO,ENAME,SAL,GRADE ?

```
SQL>SELECT e.empno,e.ename,e.sal,s.grade
         FROM emp e JOIN salgrade g
         ON ( e.sal  BETWEEN g.losal AND g.hisal) ;
```

Display EMPNO,ENAME,SAL,DNAME,LOC,GRADE ?

```
SQL>SELECT e.empno,e.ename,e.sal,d.dname,d.loc,s.grade
         FROM emp e JOIN dept d
         USING(deptno)
         JOIN salgrade s
         ON (e.sal BETWEEN g.losal and g.hisal) ;
```

Self Join :-

- Joining a table to itself is called Self Join.
- Self Join is performed when tables having self refrential integrity.
- To perform Self Join same table must be listed twice with different alias.
- Self Join is Equi Join within the table.

Syntax :-

```
SQl>SELECT <collist>
       From Table1 T1, Table1 T2
        Where T1.Column1=T2.Column2;
```

Example:-

Display EMPNO,ENAME,SAL,MGRNAME ?

```
SQL>SELECT e.empno,e.ename,e.sal,m.ename
          FROM emp e, emp m
          WHERE e.mgr = m.empno ;
```

ANSI Style:-

Display EMPNO,ENAME,SAL,MGRNAME ?

```
SQL>SELECT e.empno , e.ename,e.sal,m.ename
          FROM emp e  JOIN dept d
          ON (e.mgr = m.empno)  ;
```

Display EMPNO,ENAME,SAL,DNAME,LOC,GRADE,MGRNAME ?

```
SQL>SELECT e.empno,e.ename,e.sal,d.dname,d.loc,,s.grade,m.ename
          FROM emp e JOIN dept d
          USING(deptno)
          JOIN salgrade s
         ON (e.sal BETWEEN g.losal  AND g.hisal)
          JOIN emp m
          ON ( e.mgr = m.empno) ;
```

Outer Join:-

Equi join returns only matching records from both the tables but not unmatched record, an outer join retrieves a row even when one of the column in the join contains a null value. For example there are two tables one is CUSTOMER that stores customer information and another ORDERS table that stores orders placed by customers , INNER JOIN returns only the list of customer who placed orders,but OUTER JOIN also returns customer who did not placed any order. Outer join is 3 types.

- LEFT OUTER JOIN
- RIGHT OUTER JOIN
- FULL OUTER JOIN

To perform OUTER JOIN use Oracle Proprietary operator (+) .

Left Outer Join:-

LEFT OUTER JOIN returns all rows(matched and unmatched) from LEFT SIDE table and matching records from RIGHT SIDE table. To perform LEFT OUTER JOIN (+) should be on RIGHT SIDE.

Syntax :-

```
SELECT <collist> FROM <tablist>
          WHERE  t1.commoncolumn  =  t2.commoncolumn (+)
```

Example :-

Display EMPNO,ENAME,DNAME,LOC and also display employee list who are not assigned to any dept?

```
SQL>    SELECT e.empno,e.ename,d.dname,d.loc
        FROM emp e, dept d
        WHERE e.deptno  = d.deptno (+) ;
```

ANSI Style :-

In SQL/92 standard use keyword LEFT OUTER JOIN instead of using operator (+) .

Display EMPNO,ENAME,DNAME,LOC and also display employee list who are not assigned to any dept?

```
SQL>    SELECT e.empno,e.ename,d.dname,d.loc
        FROM emp e LEFT OUTER JOIN dept d
        USING(deptno) ;
```

Right Outer Join:-

RIGHT OUTER JOIN returns all rows(matched and unmatched) from RIGHT SIDE table and matching records from LEFT SIDE table. To perform RIGHT OUTER JOIN use (+) on LEFT SIDE.

Syntax :-

```
SELECT <collist> FROM <tablist>
WHERE  t1.commoncolumn(+)  =  t2.commoncolumn
```

Example :-

Display EMPNO,ENAME,DNAME,LOC and also display department which are empty ?

```
SQL>    SELECT e.empno,e.ename,d.dname,d.loc
        FROM emp e, dept d
        WHERE e.deptno(+)  = d.deptno ;
```

ANSI Style :-

In SQL/92 standard use keyword RIGHT OUTER JOIN instead of using operator (+) .

Display EMPNO,ENAME,DNAME,LOC and also display departments which are empty ?

```
SQL>    SELECT e.empno,e.ename,d.dname,d.loc
        FROM emp e RIGHT OUTER JOIN dept d
        USING(deptno) ;
```

Full Outer Join:-

→Returns all rows (matched and unmatched) from both tables.
→Prior to oracle 9i doesn't support FULL OUTER JOIN.
→To perform FULL OUTER JOIN in prior to ORACLE 9i.

```
SQL>    SELECT e.empno,e.ename,d.dname,d.loc
        FROM emp e, dept d
        WHERE e.deptno  = d.deptno (+) ;
        UNION
        SELECT e.empno,e.ename,d.dname,d.loc
        FROM emp e, dept d
        WHERE e.deptno(+)  = d.deptno ;
```

HINT :-

(+) should be either left side or right side but cannot be on both sides.

CROSS JOIN :-

- CROSS JOIN returns cross product of two tables.
- Each record of one table is joined to each and every record of another table.
- If table1 contains 10 records and table2 contains 5 records then CROSS JOIN between table1 and table2 returns 50 records.
- ORACLE performs CROSS JOIN when we submit query without JOIN COND.

Syntax :-

```
SQL>SELECT  col1 ,col2  FROM  tab1 , tab2 ;
```

Example:-

ORDER

ORDAMT
100000

DISCOUNTS

DIS
10
12
15

Display ORDAMT for each and every DISCOUNT percentage ?

```
SQL>SELECT o.ordamt,d.dis, (o.ordamt*d.dis)/100  AS  amount
         FROM orders o,discounts d ;
```

ANSI Style :-

```
SQL>SELECT o.ordamt,d.dis, (o.ordamt*d.dis)/100  AS  amount
         FROM orders o CROSS JOIN discounts d ;
```

Natural Join :-

- NATURAL JOIN is possible in ANSI SQL/92 standard.
- NATURAL JOIN is similar to EQUI JOIN.
- NATURAL JOIN is performed only when common column name is same.
- in NATURAL JOIN no need to specify join condition explicitly , ORACLE automatically performs join operation on the column with same name..

Example :-

```
SQL>SELECT e.empno,e.ename,e.sal,d.dname,d.loc
        FROM emp e NATURAL JOIN dept d ;
```

Above query performs JOIN operation on DEPTNO.

OCA question :-

1 Which two statements are true regarding the USING and ON clauses in table joins? (Choose two.)

A. Both USING and ON clauses can be used for equijoins and nonequijoins.
B. A maximum of one pair of columns can be joined between two tables using the ON clause.
C. The ON clause can be used to join tables on columns that have different names but compatible data types.
D. The WHERE clause can be used to apply additional conditions in SELECT statements containing the ON or the USING clause.

2 Evaluate the following SQL statement:

```
SQL>SELECT p.promo_id, p.promo_name, s.prod_id
 FROM sales s RIGHT OUTER JOIN promotions p
 ON (s.promo_id = p.promo_id);
```

Which statement is true regarding the output of the above query?

A. It gives the details of promos for which there have been sales.
B. It gives the details of promos for which there have been no sales.
C. It gives details of all promos irrespective of whether they have resulted in a sale or not.
It gives details of product IDs that have been sold irrespective of whether they had a promo or not.

3 from PRODUCTS, SALES, and CUSTOMERS tables.

You need to generate a report that gives details of the customer's last name, name of the product, and the quantity sold for all customers in 'Tokyo'.

Which two queries give the required result? (Choose two.)

```
A. SELECT c.cust_last_name,p.prod_name, s.quantity_sold
FROM sales s JOIN products p
USING(prod_id) JOIN customers c USING(cust_id)
WHERE c.cust_city='Tokyo';
```

```
B.  SELECT c.cust_last_name, p.prod_name, s.quantity_sold
   FROM products p JOIN sales s JOIN customers c ON(p.prod_id=s.prod_id)
      ON(s.cust_id=c.cust_id) WHERE c.cust_city='Tokyo';
```

```
C. SELECT c.cust_last_name, p.prod_name, s.quantity_sold
   FROM products p JOIN sales s
    ON(p.prod_id=s.prod_id) JOIN customers c ON(s.cust_id=c.cust_id)
              AND c.cust_city='Tokyo';
```

```
D. SELECT c.cust_id,c.cust_last_name,p.prod_id, p.prod_name, s.quantity_sold
        FROM products p JOIN sales s
       USING(prod_id) JOIN customers c
      USING(cust_id)
     WHERE c.cust_city='Tokyo';
```

Set Operators :-

- UNION
- UNION ALL
- INTERSECT
- MINUS

Syntax :-

SELECT statement 1
UNION / UNION ALL / INTERSECT / MINUS
SELECT statement 2 ;

Rules :-

1 No of columns returned by first query must be equal to no of columns returned by second query
2 Corrosponding columns datatype type must be same.

UNION:-

- UNION operator combines data returned by two SELECT statement.
- eliminates duplicates.
- Sorts result.

Example :-

1
```
SQL>SELECT job FROM emp WHERE deptno=10
UNION
 SELECT job FROM emp WHERE deptno=20 ;
```

2
```
SQL>SELECT job,sal FROM emp WHERE deptno=10
UNION
SELECT job,sal FROM emp WHERE deptno=20
 ORDER BY sal ;
```

NOTE:- ORDER BY clause must be used with last query.

UNION ALL:-

→UNION ALL is similar to UNION but it includes duplicates

Example :-

```
SQL>SELECT job FROM emp WHERE deptno=10
        UNION ALL
        SELECT job FROM emp WHERE deptno=20  ;
```

UNION vs JOIN :-

UNION	JOIN
Union combines data	Join relates data
Union is performed on similar structures	Join can be performed also be performed on dissimilar structures also

Scenario :-

EMP1

EMPNO	ENAME	DNO
1	A	10
2	B	20

EMP2

EMPNO	ENAME	DNO
100	X	10
101	Y	20

DEPT:-

DNO	DNAME	LOC
10	ACCT	HYD
20	SALES	HYD

Display all employee list along with department names and locations ?

Solution :- (EMP1 union EMP2) Join DEPT

INTERSECT:-

INTERSECT operator returns common values from the result of two SELECT statements.

Example:-

Display common jobs belongs to 10th and 20th departments ?

```
SQL>SELECT job FROM emp WHERE deptno=10
INTERSECT
SELECT job FROM emp WHERE deptno=20;
```

MINUS:-

MINUS operator returns values present in the result of first SELECT statement and not present in the result of second SELECT statement.

Example:-

Display jobs in 10th dept and not in 20th dept ?

```
SQL>SELECT job FROM emp WHERE deptno=10
MINUS
SELECT job FROM emp WHERE deptno=20;
```

OCA questions :-

1 Evaluate the following SQL statement:

```
SQL> SELECT cust_id, cust_last_name "Last Name"
          FROM cutomers
            WHERE country_id = 10
            UNION
            SELECT cust_id CUST_NO, cust_last_name
            FROM customers
            WHERE country_id = 30;
```

Which ORDER BY clauses are valid for the above query? (Choose all that apply.)

A. ORDER BY 2,1

B. ORDER BY CUST_NO
C. ORDER BY 2,cust_id
D. ORDER BY "CUST_NO"
E. ORDER BY "Last Name"

2 Which statement is true regarding the INTERSECT operator?
A. It ignores NULL values.
B. Reversing the order of the intersected tables alters the result.
C. The names of columns in all SELECT statements must be identical.
D. The number of columns and data types must be identical for all SELECT statements in the query.

Psuedo Columns

Pseudo columns are not actual columns in a table but they behave like columns, these columns doesn't really exist in DB but available for use. you can select values from a pseudo column but you cannot insert, update ,delete the pseudo column values. Pseudo columns are assigned with values by ORACLE like a normal db column but not stored on disk.
SQL and PL/SQL recognizes following pseudo columns which return specific data

- ROWID
- ROWNUM
- USER
- LEVEL
- CURRVAL and NEXTVAL
- SYSDATE
- SYSTIMESTAMP
- ORA_ROWSCAN

SYSDATE & SYSTIMESTAMP :-

Returns current DATE and TIMESTAMP.
SQL>SELECT sysdate , systimestamp FROM dual ;

UID & USER :-

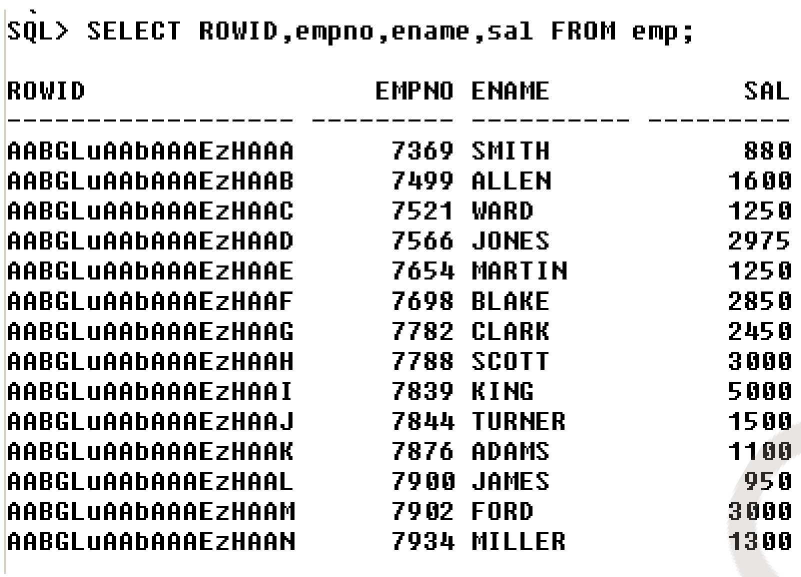
Returns User ID and name of the database user.
SQL>SELECT uid, user FROM dual ;

```
UID     USER
50      SCOTT
```

ROWID :-

- ROWID returns physical address of a row in a database table and it is the fastest way to retrieve a row. Faster than index even.
- since ROWID represents physical location of a row no two rows within in the same table will have the same ROWIDs.
- Because ROWID represent physical location of a row , the ROWID will change every time the record is physically moved.
- Use DBMS_SQL.LAST_ROW_ID to get the ROWID of the last row processed.
- ROWID can be used to DELETE duplicate records in a table.
- ROWID can be used in SELECT and WHERE clauses.
- In table records are arranged based on their ROWIDs i.e first record ROWID is always minimum and last record is always maximum.

To Display ROWIDs of employee records execute the following command ?



```
SQL> SELECT ROWID,empno,ename,sal FROM emp;

ROWID                   EMPNO ENAME             SAL
------------------ ---------- ---------- ----------
AABGLuAAbAAAEzHAAA       7369 SMITH             800
AABGLuAAbAAAEzHAAB       7499 ALLEN            1600
AABGLuAAbAAAEzHAAC       7521 WARD             1250
AABGLuAAbAAAEzHAAD       7566 JONES            2975
AABGLuAAbAAAEzHAAE       7654 MARTIN           1250
AABGLuAAbAAAEzHAAF       7698 BLAKE            2850
AABGLuAAbAAAEzHAAG       7782 CLARK            2450
AABGLuAAbAAAEzHAAH       7788 SCOTT            3000
AABGLuAAbAAAEzHAAI       7839 KING             5000
AABGLuAAbAAAEzHAAJ       7844 TURNER           1500
AABGLuAAbAAAEzHAAK       7876 ADAMS            1100
AABGLuAAbAAAEzHAAL       7900 JAMES             950
AABGLuAAbAAAEzHAAM       7902 FORD             3000
AABGLuAAbAAAEzHAAN       7934 MILLER           1300
```

We can also retrieve records based on ROWIDs as follows.

SQL>SELECT * FROM emp WHERE ROWID = 'AABGLuAAbAAAezHAAN';

Display ROWID of the first record ?

SQL>SELECT MIN(ROWID) FROM emp;

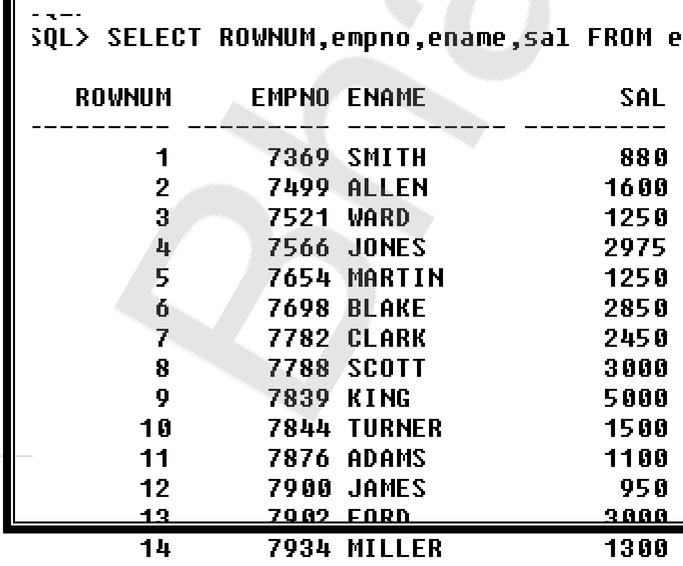
ROWNUM :-

- ROWNUM is also a pseudo column
- ROWNUM represents the sequential order in which ORACLE has retrieved the row and it will change from query to query .
- ROWNUM changes from query to query but ROWID is permanent.
- ROWNUM can be used in SELECT and WHERE clauses.

Questions based on ROWID & ROWNUM :-

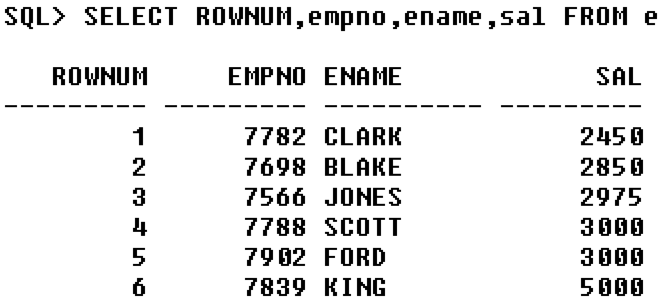
How do I limit the number of rows returned by a query?
How do I write a query to get the Top-N salaries from the employee table?
How can I add unique, sequential numbers to an existing table?
How can I differentiate between two completely identical rows?
How can I find a faster way to retrieve a row?
How can I find the last row processed in a big batch?

Display employee records with record numbers ?



```
SQL> SELECT ROWNUM,empno,ename,sal FROM emp;

    ROWNUM      EMPNO ENAME             SAL
---------- ---------- ---------- ----------
         1       7369 SMITH             800
         2       7499 ALLEN            1600
         3       7521 WARD             1250
         4       7566 JONES            2975
         5       7654 MARTIN           1250
         6       7698 BLAKE            2850
         7       7782 CLARK            2450
         8       7788 SCOTT            3000
         9       7839 KING             5000
        10       7844 TURNER           1500
        11       7876 ADAMS            1100
        12       7900 JAMES             950
        13       7902 FORD             3000
        14       7934 MILLER           1300
```

In the above result the ROWNUM generated for KING record is 9.
Dispplay employee records earning more than 2000 ?



```
SQL> SELECT ROWNUM,empno,ename,sal FROM emp WHERE sal > 2000;

    ROWNUM      EMPNO ENAME             SAL
---------- ---------- ---------- ----------
         1       7782 CLARK            2450
         2       7698 BLAKE            2850
         3       7566 JONES            2975
         4       7788 SCOTT            3000
         5       7902 FORD             3000
         6       7839 KING             5000
```

In the above result the ROWNUM generated for KING record is 6, So ROWNUM changes from query to query.

We can also retrieve records based on their record number. For example to display 1st record ?

SQL>SELECT * FROM emp WHERE ROWNUM=1 ;

To display first 5 records in emp table ?

SQL>SELECT * FROM emp WHERE ROWNUM <= 5;

But the following query returns error.

SQL>SELECT * FROM emp WHERE ROWNUM > 6;

NOTE :- Because ROWNUM is generated after retrieving record. So with ROWNUM we cannot use > , >= operators.

ORA_ROWSCAN :-

ORA_ROWSCN returns the system change number (SCN) of the last change inside the block containing a row. It can return the last modification for the row if the table is created with the option ROWDEPENDENCIES (default is NOROWDEPENDENCIES). The function SCN_TO_TIMESTAMP allows you to convert SCN to timestamp.

SQL> select ename, ORA_ROWSCN, SCN_TO_TIMESTAMP(ORA_ROWSCN)
from emp where empno=7369;

```
ENAME     ORA_ROWSCN            SCN_TO_TIMESTAMP(ORA_ROWSCN)
-----------------------------------------------------------------
SMITH      2113048              20/12/2008 16:59:51.000
```

Subqueries

Subquery:-

- Query embedded in another query is called subquery.
- One query is called inner/child/subquery.
- Another query is called outer/parent/main query.
- The result of inner query acts as an input to outer query.
- Outer query can be INSERT,UPDATE,DELETE,SELECT
- Inner query must be always SELECT
- Subqueries can appear in

  WHERE CLAUSE
HAVING CLAUSE
FROM CLAUSE
SELECT CLAUSE

Types of SUBQUERIES :-

- ➔ Single Row Subqueries
- ➔ Multi Row Subqueries
- ➔ Nested Queries
- ➔ Muliti Column Subqueries
- ➔ Co-related Subqueries

SINGLE ROW SUBQUERIES:-

If inner query returns only one row then it is called single row subquery.

Syntax :-

```
SELECT <collist> FROM <tabname>
            WHERE  colname  OP (SELECT statement)
```

OP can be < > <= >= = <>

Example :-

Subqueries in WHERE clause :-

Display employee records whose job equals to job of SMITH?

```
SQL>SELECT * FROM emp
        WHERE job = (SELECT job FROM emp WHERE  ename='SMITH') ;
```

Display employee name earning maximum salary ?

```
SQL>SELECT ename FROM emp
                WHERE sal = (SELECT MAX(sal) FROM emp) ;
```

Display all records except last record ?

```
SQL>SELECT * FROM emp
                WHERE ROWID <(SELECT MAX(ROWID) FROM emp) ;
```

Subqueries with BETWEEN operator:-

Display employee reocrds earning salary between min sal of 10 dept and max sal of 30 dept ?

```
SQL>SELECT * FROM emp
       WHERE sal BETWEEN  (SELECT MIN(sal) FROM emp WHERE deptno=10)
                                       AND
                                       (SELECT MAX(sal) FROM emp WHERE deptno=30) ;
```

Subqueries in HAVING clause:-

Display departmentss whose avg(sal) geater than avg(sal) of 10 dept?

```
SQL>SELECT deptno FROM emp
                              GROUP BY deptno
                               HAVING AVG(sal) > (SELECT AVG(sal) FROM emp
                                                                      WHERE deptno=10) ;
```

Subqueries in UPDATE command :-

Update employee salary to maximum salary whose empno=7369 ?

```
SQL>UPDATE emp SET sal = (SELECT MAX(sal) FROM emp)  WHERE  EMPNO=7369 ;
```

Swap employee salaries whose empno in (7369,7499) ?

```
SQL>UPDATE emp SET sal=DECODE(empno,7369,(SELECT sal FROM emp
                                                                  WHERE empno=7499),
                                                  7499,(SELECT sal FROM emp
                                                                    WHERE empno=7369) );
```

Subqueries in DELETE command:-

Delete employee record whose job equals to job of SMITH ?

```
SQL>DELETE FROM emp
                            WHERE    job=    (SELECT    job    FROM    emp    WHERE
                                  ename='SMITH');
```

Multi Row Subqueries:-

if inner query returns more than one row then it is called multi row subquery.

Syntax :-

```
SQL>SELECT <collist> FROM <tabname>
                               WHERE colname OP (SELECT statement) ;
```

OP must be IN ,   NOT IN,   ANY, ALL

Example :-

Displaye employee records whose job equals to job of SMITH or job of BLAKE ?

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

```
          WHERE job  IN (SELECT job FROM emp WHERE ename IN (‘SMITH’,’BLAKE’));
```

Displaye employee records who are earning minimum and maximum salaries ?

```
SQL>SELECT * FROM emp WHERE sal  IN (SELECT  MIN(sal) FROM emp
                                              UNION
                                            SELECT MAX(sal) FROM emp);
```

Display 4th,7th,11th record in EMP table ?

```
SQL>SELECT * FROM emp
                WHERE  ROWID  IN (SELECT DECODE(ROWNUM,4,ROWID,
                                                    7,ROWID,
                                                   11,ROWID)
                                  FROM emp);
```

ANY operator:-

Compares a value to each value in a list or returned by a query. Must be preceded by =, !=, >, <, <=, >=. Evaluates to FALSE if the query returns no rows.

Example:-

Select employees whose salary is greater than any salesman’s salary ?

```
SQL>SELECT ename  FROM emp
          WHERE SAL > ANY ( SELECT sal FROM emp  WHERE  job = 'SALESMAN');
```

ALL operator :-

Compares a value to every value in a list or returned by a query. Must be preceded by =, !=, >, <, <=, >=. evaluates to TRUE if the query returns no rows.

Example:-

Select employees whose salary is greater than every salesman’s salary ?

```
SQL>SELECT ename  FROM emp
          WHERE SAL > ALL ( SELECT sal FROM emp  WHERE  job = 'SALESMAN');
```

Nested Queries:-

→A subquery embedded in another subquery is called NESTED QUERY.
→Queries can be nested upto 255 level.

Example :-

Display employee name earning second maximum salary ?

```
SQL>SELECT ename FROM emp
  WHERE sal = (SELECT MAX(sal) FROM EMP
                   WHERE sal < (SELECT MAX(sal) FROM emp)) ;
```

Update the employee salary to maximum salary of SALES dept ?

```
SQL>UPDATE emp
     SET sal = (SELECT MAX(sal) FROM emp
                WHERE deptno = (SELECT deptno FROM dept
                                 WHERE dname=’SALES’)) ;
```

Multi Column Subqueries:-

If inner query returns more than one column value then it is called MULTI COLUMN subquery.

Example :-

Display employee names earning maximum salaries in their dept ?

```
SQL>SELECT ename FROM emp
                    WHERE (deptno,sal) IN (SELECT deptno,MAX(sal)
                                           FROM emp
                                           GROUP BY deptno) ;
```

Co-related Subqueries:-

If a subquery references one or more columns of parent query is called CO-RELATED subquery because it is related to outer query. This subquery executes once for each and every row of main query.

Example :-

→Display employee names earning more than avg(sal) of their dept ?

```
SQL>SELECT ename FROM emp x
                  WHERE sal > (SELECT AVG(sal) FROM emp
                                         WHERE deptno=x.deptno);
```

→Display employee names earning more than their manager ?

```
SQL>SELECT ename FROM emp x
                    WHERE sal > (SELECT sal FROM emp
                                               WHERE empno=x.mgr);
```

→Delete duplicate records in a table ?

```
SQL>DELETE FROM emp X
      WHERE ROWID > (SELECT MIN(ROWID) FROM emp
                           WHERE empno=x.empno
                                        AND
                                   ename=x.ename
                                        AND
                                   sal=x.sal) ;
```

Display top 3 maximum salaries in emp table ?

```
SQL>SELECT DISTINCT sal FROM emp a
        WHERE 3 > (SELECT COUNT(DISTINCT sal)
                    FROM emp b
                     WHERE a.sal < b.sal) ;
```

Using EXISTS operator :-

→EXISTS operator returns TRUE or FALSE.

→If inner query returns at least one record then EXISTS returns TRUE otherwise returns FALSE.

→ORACLE recommends EXISTS and NOT EXISTS operators instead of IN and NOT IN.

Display dept which not empty ?

```
SQL>SELECT * FROM dept d
      WHERE EXISTS (SELECT * FROM emp  WHERE deptno =d.deptno) ;
```

Display dept which is empty ?

```
SQL>SELECT * FROM dept d
```

```
        WHERE NOT EXISTS (SELECT * FROM emp
                                     WHERE deptno = d.deptno);
```

OCA question :-

1 Which three statements are true regarding subqueries? (Choose three.)

A. Subqueries can contain GROUP BY and ORDER BY clauses.
B. Main query and subquery can get data from different tables.
C. Main query and subquery must get data from the same tables.
D. Subqueries can contain ORDER BY but not the GROUP BY clause.
E. Only one column or expression can be compared between the main query and subquery.
F. Multiple columns or expressions can be compared between the main query and subquery.

2 Which three statements are true about multiple-row subqueries? (Choose three.)

A. They can contain a subquery within a subquery.
B. They can return multiple columns as well as rows.
C. They cannot contain a subquery within a subquery.
D. They can return only one column but multiple rows.
E. They can contain group functions and GROUP BY and HAVING clauses.
F. They can contain group functions and the GROUP BY clause, but not the HAVING clause.

3 Which two statements are true regarding the execution of the correlated subqueries? (Choose two.)

A. The nested query executes after the outer query returns the row.
B. The nested query executes first and then the outer query executes.
C. The outer query executes only once for the result returned by the inner query.
D. Each row returned by the outer query is evaluated for the results returned by the inner query.

4 Evaluate the following SQL statement:

```
SQL> SELECT cust_id, cust_last_name
     FROM customers
     WHERE cust_credit_limit IN (select cust_credit_limit
                                 FROM customers
                                 WHERE cust_city ='Singapore');
```

Which statement is true regarding the above query if one of the values generated by the subquery is NULL?

A. It produces an error.
B. It executes but returns no rows.
C. It generates output for NULL as well as the other values produced by the subquery.
D. It ignores the NULL value and generates output for the other values produced by the subquery.

5 Which statement is true regarding subqueries?

A. The LIKE operator cannot be used with single-row subqueries.
B. The NOT IN operator is equivalent to IS NULL with single-row subqueries.
C. =ANY and =ALL operators have the same functionality in multiple-row subqueries.
D. The NOT operator can be used with IN, ANY, and ALL operators in multiple-row subqueries.

6 You want to update the CUST_INCOME_LEVEL and CUST_CREDIT_LIMIT columns for the customer with the CUST_ID 2360. You want the value for the CUST_INCOME_LEVEL to have the same value as that of the customer with the CUST_ID 2560 and the CUST_CREDIT_LIMIT to have the same value as that of the customer with CUST_ID 2566.

Which UPDATE statement will accomplish the task?

```
A. UPDATE customers
           SET cust_income_level = (SELECT cust_income_level
                                          FROM customers
                                        WHERE cust_id = 2560),
             cust_credit_limit = (SELECT cust_credit_limit
                                   FROM customers
                                  WHERE cust_id = 2566)
  WHERE cust_id=2360;

B. UPDATE customers
   SET (cust_income_level,cust_credit_limit) = (SELECT cust_income_level, cust_credit_limit
                                                 FROM customers
                                                 WHERE cust_id=2560 OR cust_id=2566)
WHERE cust_id=2360;


C. UPDATE customers
SET (cust_income_level,cust_credit_limit) = (SELECT cust_income_level, cust_credit_limit
                                             FROM customers
                                             WHERE cust_id IN(2560, 2566) )
WHERE cust_id=2360;

D. UPDATE customers

        SET (cust_income_level,cust_credit_limit) = (SELECT cust_income_level, cust_credit_limit
                                                     FROM customers
                                                     WHERE cust_id=2560 AND cust_id=2566)
WHERE cust_id=2360;
```

In-line Views :-

The inline view is a construct in Oracle SQL Where you can place a query in the SQL FROM clause, just as if the query was a tablename. A common use of in-line views is to simplify complex queries by removing join operations and converting separate queries into single query.

Using in-line views :-

→Column alias can be used in WHERE clause.

→ Window functions can be used in WHERE clause.

→Result of one process can be used in another process.

Syntax :-

SELECT <collist> FROM (SELECT statement) <alias> ;

The result of inner query acts as TABLE for outer query.

Example :-

Display top 3 maximum salaries in EMP table ?

```
SQL>SELECT  sal FROM
                        (SELECT DISTINCT sal FROM emp ORDER BY sal DESC)
    WHERE ROWNUM <=3 ;
```

Using DENSE_RANK function :-

```
SQL>SELECT DISTINCT sal FROM
                                (SELECT sal,
                                DENSE_RANK() OVER (ORDER BY sal DESC) AS RNK
                                FROM emp)
   WHERE RNK<=3 ;
```

Display records N thru M from table. the general form of this is as follows ?

```
SQL>SELECT *   FROM
                (SELECT a.*, ROWNUM rn
                FROM (enter your query here) a
                 WHERE ROWNUM <= :MAX_ROW)
 WHERE rn >= :MIN_ROW;

SQL>SELECT *   FROM
 WHERE rn >= 2;
SELECT a.*, ROWNUM rn  FROM
                      (SELECT * FROM emp) a WHERE ROWNUM <= 6)
```

The above query display records from 2 to 5;

Display department wise maximum salaries , in report show department names ?

```
SQL>SELECT  d.dname,e.maxsal
    FROM   dept d , (SELECT deptno,MAX(sal)  maxsal FROM emp
                       GROUP BY deptno) e
    WHERE e.deptno = d.deptno ;
```

WITH Clause : Subquery Factoring

The WITH clause, or subquery factoring clause, is part of the SQL-99 standard and was added into the Oracle SQL syntax in Oracle 9.2. The WITH clause may be processed as an inline view or resolved as a temporary table. The advantage of the latter is that repeated references to the subquery may be more efficient as the data is easily retrieved from the temporary table, rather than being requeried by each reference. You should assess the performance implications of the WITH clause on a case-by-case basis.

example shows how the WITH clause can be used to reduce repetition and simplify complex SQL statements.

for each employee we want to know how many other people are in their department. Using an inline view we might do the following.

```
SQL>SELECT e.ename AS employee_name,
    dc.dept_count AS emp_dept_count
FROM   emp e,
    (SELECT deptno, COUNT(*) AS dept_count
     FROM   emp
```

```
        GROUP BY deptno) dc
WHERE  e.deptno = dc.deptno;
```

Using a WITH clause this would look like the following.

```
SQL>WITH dept_count AS (
  SELECT deptno, COUNT(*) AS dept_count
  FROM   emp
  GROUP BY deptno)
SELECT e.ename AS employee_name,
       dc.dept_count AS emp_dept_count
FROM   emp e,
       dept_count dc
WHERE  e.deptno = dc.deptno;
```

The difference seems rather insignificant here.

What if we also want to pull back each employees manager name and the number of people in the managers department? Using the inline view it now looks like this.

```
SELECT e.ename AS employee_name,
       dc1.dept_count AS emp_dept_count,
       m.ename AS manager_name,
       dc2.dept_count AS mgr_dept_count
FROM   emp e,
       (SELECT deptno, COUNT(*) AS dept_count
        FROM   emp
        GROUP BY deptno) dc1,
       emp m,
       (SELECT deptno, COUNT(*) AS dept_count
        FROM   emp
        GROUP BY deptno) dc2
WHERE  e.deptno = dc1.deptno
AND    e.mgr = m.empno
AND    m.deptno = dc2.deptno;
```

Using the WITH clause this would look like the following.

```
SQL>WITH dept_count AS (
  SELECT deptno, COUNT(*) AS dept_count
  FROM   emp
  GROUP BY deptno)
SELECT e.ename AS employee_name,
       dc1.dept_count AS emp_dept_count,
       m.ename AS manager_name,
       dc2.dept_count AS mgr_dept_count
FROM   emp e,
       dept_count dc1,
       emp m,
    dept_count dc2
WHERE  e.deptno = dc1.deptno
AND    e.mgr = m.empno
AND    m.deptno = dc2.deptno;
```

So we don't need to redefine the same subquery multiple times. Instead we just use the query name defined in the WITH clause, making the query much easier to read.

the WITH clause can simplify complex queries, like the following example that lists those departments with above average wages.

```
SQL>WITH
 dept_costs AS (
  SELECT dname, SUM(sal) dept_total
  FROM   emp e, dept d
  WHERE  e.deptno = d.deptno
  GROUP BY dname),
 avg_cost AS (
  SELECT SUM(dept_total)/COUNT(*) avg
  FROM   dept_costs)
SELECT *
FROM   dept_costs
WHERE  dept_total > (SELECT avg FROM avg_cost)
ORDER BY dname;
```

Subqueries follows SELECT clause:-

Subqueries can also follow SELECT clause. These subqueries returns one value per row.

Syntax :-

```
SELECT (SELCT statement) , -----FROM <tabname> ;
```

Example :-

Display no of records in EMP and DEPT table ?

```
SQL>SELECT (SELECT COUNT(*) FROM emp) AS EMP ,
                  (SELECT COUNT(*) FROM dept) AS DEPT
FROM DUAL;
```

```
EMP            DEPT
14             4
```

Display ename,sal,deptno,dept_totsal as follows ?

```
ENAME       SAL     DEPTNO       DEPT_TOTSAL
SMITH       800     20           9400
```

```
SQL>SELECT ename,sal,deptno,(SELECT SUM(sal) FROM emp
     WHERE deptno=x.deptno) AS dept_totsal
    FROM emp x ;
```

Database transaction

A database transaction is a group of SQL statements that perform a logical unit of work. Whose results should be made permanent in the database as a whole or undone as a whole .

An example of a database transaction is a transfer of money from one bank account to another. One UPDATE statement would subtract from the total amount of money from one account, and another

UPDATE would add money to the other account. Both the subtraction and the addition must be permanently recorded in the database, otherwise, money will be lost. If there is a problem with the money transfer, then the subtraction and addition must both be undone.

Starting and Ending a Transaction:

A transaction has a beginning and an end.

Transaction begins when one of the following events occurs:

→You connect to the database and perform a DML statement (an INSERT,UPDATE, OR DELETE).
→A previous transaction ends and you enter another DML statement.

A Transaction ends when one of the following events occurs:-

→You perform a COMMIT or a ROLLBACK.
→You perform a DDL statement then transaction ends with COMMIT.
→You perform a DCL statement, in which case a COMMIT is automatically performed.
→You disconnect from the database. If you exist SQL*Plus normally, by entering the EXIT command, a COMMIT is automatically performed for you. If SQL*Plus terminates abnormally- for example, if the computer on which SQL*Plus was running were to crash- a ROLLBACK is automatically performed.
→You perform a DML statement that fails, in which case a ROLLBACK is automatically performed for that individual DML statement.

Example :-

```
SQL>INSERT  transaction starts
SQL>UPDATE
SQL>COMMIT ;  transaction ends;
```

If transaction ends with COMMIT then it is called successful transaction and the operations are made permanent.

```
    SQL>INSERT;     transaction starts
    SQL>UPDATE;
    SQL>ROLLBACK; transaction ends
```

If transaction ends with ROLLBACK then it is called aborted transaction and operations are undone.

Savepoints:

You can also set a savepoint at any point with in a transaction. These allow you to roll back changes that savepoint. Savepoints can be useful to break up very long transactions, because, if you make a mistake after you've set a savepoint, you don't have to roll back the transaction all the way to the start.

Scenario:-

```
SQL>SELECT product_id, price
        FROM products
        WHERE product_id IN (4, 5);

PRODUCT_ID              PRICE
   4                  13.95
    5                   49.99
```

The price for product #4 is $13.95, and the price for product #5 is $49.99.

The following UPDATE increases the price of product #4 by 20 percent:

```
SQL>UPDATE products
                      SET price = price *1.20
                      WHERE product_id =4;
1 row updated.
```

The following statement sets a savepoint named save1:

```
SQL>SAVEPOINT save1;

Savepoint created.
```

Any DML statements run after this point can be rolled back to the savepoint, and the change made to product #4 will be kept.

The following UPDATE increases the price of product #5 by 30 percent:

```
SQL>UPDATE products
            SET price = price *1.30
              WHERE product_id = 5;
1 row updated.
SQL>ROLLBACK TO save1;
Rollback complete.
```

This has undone the price change for product #5, but left the price change for product #4 intact.

Every transaction has to support following four properties called ACID properties.

ACID Transaction Properties:

A transaction has four fundamental properties, known as ACID properties

Atomic: Transactions are atomic, meaning that the SQL statements contained in a transaction make up a single unit of work.

Consistent: Transactions ensure that the database state remains consistent, meaning that the database is in a consistent state when a transaction begins and that it ends in another consistent state when the transaction finishes.

Isolated: Separate transactions should not interfere with each other.

Durable: Once a transaction has been committed, the database changes are preserved, even if the machine on which the database software is running crashes later.

The Oracle database software handles these ACID properties and has extensive recovery facilities for restoring databases after system crashes.

Concurrent Transactions:

The Oracle database software supports many users interacting with a databse, and each user can run their own transaction at the same time. These transactions are known as concurrent transactions.If users are running transactions that affect the same table, the effects of those transactions are separated from each other until a COMMIT is performed. The following sequence of events, based on two transactions named T1 and T2 that access the customers table, illustrates the separation of transactions:
1. T1 and T2 perform a SELECT that retrieves all the rows from the customer table.
2. T1 performs an INSERT to add a row in the customers table, but T1 doesn’t perform a COMMIT.
3.T2 performs another SELECT and retrieves the same rows as those in step1. T2 doesn’t “see” the new row added by T1 in step2.
4. T1 finally performs a COMMIT to permanently record the new row added in step2.
5. T2 performs another SELECT and finally “sees” the new row added by T1.
To summarize: T2 doesn’t see the changes made by T1 until T1 commits its changes.

Transactions Locking:

To support concurrent transactions, the Oracle database software must ensure that the data in the tables remain valid. It does this through the use of locks. Consider the following example in which two transactions named T1 and T2 attempt to modify customer #1 in the customer table:
1. T1 performs an UPDATE to modify customer #1, but T1 doesn’t perform a COMMIT. T1 is said to have “locked” the row.
2. T2 also attempts to perform an UPDATE to modify customer #1, but since this row is already locked by T1,T2 is prevented from getting a lock on the row. T2’s UPDATE statement has to wait until T1 ends and frees the lock on the row.
3. T1 ends by performing a COMMIT, thus freeing the lock on the row.
4. T2 gets the lock on the row and the UPDATE is performed.T2 holds the lock on the row until T2 ends.
To summarize: A transaction cannot get a lock on a row while another transaction already holds the lock on that row.

Transaction 1 T1	Transaction 2 T2
(1) SELECT * FROM customers;	(2) SELECT * FROM customers;
(3) INSERT INTO customers (customers_id, first_name,last_name) VALUES(7, ‘Jason’, ‘Price’);	

(4) UPDATE customers
SET last_name = ‘Orange’
WHERE customer_id = 2;	

(5) SELECT * FROM customers; The returned result set contains the new row and the update.	(6)SELECT * FROM customers; The returned results set doesn’t contain the new row or the update made by T1. Instead, the result set Contains the original Rows retrieved in step2.
(7) COMMIT; This commits the new row and the update.	(8)SELECT * FROM customers: The returned result set contains the new row

and the update made by
T1 in step 3 and 4.

Transactions Isolation Levels:

The transaction isolation level is the degree to which the changes made by one transaction are separated from other transactions running concurrently. Before you see the various transaction isolation levels available, you need to understand the types of problems that may occur when current transactions attempt to access the same rows in a table.

In the following list, you'll see examples of two concurrent transactions named T1 and T2 that are accessing the same rows; listed are the three types of potential transaction processing problems.

Phantom reads:-

T1 reads a set of rows returned by a specified WHERE clause. T2 then inserts a new row, which also happens to satisfy the WHERE clause of the query previously used by T1. T1 then reads the rows again using the same query, but now sees the additional row just inserted by T2. This new row is known as a "phantom" because to T1 this row seems to have magically appeared.

Nonrepeatable reads:-

T1 reads a row, and T2 updates the same row just read by T1. T1 then reads the same row again and discovers that the row is read earlier is now different. This is known as a "non repeatable" read, because the row originally read by T1 has been changed.

Dirty reads:-

T1 updates a row, but doesn't commit the update T2 then reads the updated row.T1 then performs a rollback, Undoing the previous update. Now the row just read by T2 is no longer valid (it's "dirty") because the update made by T1 wasn't committed when the row was read by T2. To deal with these potential problems, database implement various levels of transaction isolation to prevent concurrent transactions from interfering with each other. The SQL standard defines the following transaction isolation levels,

READ UNCOMMITTED:- Phantom reads, nonrepeatable reads, and dirty reads are permitted.
READ COMMITTED:- Phantom reads and nonrepeatable reads are permitted, but dirty reads are not.

REPEATABLE READ:- Phantom reads are permitted, but Non repeatable and dirty reads are not.

SERIALIZABLE: Phantom reads, non repeatable reads, and dirty reads are not permitted.

The Oracle database software supports the READ COMMITTED and SERIALIZABLE transaction isolation levels. It doesn't support READ UNCOMMITTED or REPEATABLE READ levels.The default transaction isolation level defined by the SQL standard is SERIALIZABLE, but the default used by the Oracle database is READ COMMITTED, which is acceptable for nearly all applications.

OCA question:-

When does a transaction complete? (Choose all that apply.)
A. when a DELETE statement is executed
B. when a ROLLBACK command is executed
C. when a PL/SQL anonymous block is executed
D. when a data definition language (DDL) statement is executed

E. when a TRUNCATE statement is executed after the pending transaction

Database Security

Creating a user/schema/account in oracle DB :-

To create a user in the database, you use the CREATE USER statement. The simplified syntax for the CREATE USER statement is as follows:

Syntax :-

CREATE USER user_name IDENTIFIED BY password [DEFAULT TABLESPACE default_tablespace] [TEMPORARY TABLESPACE temporary_tablespace];

The following example creates a user called VIJAY with password ORACLE

```
SQL>CONNECT system/manager
SQL>CREATE USER VIJAY IDENTIFIED BY ORACLE;
```

The next example creates a user named VIJAY and specifies a default and temporary tablespace:

```
SQL>CREATE USER VIJAY IDENTIFIED BY Oracle
        DEFAULT TABLESPACE users
        TEMPORARY TABLESPACE temp;
```

To do operations over db the user must be granted with permissions

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

CONNECT-→TO CONNECT TO DB (CREATE SESSION)
TO CREATE BASIC OBJECTS
RESOURCE → TO CREATE OTHER OBJECTS

Changing a User's Password:-

You can change a user's password using the ALTER USER statement. For example, the following statement changes the password for oracle to oracle11g.

```
SQL>ALTER USER VIJAY IDENTIFIED BY ORACLE11G;
```

Droping user account :-

```
SQL>CONNECT system/manager
SQL>DROP USER VIJAY;
```

Privileges :-

Privileges means permissions

Privileges are two types

1 System privileges

2 Object privileges

System Privileges:-

A system privilege allows a user to perform certain actions within the database, such as executing DDL statements. For example, CREATE TABLE allows a user to create a table in their schema. Some of the commonly used system privileges are shown in the table:

Granting System Privileges to a user:-

You use GRANT to grant system privilege to a user. The following example grants some system privilege to vijay (assuming you're still connected to the database as system):

Example :-

SQL>GRANT CREATE SESSION, CREATE USER, CREATE TABLE TO VIJAY ;

Some of the system privileges are given below :-

SYSTEM PRIVILEGE	ALLOWS YOU TO……
CREATE SESSION	Connect to database.
CREATE SEQUENCE	Create a sequence
CREATE SYNONYM	Create a synonym.
CRETAE TABLE	Create a table in the user's schema.
CREATE ANY TABLE	Create a table in any schema.
DROP TABLE	Drop a table from the user's schema.
DROP ANY TABLE	Drop a table from any schema.
CREATE PROCEDURE	Create a stored procedure.
EXECUTE ANY PROCEDURE	Execute a procedure in any schema
CREATE USER	Create a user.
DROP USER	Drop a user.
CREATE VIEW	Create a view.

You can also use WITH ADMIN OPTION to allow a user to grant a privilege to another user. The following example grants the EXECUTE ANY PROCEDURE privilege with the ADMIN option to VIJAY.

SQL>GRANT EXECUTE ANY PROCEDURE TO VIJAY WITH ADMIN OPTION

EXECUTE ANY PROCEDURE can then be granted to another user by VIJAY. The following example connects as VIJAY and grants EXECUTE ANY PROCEDURE to KUMAR

SQL>CONNECT VIJAY/oracle11g
SQL>GRANT EXECUTE ANY PROCEDURE TO KUMAR;

You can grant a privilege to all users by granting to PUBLIC.

SQL>CONNECT system/manager
SQL>GRANT EXECUTE ANY PROCEDURE TO PUBLIC;

Every user in the database now has the EXECUTE ANY PROCEDURE privilege.

Checking System Privilege Granted to a User:-

You can check which system privileges a user has , by querying USER_SYS_PRIVS

Revoking System Privileges from a User:-

You revoke system privileges from a user using REVOKE. The following example connects as system and revokes the CREATE TABLE privilege from VIJAY.

SQL>CONNECT system/manager
SQL>REVOKE CREATE TABLE FROM VIJAY;

The next example revokes EXECUTE ANY PROCEDURE from VIJAY

SQL>REVOKE EXECUTE ANY PROCEDURE FROM VIJAY;

Object Privileges:-

An object privilege allows a user to perform certain actions on database objects, such as executing DML statements on tables. some of the commonly used objects privileges are shown in the table:

Object Privilege	Allows a user to…….
SELECT	Performs a select.
INSERT	Perform an insert.
UPDATE	Perform an update.
DELETE	Perform a delete.
EXECUTE	execute procedure

Granting Object Privileges to a User:-

You use GRANT to grant an object privilege to a user. The following example connects as SCOTT and grants the SELECT, INSERT, and UPDATE object privilege on the products table to VIJAY with the SELECT privilege on the employees table:

```
SQL>CONNECT SCOTT/TIGER;
SQL>GRANT SELECT, INSERT, UPDATE ON PRODUCTS TO VIJAY;

SQL>GRANT SELECT ON EMPLOYEES TO VIJAY;
```

he next example grants the UPDATE privilege on the last_name and salary columns to VIJAY

```
SQL>GRANT UPDATE (last_name, salary) ON EMPLOYEES TO VIJAY;
```

You can also use the GRANT option to enable a user to grant a privilege to another user. The following example grants the SELECT privilege on the customers table with the GRANT option to VIJAY

```
SQL>GRANT SELECT ON customers TO VIJAY WITH GRANT OPTOIN;
```

The SELECT ON customers privilege can then be granted to another user by VIJAY. The following example connects as VIJAY and grants this privilege to RAJU.

```
SQL>CONNECT vijay/oracle11g ;
SQL>GRANT SELECT ON customers TO RAJU.
```

Checking Object Privileges Made:-

You can check which table object privileges a user has made to other users by querying **USER_TAB_PRIVS_MADE**

Checking Object Privileges Retrieved:-

You can check which object privileges on a table a user has retrieved by querying the **USER_TAB_PRIVS_RECD** table.

Revoking Object Privileges:-

You revoke object privileges using REVOKE. The following example connects as SCOTT and revokes the INSERT privilege on the products table from VIJAY.

```
SQL>CONNECT SCOTT/TIGER;
SQL>REVOKE INSERT ON products FROM VIJAY;
```

The next example revokes the SELECT privilege on the customers table from VIJAY

```
SQL>REVOKE SELECT ON customers FROM VIJAY;
```

When you revoke SELECT ON customers from VIJAY who has already passed this privilege to RAJUl also loses the privilege.

Roles:-

→A role is a group of privileges that you can assign to a user or to another role. The following points summarize the benefits and features of roles:

→Rather than assigning privileges one at a time directly to a user, you can create a role, assign privileges to that role, and then grant that role to multiple users and roles.

→When you add or delete a privilege from a role, all users and roles assigned that role automatically receive or lose that privilege.

→You can assign multiple roles to a user or role.

→You can assign a password to a role.

→As you can see from these points, roles can help you manage multiple privileges assigned to multiple users.

```
SQL>CREATE ROLE TO SCOTT;
SQL>GRANT CREATE USER TO SCOTT WITH ADMIN OPTION;
```

You create a role using the CREATE ROLE statement. The following statements connects as store and create the three roles shown in table:

```
SQL>CONNECT scott/tiger
SQL>CREATE ROLE product_manager;
SQL>CREATE ROLE hr_manager;
SQL>CREATE ROLE overall_manager IDENTIFIED by manager_password;
```

Notice overall_manager has a password of manager_password.

Role Name	Has Permissions to…..
Product_manager	Perform SELECT, INSERT,UPDATE, and DELETE operations on the **product_types** and **products tables.**
hr_manager	perform SELECT,INSERT,UPDATE, and DELETE operations on the **salary_grades** and **employees** tables. Also, hr_manager is able to create users.
overall_manager	perform SELECT, INSERT,UPDATE, and DELETE operations on all the tables shown in the previous roles; overall_manager will be granted the previous roles.

Granting Privileges To Roles:-

You grant privileges to a role using the GRANT statement. You can grant both system and object privileges to a role as well as grant another role to role.

The following example grants the required privileges to the product_manager and hr_manager roles and grants these two roles to overall_manager:

```
SQL>GRANT SELECT, INSERT, UPDATE, DELETE ON product_types TO product_manager;
SQL>GRANT SELECT, INSERT, UPDATE, DELETE ON products TO product_manager;
SQL>GRANT SELECT, INSERT, UPDATE, DELETE ON salary_grades TO hr_manager;
SQL>GRANT SELECT, INSETR, UPDATE, DELETE ON employees TO hr_manager;
SQL>GRANT CREATE USER TO hr_manager;
SQL>GRANT product_manager, hr_manager TO overall_manager;
```

Granting Roles to a User:-

You grant a role to the user using GRANT. The following example grants the overall_manager role to VIJAY.

```
SQL>GRANT overall_manager TO VIJAY ;
```

Checking Roles Granted to a User:-

you can check which roles have been granted to a user querying USER_ROLE_PRIVS.

```
SQL>CONNECT  VIJAY/ORACLE11G ;
SQL>SELECT * FROM  user_role_privs;
```

Checking System Privileges Granted to a Role:-

You can check which system privileges have been granted to a role by querying **ROLE_SYS_PRIVS.**

```
SQL> SELECT * FROM role_sys_privs ;
```

Checking Object privileges Granted to a Role:-

You can check which object privileges have been granted to a role by querying **ROLE_TAB_PRIVS**

The following example queries role_tab_privs where rolE equals HR_MANAGER:

```
SQL>SELECT *FROM role_tab_privs
            WHERE role= 'HR_MANAGER'
           ORDER BY table_name;
```

Revoking a Role:-

You revoke a role using REVOKE. The following example revokes the overall_manager role from VIJAY

```
SQL>REVOKE overall_manager FROM VIJAY;
```

Revoking Privileges from a Role:-

You revoke a privilege from a role using REVOKE.

```
SQL>REVOKE ALL ON products FROM product_manager;
SQL>REVOKE ALL ON product_types FROM product_manager;
```

Dropping a Role:-

You drop a role using DROP ROLE.

```
SQL>DROP ROLE overall_manager;
SQL>DROP ROLE product_manager;
SQL>DROP ROLE hr_manager;
```

Schema Objects

- ➔ TABLES
- ➔ SEQUENCES
- ➔ VIEWS
- ➔ MATERIALIZED VIEWS
- ➔ SYNONYMS
- ➔ TYPES
- ➔ INDEXES
- ➔ CLUSTERS
- ➔ PROCEDURES
- ➔ FUNCTIONS
- ➔ PACKAGE
- ➔ DB TRIGGER

SEQUENCE

→A SEQUENCE is a schema object that can generate unique sequential values.
→The SEQUENCE Values are often used for PRIMARY KEY'S and UNIQUE KEY'S.

CREATING SEQUENCES:-

Syntax:-

```
CREATE SEQUENCE sequenceName
[INCREMENT BY n]
[START WITH n]
[MAXVALUE n|NOMAXVALUE]
[MINVALUE n|NOMINVALUE]
[CYCLE |NOCYCLE]
[CACHE n|NOCACHE]
ORDER/NORDER;
```

Increment By :-

→Specifies the interval between the Sequence Numbers.
→Value can be Positive or Negative, but can not be 0.
→If the value is positive it is Incremental Sequence else it Decremental Sequence.

Minvalue:-

Specifies the sequence's Minimum Value.

NoMinvalue

Specifies a minimum value of 1 for an ascending sequence and –(10)^26) for a descending sequence.

Maxvalue:-

Specifies the maximum value that sequence can generate.

NoMaxvalue:-

Specifies a maximum value of 10^27 for an ascending sequence and -1 for a descending sequence.

Start With:-

→Specifies the first sequence number to be generated.
→For ascending sequence the default value is SEQUENCES'S MINIMUM value.

Cycle:-

Specifies whether the sequence contains to generate values after reaching its maximum or minimum value.

NoCycle:-

Specifies the SEQUENCE cannot general more values after the targeted limit.

Cache:-

Specifies how many values the Oracle Server Preallocates and keep in memory.

NoCache:-

Specifies the values of a SEQUENCE are not preallocated.
If the above parameters are not specified by default 20 values are cached.

Order:-

Gurantee the sequence numbers to be generated in the order of request.

NoOrder:-

→Does not guranteee the sequence Number to be generated in order.
If the above parameters are not specified by default

START WITH Will be 1.
INCREMENT BY Will be positive 1.
SEQUENCE is NOCYCLE.
The CACHE Value Will be 20
SEQUENCE is ORDER.

Test Table:-

```
SQL>CREATE TABLE customer
        (cid NUMBER(2)  constraint pk_customer PRIMARY KEY,
        Cname VARCHAR2(20) ,
        Address VARCHAR2(20));
```

Creation of Incremental Sequence :-

```
SQL>CREATE SEQUENCE SEQ1
         MINVALUE 1
         INCREMENT BY 1
         MAXVALUE 9999
         NOCACHE
         NOCYCLE;
```

USING SEQUENCE GENERATE VALUE :-

Every sequence has two psuedo columns

1 CURRVAL...........it is showing last value which value is inserted that value showing
2 NEXTVAL.......... it is show showing next value from the inserted last value

→CURRVAL returns current value of the sequence .
→NEXTVAL returns next value of the sequence.
Both values are accessed by using sequence name as follows

```
SEQ1.CURRVAL
SEQ1.NEXTVAL
```

Ofter created the sequence then user sequence name below as
===

Example :-

```
SQL>INSERT INTO customers
             VALUES(SEQ1.Nextval,'sachin','mumbai');
```

Creating A Sequence with CYCLE:-

```
SQL>CREATE SEQUENCE SEQ2
                  INCREMENT BY 10
                  START WITH 10
                  MINVALUE 0
                  MAXVALUE 9999
                  NOCACHE
                  CYCLE;
```

How can see the currval and nextval

SELECT SS.CURRVAL FROM DUAL; … using for showing currval last inserted value. Here, use dual table name .

SELECT SS.NEXTVAL FROM DUAL; …using for showing nextval which value need to insert next from the current value that value showing , here also should use same dual table while we was showing the curval and nextval

Creation of Decremental Sequence:-

```
SQL>CREATE SEQUENCE SEQ3
                  INCREMENT BY-1
                  START WITH 10
                  MINVALUE 0
                  MAXVALUE 10
                  NOCACHE
                  NOCYCLE;
```

Modifying a Sequence:-

→The ALTER command can be used to change the present status of a SEQUENCE.

→The ALTER SEQUENCE command can be used to change

Increment Value
Maximum Value
Minimum Value
Cycle Option
Cache Option

Syntax:-

```
ALTER SEQUENCE sequence
[INCREMENT BY n]
[{MAXVALUE n| NOMAXVALUE}]
[{MINVALUE n| NOMINVALUE}]
[{CYCLE | NOCYCLE}]
[{CACHE n |NOCACHE}];
```

Example :-

SQL>ALTER SEQUENCE SEQ1 MAXVALUE 500

Guidelines for Modifying a Sequence:-

→You must be the owner or have the ALTER privilege for the sequence modify it.
→Only future sequence numbers are affected by the ALTER SEQUENCE statement.
→The START WITH option cannot be changed using ALTER SEQUENCE.
→The sequence must be dropped and re-created in order to restart the sequence at a different number.

Viewing the Current value of a Sequence:-

SQL>Select SEQ1.Currval from Dual;

Removing a Sequence:-

SQL>DROP SEQUENCE SEQ1 ;

Retrieving SEQUENCE information :-

USER_SEQUENCES
ALL_SEQUENCES
DBA_SEQUENCES

The setting of the SEQUENCE can be confirmed by selecting on USER_SEQUENCES catalog.

```
SQL>SELECT sequence_name,min_value,max_value,increment_by,last_number
        FROM user_Sequences
        WHERE sequence_Name= 'SEQ1';
```

OCA question :-

1 Which two statements are true about sequences created in a single instance database? (Choose two.)

A. The numbers generated by a sequence can be used only for one table.
B. DELETE <sequencename> would remove a sequence from the database.
C. CURRVAL is used to refer to the last sequence number that has been generated.
D. When the MAXVALUE limit for a sequence is reached, you can increase the MAXVALUE limit by using the ALTER SEQUENCE statement.
E. When a database instance shuts down abnormally, the sequence numbers that have been cached but not used would be available once again when the database instance is restarted.

2 SQL>CREATE TABLE ord_items

```
(ord_no NUMBER(4) DEFAULT ord_seq.NEXTVAL NOT NULL,
item_no NUMBER(3), qty NUMBER(3) CHECK (qty BETWEEN 100 AND 200),
expiry_date date CHECK (expiry_date > SYSDATE),
CONSTRAINT it_pk PRIMARY KEY (ord_no,item_no),
CONSTRAINT ord_fk FOREIGN KEY(ord_no) REFERENCES orders(ord_no));
```

The command to create a table fails. Identify the reason for the SQL statement failure? (Choose all that apply.)

A. You cannot use SYSDATE in the condition of a CHECK constraint.

B. You cannot use the BETWEEN clause in the condition of a CHECK constraint.

C. You cannot use the NEXTVAL sequence value as a DEFAULT value for a column.

D. You cannot use ORD_NO and ITEM_NO columns as a composite primary key because ORD_NO is also the FOREIGN KEY.

3 Evaluate the following CREATE SEQUENCE statement:

```
CREATE SEQUENCE seq1
START WITH 100
INCREMENT BY 10
MAXVALUE 200
CYCLE
NOCACHE;
```

The SEQ1 sequence has generated numbers up to the maximum limit of 200. You issue the following SQL stmt :-

```
SELECT seq1.nextval FROM dual;
```

What is displayed by the SELECT statement?

A. 1

B. 10

C. 100

D. an error

VIEWS

Data abstraction is usually required after a table is created and populated with data. Data held by some tables might require restricted access to prevent all users from accessing all columns of a table, for data security reasons. Such a security issue can be solved by creating several tables with appropriate columns and assigning specific users to each such table, as required. This answers data security requirements very well but gives rise to a great deal of redundant data being resident in tables, in the database.To reduce redundant data to the minimum possible, Oracle provides Virtual tables which are Views.

View Definition :-

→A View is a virtual table based on the result returned by a SELECT query.

→The most basic purpose of a view is restricting access to specific column/rows from a table thus allowing different users to see only certain rows or columns of a table.

Composition Of View:-

→A view is composed of rows and columns, very similar to table. The fields in a view are fields from one or more database tables in the database.

→SQL functions, WHERE clauses and JOIN statements can be applied to a view in the same manner as they are applied to a table.

View storage:-

→Oracle does not store the view data. It recreates the data, using the view's SELECT statement, every time a user queries a view.

→A view is stored only as a definition in Oracle's system catalog.

→When a reference is made to a view, its definition is scanned, the base table is opened and the view is created on top of the base table.This, therefore, means that a view never holds data, until a specific call to the view is made. This reduces redundant data on the HDD to a very large extent.

Advantages Of View:-

Security:- Each user can be given permission to access only a set of views that contain specific data.

Query simplicity:- A view can drawn from several different tables and present it as a single table turning multiple table queries into single table queries against the view.

Data Integrity:- If data is accessed and entered through a view, the DBMS can automatically check the data to ensure that it meets specified integrity constraints.

Disadvantage of View:-

Performance:- Views only create the appearance of the table but the RDBMS must still translate queries against the views into the queries against the underlined source tables. If the view is defined on a complex multiple table query then even a simple query against the view becomes a complicated join and takes a long time to execute.

Types of Views :-

- **Simple Views**
- **Complex Views**

Simple Views :-

a View based on single table is called simple view.

Syntax:-

```
CREATE VIEW <View Name>
AS
SELECT<ColumnName1>,<ColumnName2>
FROM <TableName>
[WHERE <COND>]
[WITH CHECK OPTION]
[WITH READ ONLY]
```

Example :-

```
SQL>CREATE VIEW emp_v
        AS
        SELECT empno,ename,sal FROM emp ;
```

→Views can also be used for manipulating the data that is available in the base tables[i.e. the user can perform the Insert, Update and Delete operations through view.

→Views on which data manipulation can be done are called Updateable Views.
→If an Insert, Update or Delete SQL statement is fired on a view, modifications to data in the view are passed to the underlying base table.
→For a view to be updatable,it should meet the following criteria:
→Views defined from Single table.
→If the user wants to INSERT records with the help of a view, then the PRIMARY KEY column(s) and all the NOT NULL columns must be included in the view.

Inserting record through view :-

```
SQL>INSERT INTO emp_v VALUES(1,'A',5000,200) ;
```

Updating record throught view :-

```
SQL>UPDATE emp_v SET sal=2000 WHERE empno=1;
```

Deleting record throught view :-

```
SQL>DELETE FROM emp_v WHERE empno=1;
```

With Check Option :-

If VIEW created with WITH CHECK OPTION then any DML operation through that view violates where condition then that DML operation returns error.

Example :-

```
SQL>CREATE VIEW V2
        AS
        SELECT empno,ename,sal,deptno FROM emp
        WHERE deptno=10
        WITH CHECK OPTION ;
```

Then insert the record into emp table through view V2

```
SQL>INSERT INTO V2 VALUES(2323,'RAJU',4000,20) ;
```

The above INSERT returns error because DML operation violating WHERE clause.

Complex Views :-

A view is said to complex view

→If it based on more than one table
→Query contains
AGGREGATE functions
DISTINCT clause
GROUP BY clause

HAVING clause
Sub-queries
Constants
Strings or Values Expressions
UNION,INTERSECT,MINUS operators.

Example 1 :-

```
SQL>CREATE VIEW V3
AS
SELECT E.empno,E.ename,E.sal,D.dname,D.loc
FROM emp E JOIN dept D
USING(deptno) ;
```

Complex views are not updatable i.e. we cannot perform insert or update or delete operations on base table through complex views.

Example 2 :-

```
SQL>CREATE VIEW V2
AS
SELECT deptno,SUM(sal) AS sumsal
FROM EMP
GROUP BY deptno;
```

Destroying a View:-

The DROP VIEW command is used to destroy a view from the database.

Syntax:-

```
DROP VIEW<viewName>
```

Example :-

```
SQL>DROP VIEW emp_v;
```

Querying VIEWS information :-

USER_VIEWS
ALL_VIEWS
DBA_VIEWS

OCA questions :-

1. Which two statements are true regarding views? (Choose two.)

A.A subquery that defines a view cannot include the GROUP BY clause.
B.A view that is created with the subquery having the DISTINCT keyword can be updated.
C.A view that is created with the subquery having the pseudo column ROWNUM keyword cannot be updated.
D.A data manipulation language (DML) operation can be performed on a view that is created with the subquery having all the NOT NULL columns of a table.

2 You want to create a SALE_PROD view by executing the following SQL statement:

```
CREATE VIEW sale_prod
AS SELECT p.prod_id, cust_id, SUM(quantity_sold) "Quantity", SUM(prod_list_price) "Price"
FROM products p, sales s
WHERE p.prod_id=s.prod_id
GROUP BY p.prod_id, cust_id;
```

Which statement is true regarding the execution of the above statement?

A. The view will be created and you can perform DML operations on the view.

B. The view will be created but no DML operations will be allowed on the view.

C. The view will not be created because the join statements are not allowed for creating a view.

D. The view will not be created because the GROUP BY clause is not allowed for creating a view.

3 Evaluate the following command:

```
CREATE TABLE employees
(employee_id NUMBER(2) PRIMARY KEY,
last_name VARCHAR2(25) NOT NULL,
department_id NUMBER(2)NOT NULL,
job_id VARCHAR2(8),salary NUMBER(10,2));
```

You issue the following command to create a view that displays the IDs and last names of the sales staff in the organization:

```
CREATE OR REPLACE VIEW sales_staff_vu
AS
SELECT employee_id,last_name,job_id
FROM employees
WHERE job_id LIKE 'SA_%'
WITH CHECK OPTION;
```

Which two statements are true regarding the above view? (Choose two.)

A. It allows you to insert rows into the EMPLOYEES table.

B. It allows you to delete details of the existing sales staff from the EMPLOYEES table.

C. It allows you to update job IDs of the existing sales staff to any other job ID in the EMPLOYEES table.

D. It allows you to insert IDs, last names, and job IDs of the sales staff from the view if it is used in multitable INSERT statements.

Synonyms

A synonym is another name or alternative name for a table.

Synonyms are created

- ➔ If tablename if lengthy.
- ➔ To provide local transparency for remote objects.

Syntax:-

```
CREATE SYNONYM <NAME> FOR <TABLENAME>;
```

Example :-

```
SQL>CREATE SYNONYM EMP FOR EMPLOYEE_INFORMATION;
```

After creating synonym , now EMPLOYEE_INFORMATION table can be accessed by using name EMP. For Example

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

NOTE:- difference between synonym and table alias is , the scope of the table alias is upto that query only but synonym can be used any query.

VIEWS vs SYNONYMS:-

VIEWS	SYNONYMS
Subset of a table	mirror of a table
Can be based on more than one table	based on only one table

Public Synonyms:-

You can also create a public synonym for a table. When you do this , all users can see the synonym.

```
SQL>CONNECT system/manager
SQL>GRANT CREATE PUBLIC SYNONYM TO scott;
SQL>CONNECT scott/tiger
SQL>CREATE PUBLIC SYNONYM products FOR SCOTT.products;
```

If you connect as vijay, who has the SELECT privilege on SCOTT.products, you can now retrieve rows from SCOTT.products through the products public synonym:

```
SQL>CONNECT vijay/vijay;
SQL>SELECT * FROM products;
```

Retrieving Synonyms Information :-

USER_SYNONYMS , ALL_SYNONYMS , DBA_SYNONYMS

OCA question :-

1 The ORDERS table belongs to the user OE. OE has granted the SELECT privilege on the ORDERS table to the user HR. Which statement would create a synonym ORD so that HR can execute the following query successfully?

SELECT * FROM ord;

A. CREATE SYNONYM ord FOR orders; This command is issued by OE.
B. CREATE PUBLIC SYNONYM ord FOR orders; This command is issued by OE.
C. CREATE SYNONYM ord FOR oe.orders; This command is issued by the database administrator.
D. CREATE PUBLIC SYNONYM ord FOR oe.orders; This command is issued by the database administrator.

2 Which statement is true regarding synonyms?

A. Synonyms can be created only for a table.
B. Synonyms are used to reference only those tables that are owned by another user.
C. A public synonym and a private synonym can exist with the same name for the same table.
D. The DROP SYNONYM statement removes the synonym, and the table on which the synonym has been created becomes invalid.

Indexes

When looking for a particular topic in a book, you either scan the whole book, or you can use the index to find the location. An index for a database table is similar in concept to a book index, except that database indexes are used to find specific rows in a table.

Oracle server uses following methods to locate the desired information

- Table Scan
- Index Scan

Table Scan :-

In table scan oracle scans the entire table to locate the desired information.

Index Scan :-

In index scan oracle uses index to locate the place that holds the required data and then jumps to that place to get required data. This is much faster than table scan.

How Does Indexing Work :-

- When an index is created on a table, oracle internally forms a two dimensional matrix that contains Data extracted from the column on which index is created and Physical Address of the record (rowid) .

- When an SQL query that has a WHERE clause based on the column on which index is fired , oracle finds the value in index and locates the record in the table using ROWID .

TYPES OF INDEXES :-

➔ B-tree Indexes
  - Simple Index
  - Composite Index
  - Unique Index
  - Function based Index

➔ Bitmap Indexes

B-TREE INDEXES :-

When to use BTREE Indexes:-

1 if table contains huge amount of data
2 when a query retrieves <=10 percent of the total rows in a table.
3 if column for the index contains a wide range of values.
4 A good candidate for B-tree indexing would be a column containing a unique value for each row
5 on the common columns that used in JOIN operation.

Simple B-tree Index:-

If index is created on single column then it is called simple index.

Syntax:-

CREATE [UNIQUE] INDEX index_name ON table_ name (column_name)
SQL> CREATE INDEX I1 ON EMP(sal)
When ORACLE Iinvokes Index:-

→A SELECT query with a WHERE clause spceicifed on column on which index is created

Example :-SQL>SELECT * FROM EMP WHERE sal > 2000 ;

→A SELECT query with ORDER BY clause specified on column on which index is created

When ORACLE doesn't Invoke Index:-

→ SELECT query is fired without a WHERE clause

Example :- SQL>SELECT * FROM EMP ;

→ SELECT query is fired with WHERE clause specified on the column on which index is not defined

Example:- SQL>SELECT * FROM EMP WHERE ename='SMITH';

→ SELECT query is fired with ORDER BY clause specified on the column on which index is not defined.

→ SELECT query is fired with a WHERE clause with a != condition.

Example:- SQL>SELECT * FROM EMP WHERE sal <> 5000 ;
→SELECT query is fired with IS NULL or IS NOT NULL operators
Composite Index :-

If an index is created on multiple columns then it is called composite index.
Example :-

SQL>CREATE INDEX I2 ON EMP(deptno,job) ;

Oracle server uses above index when SELECT query with WHERE clause is based on leading column of index is fired.

Example :-

SQL>SELECT *FROM EMP WHERE DEPTNO=10 ; (index scan)
SQL>SELECT * FROM EMP WHERE DEPTNO=10 AND JOB='CLERK'; (index scan)
SQL>SELECT * FROM EMP WHERE JOB='CLERK'; (table scan)
Unique Index :-

UNIQUE index doesn't allow duplicate values into the column on which INDEX is created.
Example :-

SQL> CREATE UNIQUE INDEX I3 ON DEPT(dname);

NOTE:-

PRIMARY KEY columns and UNIQUE columns are automatically indexed by ORACLE.

Function Based Index :-

There are times when even though an index exists, oracle doesn't use it and instead follows table scan. This is usually happens when index created on a column, but the SQL query reference that column with a function or arithmetic expression.

For example , an index is created on the City column of the Customers table and the following query is fired to retrieve all those row who belong to MUMBAI.

```
SQL> SELECT * FROM CUSTOMER WHERE UPPER(city)='MUMBAI';
```

Above query reference the City column along with UPPER function and hence oracle doesn't use the index. To overcome such issue, oracle provides function based index

```
SQL>CREATE INDEX I4 ON CUSTOMER (UPPER(CITY));
```

In addition, the database administrator must set the initialization parameter QUERY_REWRITE_ENABLED to true (the default is false) in order to take advantage of function – based indexes.

The following query example sets QUERY_REWRITE_ENABLED to true:

```
SQL>CONNECT system/manager
SQL>ALTER SYSTEM SET QUERY_REWRITE_ENABLED=TRUE;
```

Reverse Index:-

In the index leaf block , Oracle stores the index key value and ROWID.

Assume that there is an unique index on custid , suppose 3 individuals concurrently hit the database to insert rows with customer numbers 101,102,103 then index entries are stored in same leaft block , which causes buffer busy waits.

If the index is reverse unique index then the entries will be stored in different leaft block

Example :-

```
SQL>CREATE INDEX I7 ON CUSTOMER(CUSTID) REVERSE ;
```

NOTE :-

A normal index cannot be rebuild as a reverse key index.

BITMAP INDEXES :-

- ➔ Bitmap indexes are typically used in Data Warehouse.
- ➔ Bitmap indexes are created on low cardinality columns.
- ➔ Bitmap indexes are useful when data is not modified by concurrent transactions.
- ➔ Bitmap is used for each key value, the bitmap enables the database to locate a row.
- ➔ A mapping function converts bitmaps to rowids

Example :-

```
SQL> CREATE BITMAP INDEX  BI6 ON EMP(JOB);
```

Modifying an Index:-

ALTER command is used to Modify an INDEX.

Example :-

```
SQL>ALTER  INDEX I1 RENAME TO I10;
```

DROPPING INDEX :-

SYNTAX :-

SQL>DROP INDEX <NAME> ;

Example:-

SQL>DROP INDEX I1;

Retrieving index information :-

USER_INDEXES
USER_IND_COLUMNS
ALL_INDEXES
DBA_INDEXES

OCA question :-

Which statements are correct regarding indexes? (Choose all that apply.)
When a table is dropped, the corresponding indexes are automatically dropped.

A. A FOREIGN KEY constraint on a column in a table automatically creates a nonunique index.
B. A nondeferrable PRIMARY KEY or UNIQUE KEY constraint in a table automatically creates a unique index.
C. For each data manipulation language (DML) operation performed, the corresponding indexes are automatically updated.

Clusters

A cluster is a data structure that improves retrieval performance. A cluster, like an index, does not affect the logical view of the table. A cluster is a way of storing related data values together on disk. Oracle reads data a block at a time, so storing related values together reduces the number of I/O operations needed to retrieve related values, since a single data block will contain only related rows.

A cluster is composed of one or more tables. The cluster includes a cluster index, which stores all the values for the corresponding cluster key. Each value in the cluster index points to a data block that contains only rows with the same value for the cluster key.

If a cluster contains multiple tables, the tables should be joined together and the cluster index should contain the values that form the basis of the join. Because the value of the cluster key controls the placement of the rows that relate to the key, changing a value in that key can cause Oracle to change the location of rows associated with that key value.

Clusters may not be appropriate for tables that regularly require full table scans, in which a query requires the Oracle database to iterate through all the rows of the table. Because you access a cluster table through the cluster index, which then points to a data block, full table scans on clustered tables can actually require more I/O operations, lowering overall performance.

Creating a Cluster: Example

The following statement creates a cluster named personnel with the cluster key column department, a cluster size of 512 bytes, and storage parameter values:

```
SQL>CREATE CLUSTER personnel
                  (department NUMBER(4))
SIZE 512
STORAGE (initial 100K next 50K);
```

Cluster Keys: Example

The following statement creates the cluster index on the cluster key of personnel:

```
SQL>CREATE INDEX idx_personnel ON CLUSTER personnel;
```

After creating the cluster index, you can add tables to the index and perform DML operations on those tables.

Adding Tables to a Cluster: Example

The following statements create some tables and add them to the personnel cluster created in the earlier example:

```
SQL>CREATE TABLE dept
         (deptno  NUMBER(2) PRIMARY KEY,
         dname VARCHAR2(20),
         loc       VARCHAR2(20))
   CLUSTER personnel (deptno);

SQL>CREATE TABLE emp
         (empno NUMBER(4),
          ename VARCHAR2(20),
          sal       NUMBER(7,2),
         dno      NUMBER(2) references dept(deptno))
   CLUSTER personnel (department_id);
```

Retrieving Clusters Information :-

USER_CLUSTERS
ALL_CLUSTERS
DBA_CLUSTERS

Materialized Views

→A materialized view is a database object that contains the results of a query unlike normal views that only contains the query definition and not the results.

→Materialized views are usually a choice when creating summary tables on aggregate of a table's data.

→A materialized view can query tables, view's and other materialized views.

→A materialized takes a different approach in which the query result is cached as a concrete table that may be updated from the original base tables from time to time.

→ Materialized view does not contain up-to-the–minute information. When an ordinary view is queried, the data retrieved includes changes made up to the last committed transaction.However, when an materialized view is queried the data retrieved would be at a state when the view was created or last refreshed.

Syntax:-

```
CREATE MATERIAIZED VIEW <schema.Name>
REFRESH [FAST|COMPLETE|FORCE][ON DEMAND|COMMIT]
START WITH DATE
NEXT DATE
WITH [PRIMARY KEY|RowID]
```

REFRESH:-

Since the materialized view is built on underlying data that is periodically changed, specify how and when to refresh the data in the view. The following keywords in the REFRESH clause can also be used to create a schedule for recurring refresh operations.

FAST: - Updates only the values in the materialized view, assuming that some preconditions are met.

COMPLETE:- Recreates the view completely.

FORCE: - Does a FAST refresh if possible and a COMPLETE refresh if the preconditions for a FAST refresh are not available.

ON COMMIT:- Causes a refresh to occur whenever the underlying data is changed and the changes are committed.

ON DEMAND:- Performs a refresh when it is scheduled or explicitly called.

START WITH DATE:- Indicates the date and interval at which the materialized view is to be refreshed

NEXT DATE:- Indicates the time and interval at which the materialized view is to be refreshed next

WITH PRIMARY KEY: - Indicates whether the materialized view is based on Primary Key

WITH ROWID:- Indicates whether the materialized view is based on RowID.

Example :-

```
SQL>CREATE MATERIALIZED VIEW mv1
        REFRESH ON COMMIT
        START WITH SYSDATE   NEXT SYSDATE+7
        AS
        SELECT deptno,SUM(sal) SUMSAL FROM emp
        GROUP BY deptno ;
```

In the above example materialized view is automatically refreshed for every one week.

RowID Materialized Views:-

RowID materialized views should have a single master table and cannot contain any of the following:

Distinct or aggregate functions
GROUP BY
Subqueries
Joins and Set operations

The following statement creates the RowID materialized view on table Empoyees:

Example:-

```
SQL>CREATE MATERIALIZED VIEW mv_rowid_emp
       REFRESH ON COMMIT
        WITH RowID
        AS
       SELECT *FROM emp ;
```

Retrieving Materialized View Information :-

USER_MVIEWS
ALL_MVIEWS
DBA_MVIEWS

Advanced Features:-

Table Partitioning

→As the number of rows in your table grows, the management and performance impacts will increase. Backups will take longer, recoveries will take longer and queries on that will take longer.
→Administrative and performance issues can be simplified by seperating rows of a single table into multiple parts.
→Dividing a table's data in this manner is called partitioning the table, and table is called partitioned table and parts are called partitions.
→Partitioning is useful for very large tables.
→Partitioning is based on particular column , the column on which table is partitioned is called partition key.

Advantages :-

- ➔ The performance of queries against the table may improve.
- ➔ The tables may be easier to manage.
- ➔ Backup and recover operations may perform better.
- ➔ Improves availability.

a table can be partitioned in different ways

1. **RANGE PARTITION**
2. **LIST PARTITION**
3. **HASH PARTITION**

RANGE partition :-

Which records are assigned to which partition depends on range of the partition key.

Example :-

```
SQL>CREATE TABLE emp_range
            (empno      NUMBER(4) ,
             ename      VARCHAR2(20) ,
             sal        NUMBER(7,2))
            PARTITION BY RANGE(sal)
            (
            PARTITION P1 VALUES LESS THAN(2000) ,
            PARTITION P2 VALUES LESS THAN(4000),
            PARTITION P3 VALUES LESS THAN(MAXVALUE)
            ) ;
```

→Employee whose salaries less than 2000 all those records are assinged to partition P1.
→Employee whose salaries less than 4000 all those records are assinged to partition P2 .
→MAXVALUE is a keyword , any data that could not be stored in earlier partitions are assigned to partition P3.

INSERTING RECORDS INTO PARTITIONED TABLE :-

SQL>INSERT INTO emp_range VALUES(1,'A',1500) ; (inserted into partition P1)
SQL>INSERT INTO emp_range VALUES(2,'B',3500) ; (inseted into partition P2)
SQL>INSERT INTO emp_range VALUES(3,'C',5000) ; (inserted into partition P3)
To view records assigned to particular partition , execute the following command
SQL>SELECT * FROM emp_range partition (p1) ;
The above query displays only the records assigned to partition P1.

Managing Partitions:-

Partitions can be dropped , new partitions can be added , and two partitions can be merged.

Droping Partition:-

```
SQL>ALTER TABLE EMP_RANGE DROP PARTITION P3;
```

When partition is dropped then records assigned to that partition are also dropped.

Adding New Partition:-

```
SQL>ALTER TABLE EMP_RANGE ADD PARTITION P3 VALUES LESS THAN(6000) ;
```

Merging Two Partitions :-

```
SQL>ALTER TABLE EMP_RANGE
          MERGE PARTITION P2,PARTITION P3 INTO PARTITION P3;
```

LIST partition:-

Which record is assinged to which partition depends on value of partition key value.

```
SQL>CREATE TABLE emp_list
         (empno      NUMBER(4) ,
          ename      VARCHAR2(20),
          job        VARCHAR2(20)
         )
         PARTITION BY LIST(job)
         (
         PARTITION P1 VALUES (‘CLERK’) ,
         PARTITION  P2 VALUES(‘MANAGER’) ,
         PARTITION P3 VALUES (DEFAULT)
         ) ;
```

HASH Partition:-

A Hash paritition determines the physical placement of data by applying hash function on partition key .

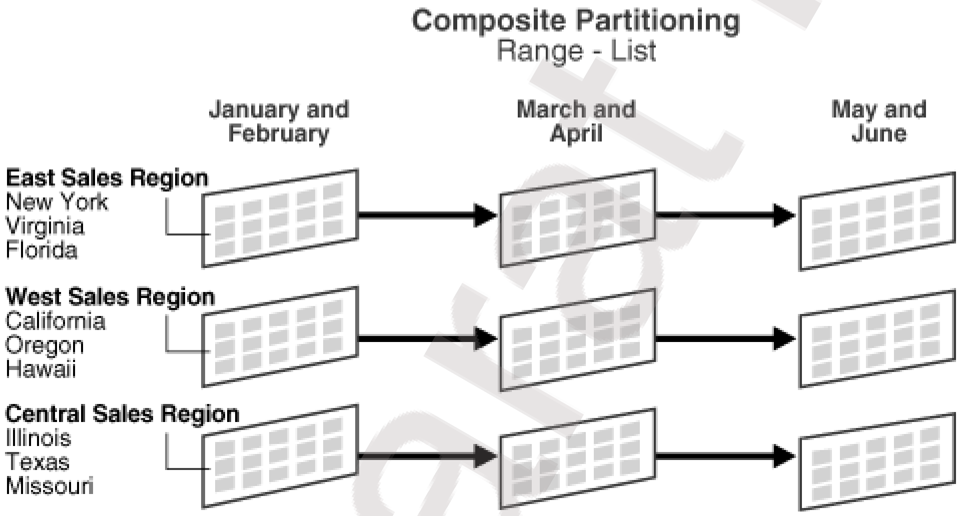
```
SQL>CREATE TABLE emp_hash
         (empno      NUMBER(4) ,
          ename      VARCHAR2(20) ,
          sal        NUMBER(7,2) ,
         deptno      NUMBER(2)
         )
         PARTITION BY HASH(deptno)
         PARTITIONS 4;
```

Creating SUBPARTITIONS or COMPOSITE Partitions :-

We can create subpartitions that is, paritions of partitions. You can use subpartitions to combine all types of partitions like range partitions, hash partitions,list partitions.

Composite Partitioning Range-List Example

```
CREATE TABLE bimonthly_regional_sales
( item_no VARCHAR2(20),
 txn_date DATE,
 txn_amount NUMBER,
 state VARCHAR2(2))
PARTITION BY RANGE (txn_date)
SUBPARTITION BY LIST (state)
SUBPARTITION TEMPLATE(
                SUBPARTITION east VALUES('NY', 'VA', 'FL'),
                SUBPARTITION west VALUES('CA', 'OR', 'HI'),
                SUBPARTITION central VALUES('IL', 'TX', 'MO'))
                (
PARTITION janfeb_2000 VALUES LESS THAN (TO_DATE('1-MAR-2000','DD-MON-YYYY')),
PARTITION marapr_2000 VALUES LESS THAN (TO_DATE('1-MAY-2000','DD-MON-YYYY')),
PARTITION mayjun_2000 VALUES LESS THAN (TO_DATE('1-JUL-2000','DD-MON-YYYY'))
);
```



Composite partitioning Range-Hash Example:-

```
SQL>CREATE TABLE emp_range_hash
        (empno      NUMBER(4) ,
        Ename      VARCHAR2(20) ,
        sal        NUMBER(7,2) ,
        deptno      NUMBER(2)
        )
        PARTITION BY RANGE(sal)
        SUBPARTITION BY HASH(deptno)
```

```
SUBPARTITIONS 4
(PARTITION P1 VALUES LESS THAN(2000) ,
PARTITION P2 VALUES LESS THAN(4000) ,
PARTITION P3 VALUES LESS THAN(MAXVALUE)
) ;
```

Retrieving PARTITIONS information:-

USER_TAB_PARTITIONS :- maintains information about partitions.

USER_TABLES:- to know whether table is partitioned or not

Display whether EMP table is partitioned or not ?

```
SQL>SELECT partitioned FROM user_tables WHERE table_name='EMP' ;
NO
```

Object Relational Features

Oracle Objects :-

Oracle object types are user-defined types that make it possible to model real-world entities.

New object types can be created from any built-in database types and any previously created object types, object references, and collection types. Metadata for user-defined types is stored in a schema that is available to SQL, PL/SQL, Java.

Object types and related object-oriented features such as variable-length arrays and nested tables provide higher-level ways to organize and access data in the database. Underneath the object layer, data is still stored in columns and tables, but you are able to work with the data in terms of the real-world entities, such as customers and purchase orders, that make the data meaningful. Instead of thinking in terms of columns and tables when you query the database, you can simply select a customer.

Internally, statements about objects are still basically statements about relational tables and columns, and you can continue to work with relational data types and store data in relational tables as before. But now you have the option to take advantage of object-oriented features too. You can begin to use object-oriented features while continuing to work with most of your data relationally, or you can go over to an object-oriented approach entirely. For instance, you can define some object data types and store the objects in columns in relational tables, which enables you to extend the system built-in types with user-defined ones. You can also create object views of existing relational data to represent and access this data according to an object model. Or you can store object data in object tables, where each row is an object.

Advantages of Objects:-

In general, the object-type model is similar to the class mechanism found in C++ and Java. Like classes, objects make it easier to model complex, real-world business entities and logic, and the reusability of objects makes it possible to develop database applications faster and more efficiently. By natively supporting object types in the database, Oracle enables application developers to directly access the data structures used by their applications. No mapping layer is required between client-side objects and the relational database columns and tables that contain the data.

Objects Can Encapsulate Operations Along with Data

Database tables contain only data. Objects can include the ability to perform operations that are likely to be needed on that data. Thus a purchase order object might include a method to sum the cost of all the items purchased. Or a customer object might have methods to return the customer's buying history and payment pattern. An application can simply call the methods to retrieve the information.

Objects Are Efficient

Object types and their methods are stored with the data in the database, so they are available for any application to use. Developers can benefit from work that is already done and do not need to re-create similar structures in every application.

You can fetch and manipulate a set of related objects as a single unit. A single request to fetch an object from the server can retrieve other objects that are connected to it. For example, when you select a customer object and get the customer's name, phone, and the multiple parts of his address in a single round-trip between the client and the server. When you reference a column of a SQL object type, you retrieve the whole object.

Syntax :-

```
CREATE TYPE <NAME> AS OBJECT
(COLNAME DATATYPE(SIZE) ,
COLNAME DATATYPE(SIZE) ,
 ----------------
 ) ;
```

Example :-

Create object type for storing ADDRESS

```
SQL>CREATE OR REPLACE TYPE ADDR AS OBJECT
                    (hno     VARCHAR2(20) ,
                     street VARCHAR2(20) ,
                     city     VARCHAR2(20),
                     state   CHAR(2));
```

Creating Table :-

After creating user defined datatype then it can be used as a datatype for any column in any table.

```
SQL>CREATE TABLE customer
        (cid              NUMBER(2) ,
         cname           VARCHAR2(20) ,
         caddr           ADDR);
```

Inserting Record Into Table :-

```
SQL>INSERT INTO customer VALUES(1, ’RAJU’, ADDR(‘102A’,’AMEERPET’,’HYD’,’AP’)) ;
```

Display customer records residing in AMEERPET ?

```
SQL>SELECT * FROM customer c  WHERE c.caddr.street=’AMEERPET’ ;
```

Update customer HNO whose cid =1 ?

```
SQL>UPDATE customer c SET  c.caddr.hno = ‘110’  WHERE   cid=1;
```

Methods :-

You can also defined methods that apply to datatype ,and by applying those datatypes to table , you can apply those methods to the data in those tables.
Before creating the body for a method , you must name the method within the datatype declaration.

Type Declaration :-

```
        SQL>CREATE OR REPLACE TYPE emp_type AS OBJECT
                (ename          VARCHAR2(20),
                Birthdate       DATE ,
                Member function AGE(birthdate IN DATE)  return NUMBER);
```

Type Body :-

```
        SQL>CREATE OR REPLACE TYPE BODY emp_type AS
                Member function AGE(birthdate IN DATE) return NUMBER is
                Begin
                        Return (round((sysdate-birthdate)/365));
                End;
```

Creating table :

```
SQL>CREATE TABLE emp
                (empid          NUMBER(4),
                Emp_details     emp_type) ;

SQL>INSERT INTO emp VALUES(1, emp_type('smith','10-JAN-1981'));
```

You can use AGE function within a QUERY as follows.

```
SQL>SELECT E.EMP_DETAILS.AGE(E.EMP_DETAILS.BIRTHDATE) AS AGE FROM EMP E;
```

Implementing Object Views :-

After the DB design is properly normalized , you should look for group of columns that consistitute the representation of an object type. You can then create your tables based on the object types. But what if your tables already exists and what if you have previously created a relational database application and now want to implement object-relational concepts in your application without rebuilding and re-creating the entire application , to do this oracle provides **object views** as a means for defining objects used by existing relational tables.

Example :-

```
SQL>CREATE TABLE customer(
                customer_id   NUMBER  PRIMARY KEY,
                cname           VARCHAR2(20),
                street          VARCHAR2(20),
                city            VARCHAR2(20),
                state           CHAR(2),
                zip             NUMBER);
```

if you want to create another table or application that stores information about people and addresses, you may choose to create **ADDRESS_TY** and **PERSON_TY** object types and you want to apply these types to **CUSTOMER** table

```
SQL>CREATE OR REPLACE TYPE ADDRESS_TY as object
                (Street         VARCHAR2(20),
                City            VARCHAR2(20),
                State           VARCHAR2(20),
                Zip             NUMBER);

SQL>CREATE OR REPLACE TYPE PERSON_TY as object
                (Name           VARCHAR2(20),
                 Address        ADDRESS_TY);
```

Now create an object view based on the CUSTOMER table using the object types you have defined.

```
SQL>CREATE VIEW CUSTOMER_OV(Customer_ID,Person) as
          SELECT Customer_ID,
                    PERSON_TY(Name, ADDRESS_TY(Street,City,State,Zip))
          FROM CUSTOMER;
```

Now you can insert the data into CUSTOMER table via CUSTOMER_OV as follows

```
SQL>insert into CUSTOMER_OV values (1234,
                              PERSON_TY(‘smith’, ADDRESS_TY(‘SRNAGAR’,’HYD’,’AP’,507001);
```

VARRAYS :-

A VARYING ARRAY allows you to store array of values of a numbers or strings or dates.

Example :-

```
SQL>CREATE OR REPLACE TYPE  PHONE_ARRAY AS VARRAY(2) OF NUMBER(10) ;
```

Creating table based on VARRAY :-

```
SQL>CREATE TABLE customer
             (CID           NUMBER(2) ,
              CNAME         VARCHAR2(20) ,
              CPHONE        phone_array ) ;
```

Inserting values :-

```
SQL>INSERT INTO CUST VALUES(1,’JAMES’,PHONE_ARRAY(9878787877,6767673663));
```

NESTED TABLE :-

Whereas varying array have a limited number of entries , a Second type called NESTED TABLE has no limit on the number Of entries per row. A nested table as its name implies a table within a table.It is a table that is represented as a column within in another table.

Example :-

```
SQL>CREATE TYPE NEST_TYPE AS  TABLE OF ADDR ;
```

Using NESTED TABLE :-

```
SQL>CREATE TABLE customer
             (cid           NUMBER(2) ,
              cname          VARCHAR2(20) ,
              caddr          NEST_TYPE)
              NESTED TABLE caddr STORE AS NEST_TYPE_TAB ;
```

Inserting values into Nested Table :-

```
SQL>INSERT INTO CUST VALUES(1,’VIJAY’,
                              NEST_TYPE( ADDR('100A','SRNAGAR','HYD','AP') ,
                                         ADDR('200B','SRNAGAR','HYD','AP')));
```

Performing DML operations on NESTED TABLE :-

to perform DML operations on NESTED TABLE use keyword "THE" .
Display addresses of customer whose CID=1 ?

```
SQL>SELECT hno,street,city,state  FROM  THE (SELECT CADDR FROM cust WHERE cid=1)
```

Update customer street of HNO=100A of customer id =1 ?

```
SQL>UPDATE THE (SELECT caddr FROM customer WHERE cid=1)
                    SET street='AMEERPET'  WHERE hno='100A';
```

Delete the customer address where HNO=100A whose cid=1 ?

```
SQL>DELETE FROM THE (SELECT caddr FROM customer WHERE cid=1)  WHERE hno='100A' ;
```

Insert new address for customer whose CID=1 ?

```
SQL>INSERT INTO THE (SELECT caddr FROM customer WHERE cid=1)
            VALUES('500X','SRNAGAR','HYD','AP') ;
```

Retrieving USER DEFINED TYPES Information :-

USER_TYPES :- Which maintains information about TYPES created by user.

Hierarchical Queries

- A hierarchical query presents data in a inverted tree structure.
- The tree comprises of interconnected nodes.
- Each node may be connected to none,one or more childnodes.
- Each node is connected to one parent node. The top most node is the Root Node that has no parent.
- Nodes that do not have child nodes are called Leaf Nodes.

Syntax :-

```
SELECT [LEVEL] , <collist>
FROM <tabname>
[WHERE <cond>]
[START WITH <cond>]
[CONNECT BY PRIOR <cond>]
```

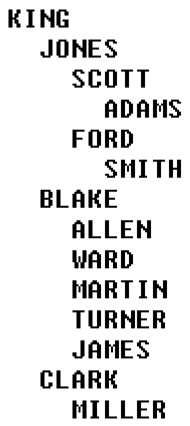
LEVEL :- LEVEL is a pseudo column that returns level of the tree.
START CONDITION :- used to specify root of the tree.
CONNECT BY PRIOR :-

used to specify relationship between Parent and child rows. This clause cannot be used to perform Join operation.

Example :-

Display employee names and their manager name in tree structure ?

```
SQL>SELECT RPAD(' ',LEVEL*2,' ')||ename
           FROM emp
          START WITH ename='KING'
          CONNECT BY PRIOR empno=mgr ;
```



```
KING
  JONES
    SCOTT
      ADAMS
    FORD
      SMITH
  BLAKE
    ALLEN
    WARD
    MARTIN
    TURNER
    JAMES
  CLARK
    MILLER
```

To sort siblings use ORDER SIBLINGS clause.

Example :-

```
SQL>SELECT RPAD(' ',LEVEL*2,' ')||ename FROM emp
          START WITH ename='KING'
          CONNECT BY PRIOR empno=mgr ;
          ORDER SIBLINGS BY ename ;
```

SYS_CONNECT_BY_PATH :-

Returns child node along with path

Example :-

```
SQL>SELECT ename,SYS_CONNECT_BY_PATH(ename,'\')    FROM emp
        START WITH ename='KING'
        CONNECT BY PRIOR empno=mgr
        ORDER SIBLINGS BY ename ;
```

CONNECT_BY_ISLEAF :-

Returns whether the node is leaf node or not , If node is leaf node then it returns 1 otherwise returns 0.

```
SQL>SELECT ename,CONNECT_BY_ISLEAF  as  root   FROM emp
        START WITH ename='KING'
        CONNECT BY PRIOR empno=mgr
        ORDER SIBLINGS BY ename ;
```

Join With CONNECT BY:-

```
SQL>SELECT E.emplevel, SUBSTR(E.ename,1,15) "ENAME", E.empno, D.deptno, D.dname
    FROM dept D, (SELECT level emplevel, LPAD('  ',2*level-2)||ename ename, empno, mgr, deptno
                       FROM emp
                       CONNECT BY PRIOR empno = mgr
                        START WITH ename='KING') E
WHERE E.deptno = D.deptno ;
```

```
LEVEL        ENAME                  EMPNO     DEPTNO     DNAME
  1          KING                   7839      10         ACCOUNTING
  2              CLARK              7782      10         ACCOUNTING
  3                MILLER           7934      10         ACCOUNTING
  2          JONES                  7566      20         RESEARCH
  3              SCOTT              7788       20        RESEARCH
  4                   ADAMS         7876      20         RESEARCH
  3              FORD               7902       20        RESEARCH
  4                   SMITH         7369      20         RESEARCH
  2          BLAKE                  7698      30         SALES
  3              ALLEN              7499       30        SALES
  3              WARD               7521      30         SALES
```

Flashback Queries

→Most of the times an application crashes only because of Human errors. A human error could result in data corruption Due to which the application simply halts.The most human errors that causes an application to go down are Accidental deletion of valuable Data , droping the table.
→Oracle offers a solution called Flashbacking to recover data Due to human errors.
→Flashback technology allows viewing data back in time.
→Flashback technology is introduced in ORACLE 9i
→In ORACLE 10g Flashbacking made simple and flexible.
→A flashback allows reverting mistakenly committed changes by viewing the records before the commit was executed.

Flashbacking Data :-

Flashbacking is based on →Timestamp
→SCN (system change number)

Flashbacking based on Timestamp :-

To view data that exists before 1 day in emp table execute the following query ?

```
SQL>SELECT * FROM EMP
              AS OF TIMESTAMP (SYSTIMESTAMP – INTERVAL '1' DAY);
```

To view data that exists before 30 minutes ?

```
SQL>SELECT * FROM EMP
               AS OF TIMESTAMP(SYSTIMESTAMP – INTERVAL '30'  MINUTE) ;
```

To view data that exist at particular DATE & TIME ?

```
SQL>SELECT * FROM EMP
        AS OF TIMESTAMP(TO_TIMESTAMP('01-JAN-12 10:00:00 AM','DD-MON-YY HH:MI:SS AM'));
```

Using DBMS_FLASHBACK package :-

```
SQL>EXECUTE DBMS_FLASHBACK.ENABLE_AT_TIME(SYSDATE-2) ;
```

When the above procedure is executed it enables flashback mode and any query runs after this it display data that exists before 2 days

Example :-

```
SQL>SELECT * FROM EMP ;
```

Flashback can be disabled by

```
SQL>EXECUTE DBMS_FLASHBACK.DISABLE() ;
```

Flashback Table :-

→Oracle flashback feature allows recovering a table after drop.When a table is dropped oracle moves it to the recyclebin rather than actually droping it.

→A Recycle Bin is logical collection of dropped objects. The contents of Recycle Bin is viewed by using SHOW RECYCLEBIN command.

→Flashback command is introduced in ORACLE 10g , which is used to restore a table after drop.

Example :-

```
SQL>DROP TABLE EMP ;  (table is moved to recyclebin)
```

To restore the table EMP issue the following command

```
SQL>FLASHBACK TABLE EMP TO BEFORE DROP;
```

A table can be restored with a new name.

```
SQL>FLASHBACK TABLE EMP TO BEFORE DROP RENAME TO EMPL;
```

PURGE command :-

PURGE command releases space occupied by the object in recyclebin .

```
SQL>PURGE TABLE EMP ;
```

After purging the table FLASHBACKING is not possible

To empty the RECYCLEBIN issue the following command

```
SQL>PURGE RECYCLEBIN ;
```

The table can be dropped without being sent to the RECYCLEBIN. To do so issue the following command.

```
SQL>DROP TABLE EMP PURGE ;
```

ENABLING & DISABLING RECYCLEBIN :-

→Recyclebin can be enabled & disabled.

→By default recyclebin is enabled.

→When it is enabled the dropped objects are placed in recyclebin.

→When it is disabled the dropped objects are not placed in recyclebin.

→To disable the recyclebin issue the following command

```
SQL>ALTER SESSION SET RECYCLEBIN = OFF
```

To enable the Recyclebin issue the following command

SQL>ALTER SESSION SET RECYCLEBIN =ON

OCA question :-

Evaluate the following SQL statements in the given order:

```
DROP TABLE dept;
CREATE TABLE dept
(deptno NUMBER(3) PRIMARY KEY,
deptname VARCHAR2(10));
DROP TABLE dept;
FLASHBACK TABLE dept TO BEFORE DROP;
```

Which statement is true regarding the above FLASHBACK operation?

A. It recovers only the first DEPT table.
B. It recovers only the second DEPT table.
C. It does not recover any of the tables because FLASHBACK is not possible in this case.
D. It recovers both the tables but the names would be changed to the ones assigned in the RECYCLEBIN.

ORACLE 12c FEATURES :-

In Oracle 12c, it is now possible to specify the CURRVAL and NEXTVAL sequence pseudocolumns as the default values for a column. You should also consider using Identity columns for this purpose.In the following example you can see the effect of specifying a sequence as the default value for a column. The default value is only used when the column is not referenced by the insert. This behaviour can be modified using the ON NULL clause described in the next section.

```
CREATE SEQUENCE t1_seq;
CREATE TABLE t1 (
  id        NUMBER DEFAULT t1_seq.NEXTVAL,

  description VARCHAR2(30)

);

INSERT INTO t1 (description) VALUES ('DESCRIPTION only');

INSERT INTO t1 (id, description) VALUES (999, 'ID=999 and DESCRIPTION');
INSERT INTO t1 (id, description) VALUES (NULL, 'ID=NULL and DESCRIPTION');
SELECT * FROM t1;
       ID              DESCRIPTION

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

        1              DESCRIPTION only

       999      ID=999 and DESCRIPTION

                       ID=NULL and DESCRIPTION
3    rows selected.
```

Identity Columns in Oracle Database 12c Release 1 (12.1)

In previous releases of the Oracle database, there was no direct equivalent of the AutoNumber or

Identity functionality of other database engines. Instead, this behaviour had to be implemented using a combination of sequences and triggers. Oracle 12c introduces two alternatives to this by providing identity columns and the ability to use sequence pseudocolumns as default values. This article will focus on the use of identity columns.

12c database introduces the ability define an identity clause against a table column defined using a numeric type. The syntax is show below.

GENERATED
[ALWAYS | BY DEFAULT [ON NULL]]
AS IDENTITY [(identity_options)]

Ignoring the identity_options, which match those of the CREATE SEQUENCE statement, this syntax allows us to use three variations on the identity functionality.

Before we can look at some examples, you need to make sure your test user has the CREATE SEQUENCE privilege. Without it, attempts to define an identity column will produce a "ORA-01031: insufficient privileges" error.

```
CONN / AS SYSDBA

ALTER SESSION SET CONTAINER=pdb1;

GRANT CREATE TABLE, CREATE SEQUENCE TO test;

CONN test/test@pdb1
```

Using ALWAYS forces the use of the identity. If an insert statement references the identity column, even to specify a NULL value, an error is produced.

```
CREATE TABLE identity_test_tab (

  id          NUMBER GENERATED ALWAYS AS IDENTITY,

  description VARCHAR2(30)

);

SQL> INSERT INTO identity_test_tab (description) VALUES ('Just DESCRIPTION');
1 row created.
SQL> INSERT INTO identity_test_tab (id, description) VALUES (NULL, 'ID=NULL and DESCRIPTION');
ERROR at line 1:

ORA-32795: cannot insert into a generated always identity column

SQL> INSERT INTO identity_test_tab (id, description) VALUES (999, 'ID=999 and DESCRIPTION');
INSERT INTO identity_test_tab (id, description) VALUES (999, 'ID=999 and DESCRIPTION')
ERROR at line 1:
ORA-32795: cannot insert into a generated always identity column
```

Using BY DEFAULT allows you to use the identity if the column isn't referenced in the insert statement, but if the column is referenced, the specified value will be used in place of the identity. Attempting to specify the value NULL in this case results in an error, since identity columns are always NOT NULL.

```
DROP TABLE identity_test_tab PURGE;
CREATE TABLE identity_test_tab (
 id         NUMBER GENERATED BY DEFAULT AS IDENTITY,

 description VARCHAR2(30)

);

SQL> INSERT INTO identity_test_tab (description) VALUES ('Just DESCRIPTION');
1 row created.
SQL> INSERT INTO identity_test_tab (id, description) VALUES (999, 'ID=999 and DESCRIPTION');
1 row created.

SQL> INSERT INTO identity_test_tab (id, description) VALUES (NULL, 'ID=NULL and DESCRIPTION');
INSERT INTO identity_test_tab (id, description) VALUES (NULL, 'ID=NULL and DESCRIPTION')
ERROR at line 1:

ORA-01400: cannot insert NULL into ("TEST"."IDENTITY_TEST_TAB"."ID")
```

Using BY DEFAULT ON NULL allows the identity to be used if the identity column is referenced, but a value of NULL is specified.

```
DROP TABLE identity_test_tab PURGE;
CREATE TABLE identity_test_tab (
 id         NUMBER GENERATED BY DEFAULT ON NULL AS IDENTITY,

 description VARCHAR2(30)

);


SQL> INSERT INTO identity_test_tab (description) VALUES ('Just DESCRIPTION');
1 row created.
SQL> INSERT INTO identity_test_tab (id, description) VALUES (999, 'ID=999 and DESCRIPTION');
1 row created.

SQL> INSERT INTO identity_test_tab (id, description) VALUES (NULL, 'ID=NULL and DESCRIPTION');
1 row created.

SQL> SELECT * FROM identity_test_tab;
     ID                  DESCRIPTION
------------            -----------------------------------


      1                 Just DESCRIPTION
```

999	ID=999 and DESCRIPTION
2	ID=NULL and DESCRIPTION

Top –N Queries :-

Top-N query is used to retrieve the top or bottom N rows from an ordered set. Combining two Top-N queries gives you the ability to page through an ordered set. This concept is not a new one. In fact, Oracle already provides multiple ways to perform Top-N queries, as discussed here. These methods work fine, but they look rather complicated compared to the methods provided by other database engines.
To be consistent, we will use the same example table used in the Top-N Queries article.

Create and populate a test table.

```
DROP TABLE rownum_order_test;

CREATE TABLE rownum_order_test (
  val  NUMBER
);

INSERT ALL
  INTO rownum_order_test
INTO   rownum_order_test
SELECT level
FROM   dual
CONNECT BY level <= 10;

COMMIT;
```

The following query shows we have 20 rows with 10 distinct values.

```
SELECT val
FROM  rownum_order_test
ORDER BY val;

       VAL

         1
         1
-------------
         2
         2
         3
         3
         4
         4
         5
         5
         6

       VAL

         6
         7
---------7---
         8
         8
         9
         9
        10
        10
```

```
20 rows selected.

SQL>
```

Top-N Queries

The syntax for the row limiting clause looks a little complicated at first glance.

```
[ OFFSET offset { ROW | ROWS } ]
[ FETCH { FIRST | NEXT } [ { rowcount | percent PERCENT } ]
    { ROW | ROWS } { ONLY | WITH TIES } ]
```

Actually, for the classic Top-N query it is very simple. The example below returns the 5 largest values from an ordered set. Using the ONLY clause limits the number of rows returned to the exact number requested.

```
SELECT val
FROM   rownum_order_test
ORDER BY val DESC
FETCH FIRST 5 ROWS ONLY;

       VAL

        10
        10
         9
------------
         9
         8

5 rows selected.

SQL>
```

Using the WITH TIES clause may result in more rows being returned if multiple rows match the value of the Nth row. In this case the 5th row has the value "8", but there are two rows that tie for 5th place, so both are returned.

```
SELECT val
FROM   rownum_order_test
ORDER BY val DESC
FETCH FIRST 5 ROWS WITH TIES;

       VAL

        10

------------
```

```
        10
         9
         9
         8
         8

6 rows selected.

SQL>
```

In addition to limiting by row count, the row limiting clause also allows us to limit by percentage of rows. The following query returns the bottom 20% of rows.

```
SELECT val
FROM  rownum_order_test
ORDER BY val
FETCH FIRST 20 PERCENT ROWS ONLY;

       VAL
----------
         1
         1
         2
         2

4 rows selected.

SQL>
```

Paging Through Data

Paging through an ordered resultset was a little annoying using the classic Top-N query approach, as it required two Top-N queries, one nested inside the other. For example, if we wanted the second block of 4 rows we might do the following.

```
SELECT val
FROM  (SELECT val, rownum AS rnum
       FROM  (SELECT val
              FROM  rownum_order_test
              ORDER BY val)
       WHERE rownum <= 8)
WHERE  rnum >= 5;

       VAL
----------
         3
       3
       4
       4

4 rows selected.

SQL>
```

With the row limiting clause we can achieve the same result using the following query.

```
SELECT val
FROM  rownum_order_test
ORDER BY val
OFFSET 4 ROWS FETCH NEXT 4 ROWS ONLY;

      VAL

       3
       3
------------
       4
       4

4 rows selected.

SQL>
```

The starting point for the FETCH is OFFSET+1.

The OFFSET is always based on a number of rows, but this can be combined with a FETCH using a PERCENT.

```
SELECT val
FROM  rownum_order_test
ORDER BY val
OFFSET 4 ROWS FETCH NEXT 20 PERCENT ROWS ONLY;

      VAL

       3
       3
       4
       4
------------
4 rows selected.
```

```
SQL>
```

Not surprisingly, the offset, rowcount and percent can, and probably should, be bind variables.

```
VARIABLE v_offset NUMBER;
VARIABLE v_next NUMBER;

BEGIN
  :v_offset := 4;
  :v_next  := 4;
END;
/

SELECT val
FROM  rownum_order_test
ORDER BY val
OFFSET :v_offset ROWS FETCH NEXT :v_next ROWS ONLY;

       VAL
------------
         3
         3
         4
         4

SQL>
```

Extra Information

- The keywords ROW and ROWS can be used interchangeably, as can the FIRST and NEXT keywords. Pick the ones that scan best when reading the SQL like a sentence.
- If the offset is not specified it is assumed to be 0.
- Negative values for the offset, rowcount or percent are treated as 0.
- Null values for offset, rowcount or percent result in no rows being returned.
- Fractional portions of offset, rowcount or percent are truncated.
- If the offset is greater than or equal to the total number of rows in the set, no rows are returned.
- If the rowcount or percent are greater than the total number of rows after the offset, all rows are returned.
- The row limiting clause can not be used with the FOR UPDATE clause, CURRVAL and NEXTVAL sequence pseudocolumns or in an fast refresh materialized view.

SQL*LOADER

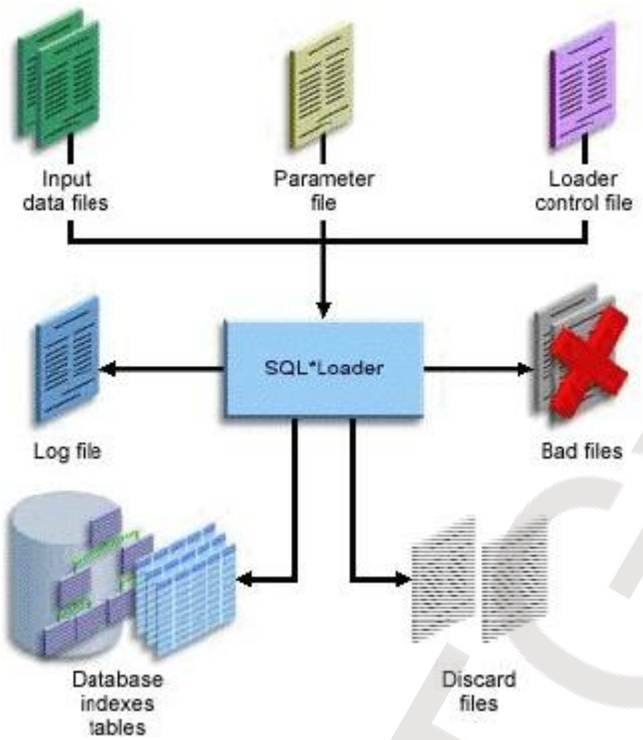
SQL*LOADER utility is used to load data from other data source into Oracle. For example, if you have a table in FOXPRO, ACCESS or SYBASE or any other third party database, you can use SQL Loader to load the data into Oracle Tables. SQL Loader will only read the data from Flat files. So If you want to load the data from Foxpro or any other database, you have to first convert that data into Delimited Format flat file or Fixed length format flat file, and then use SQL loader to load the data into Oracle.

SQL*Loader:

- Has a powerful data parsing engine which puts little limitation on the format of the data in the datafile.
- Can load data from multiple datafiles during the same load session.
- Can load data into multiple tables during the same load session.
- Is character set aware (you can specify the character set of the data).
- Can selectively load data (you can load records based on the records' values).
- Can manipulate the data before loading it, using SQL functions.
- Can generate unique sequential key values in specified columns.
- Can use the operating system's file system to access the datafile(s).
- Can load data from disk, tape, or named pipe.
- Can load arbitrarily complex object-relational data.
- Supports secondary datafiles for loading of LOBs and collections.

SQL*LOADER overview :-

SQL*Loader takes as input a control file, which controls the behavior of SQL*Loader, and one or more datafiles. Output of the SQL*Loader is an Oracle database (where the data is loaded), a log file, a bad file, and potentially a discard file.



Input Data and Datafiles:-

SQL*Loader reads data from one or more files (or operating system equivalents of files) specified in the control file.

INFILE: Specifying Datafiles. From SQL*Loader's perspective, the data in the datafile is organized as records. A particular datafile can be in fixed record format, variable record format, or stream record format.

Fixed Record Format

When all the records in a datafile are of the same byte length, the file is in fixed record format. Although this format is the least flexible, it does result in better performance than variable or stream format.

Variable Record Format

When you specify that a datafile is in variable record format, SQL*Loader expects to find the length of each record in a character field at the beginning of each record in the datafile.

SQL*Loader Control File :-

The control file is a text file written in a language that SQL*Loader understands. The control file describes the task that the SQL*Loader is to carry out. The control file tells SQL*Loader where to find the data, how to parse and interpret the data, where to insert the data, and more.

a control file can be said to have three sections:

The first section contains session-wide information, for example: global options such as bindsize, rows, records to skip, etc. INFILE clauses to specify where the input data is located

The second section consists of one or more "INTO TABLE" blocks. Each of these blocks contains information about the table into which the data is to be loaded such as the table name and the columns of the table.

The third section is optional and, if present, contains input data.

CASE STUDY (Loading Data from MS-ACCESS to Oracle)

Suppose you have a table in MS-ACCESS by name EMP, running under Windows O/S, with the following structure

EMPNO	INTEGER
NAME	TEXT(50)
SAL	CURRENCY
JDATE	DATE

This table contains some 10,000 rows. Now you want to load the data from this table into an Oracle Table. Oracle Database is running in LINUX O/S.

Steps:-

Start MS-Access and convert the table into comma delimited flat (popularly known as csv) , by clicking on File/Save As menu. Let the delimited file name be emp.csv

Now transfer this file to Linux Server using FTP command
Go to Command Prompt in windows
At the command prompt type FTP followed by IP address of the server running Oracle.
FTP will then prompt you for username and password to connect to the Linux Server. Supply a valid username and password of Oracle User in Linux

For example:-

```
C:\>ftp 200.200.100.111

Name: oracle
Password:oracle

FTP>
```

Now give PUT command to transfer file from current Windows machine to Linux machine.

```
FTP>put

Local file:C:\>emp.csv
remote-file: /u01/oracle/emp.csv
File transferred in 0.29 Seconds
FTP>
```

Now after the file is transferred quit the FTP utility by typing bye command.

```
FTP>bye

Good-Bye
```

Now come the Linux Machine and create a table in Oracle with the same structure as in MS-ACCESS by taking appropriate datatypes. For example, create a table like this

```
$sqlplus scott/tiger

SQL>CREATE TABLE emp (empno number(5),
                       name varchar2(50),
                      sal  number(10,2),
                      jdate date);
```

After creating the table, you have to write a control file describing the actions which SQL Loader should do. You can use any text editor to write the control file. Now let us write a controlfile for our case study

```
$vi emp.ctl
    LOAD DATA
    INFILE    ‘/u01/oracle/emp.csv’
    BADFILE       ‘/u01/oracle/emp.bad’
    DISCARDFILE     ‘/u01/oracle/emp.dsc’
    INSERT INTO TABLE emp
    FIELDS TERMINATED BY “,” OPTIONALLY ENCLOSED BY ‘”’  TRAILING NULLCOLS
    (empno,name,sal,jdate date ‘mm/dd/yyyy’)
```

After you have wrote the control file save it and then, call SQL Loader utility by typing the following command

```
$sqlldr userid=scott/tiger
control=emp.ctl log=emp.log
```

After you have executed the above command SQL Loader will shows you the output describing how many rows it has loaded.

The LOG option of sqlldr specifies where the log file of this sql loader session should be created. The log file contains all actions which SQL loader has performed i.e. how many rows were loaded, how many were rejected and how much time is taken to load the rows and etc. You have to view this file for any errors encountered while running SQL Loader.

CASE STUDY (Loading Data from Fixed Length file into Oracle)

Suppose we have a fixed length format file containing employees data, as shown below, and wants to load this data into an Oracle table.

```
7782 CLARK     MANAGER      7839      2572.50                    10
7839 KING      PRESIDENT              5500.00                    10

7934 MILLER    CLERK        7782      920.00                     10

7566 JONES     MANAGER      7839      3123.75                    20
```

```
7499 ALLEN     SALESMAN    7698      1600.00              300.00       30

7654 MARTIN    SALESMAN    7698      1312.50              1400.00      30

7658 CHAN      ANALYST     7566      3450.00                           20

7654 MARTIN    SALESMAN    7698      1312.50              1400.00      30
```

Steps :-

1. First Open the file in a text editor and count the length of fields, for example in our fixed length file, employee number is from 1st position to 4th position, employee name is from 6th position to 15th position, Job name is from 17th position to 25th position. Similarly other columns are also located.
2. Create a table in Oracle, by any name, but should match columns specified in fixed length file. In our case give the following command to create the table.

```
SQL> CREATE TABLE emp (empno  NUMBER(5),
                       name VARCHAR2(20),
                       job  VARCHAR2(10),
                       mgr  NUMBER(5),
                       sal  NUMBER(10,2),
                       comm NUMBER(10,2),
                       deptno     NUMBER(3));
```

3. After creating the table, now write a control file by using any text editor

```
$vi empfix.ctl
  LOAD DATA
  INFILE '/u01/oracle/fix.dat'
  INTO TABLE emp
  (  empno      POSITION(01:04)   INTEGER EXTERNAL,
     name       POSITION(06:15)   CHAR,
     job        POSITION(17:25)   CHAR,
     mgr        POSITION(27:30)   INTEGER EXTERNAL,
     sal        POSITION(32:39)   DECIMAL EXTERNAL,
     comm       POSITION(41:48)   DECIMAL EXTERNAL,
     deptno     POSITION(50:51)   INTEGER EXTERNAL)
```

CASE STUDY (Loading Data into Multiple Tables using WHEN condition)

You can simultaneously load data into multiple tables in the same session. You can also use WHEN condition to load only specified rows which meets a particular condition (only equal to "=" and not equal to "<>" conditions are allowed).
For example, suppose we have a fixed length file as shown below

```
7782 CLARK    MANAGER    7839   2572.50                     10
7839 KING     PRESIDENT         5500.00                     10
7934 MILLER   CLERK      7782   920.00                      10
7566 JONES    MANAGER    7839   3123.75                     20
7499 ALLEN    SALESMAN   7698   1600.00      300.00         30
7654 MARTIN   SALESMAN   7698   1312.50      1400.00        30
```

```
7658 CHAN      ANALYST      7566    3450.00                    20
7654 MARTIN     SALESMAN   7698    1312.50        1400.00      30
```

Now we want to load all the employees whose deptno is 10 into emp1 table and those employees whose deptno is not equal to 10 in emp2 table. To do this first create the tables emp1 and emp2 by taking appropriate columns and datatypes. Then, write a control file as shown below

$vi emp_multi.ctl

```
Load Data
infile ‘/u01/oracle/empfix.dat’
APPEND into table scott.emp1
WHEN (deptno=’10 ‘)
 (empno       POSITION(01:04)  INTEGER EXTERNAL,
  name        POSITION(06:15)  CHAR,
  job         POSITION(17:25)  CHAR,
  mgr         POSITION(27:30)  INTEGER EXTERNAL,
  sal         POSITION(32:39)  DECIMAL EXTERNAL,
  comm        POSITION(41:48)  DECIMAL EXTERNAL,
  deptno      POSITION(50:51)  INTEGER EXTERNAL)
 INTO TABLE scott.emp2
 WHEN (deptno<>’10 ‘)
 (empno       POSITION(01:04)  INTEGER EXTERNAL,
  name        POSITION(06:15)  CHAR,
  job         POSITION(17:25)  CHAR,
  mgr         POSITION(27:30)  INTEGER EXTERNAL,
  sal         POSITION(32:39)  DECIMAL EXTERNAL,
  comm        POSITION(41:48)  DECIMAL EXTERNAL,
  deptno      POSITION(50:51)  INTEGER EXTERNAL)
```

After saving the file emp_multi.ctl run sqlldr

```
$sqlldr userid=scott/tiger
    control=emp_multi.ctl
```

Example where datafile is in the Control file:

```
LOAD DATA
INFILE *
INTO TABLE emp
FIELDS TERMINATED BY “,”
( emp_num, emp_name, mgr, department_name )

BEGINDATA
7369,SMITH,7902,Accounting
7499,ALLEN,7698,Sales
7521,WARD,7698,Accounting
7566,JONES,7839,Sales
7654,MARTIN,7698,Accounting
```

TYPE OF LOADING:

INSERT — If the table you are loading is empty, INSERT can be used.

APPEND — If data already exists in the table, SQL*Loader appends the new rows to it. If data doesn't already exist, the new rows are simply loaded.

REPLACE — All rows in the table are deleted and the new data is loaded

TRUNCATE — SQL*Loader uses the SQL TRUNCATE command.

External Tables

The external tables feature is a complement to existing SQL*Loader functionality. It enables you to access data in external sources as if it were in a table in the database. Prior to Oracle Database 10g, external tables were read-only. However, as of Oracle Database 10g, external tables can also be written to.
External tables are created using the **SQL CREATE TABLE...ORGANIZATION EXTERNAL** statement. When you create an external table, you specify the following attributes:
TYPE :- specifies the type of external table. The two available types are the ORACLE_LOADER type and the ORACLE_DATAPUMP type.
ORACLE_LOADER :- access driver is the default. It can perform only data loads, and the data must come from text datafiles. Loads from external tables to internal tables are done by reading from the external tables' text-only datafiles.
ORACLE_DATAPUMP :- access driver can perform both loads and unloads. The data must come from binary dump files. Loads to internal tables from external tables are done by fetching from the binary dump files. Unloads from internal tables to external tables are done by populating the external tables' binary dump files.

DEFAULT DIRECTORY :- specifies the default location of files that are read or written by external tables. The location is specified with a directory object, not a directory path.
ACCESS PARAMETERS :- describe the external data source and implements the type of external table that was specified. Each type of external table has its own access driver that provides access parameters unique to that type of external table.
LOCATION:-- specifies the location of the external data. The location is specified as a list of directory objects and filenames. If the directory object is not specified, then the default directory object is used as the file location.
To create EXTERNAL TABLE follow below steps.
Step 1 :- Creating Directory Object :-

The access driver does not allow you to specify a complete specification for files. This is because the server may have access to files that you do not, and allowing you to read this data would affect security.
Instead, you are required to specify directory objects as the locations from which to read files and write files. A directory object maps a name to a directory name on the file system. For example, the following statement creates a directory object named ext_tab_dir that is mapped to a directory located at /usr/apps/datafiles.
Directory objects can be created by DBAs or by any user with the CREATE ANY DIRECTORY privilege.
SQL>CREATE DIRECTORY ext_tab_dir AS '/usr/apps/datafiles';
After a directory is created, the user creating the directory object needs to grant READ and WRITE privileges on the directory to other users.

```
SQL>GRANT READ,WRITE ON DIRECTORY ext_tab_dir TO scott;
```

Step 2 :- CREATE TABLE EMP :-

```
SQL>CREATE TABLE emp (      emp_no CHAR(6),
                            last_name CHAR(25),
                            first_name CHAR(20),
                            middle_initial CHAR(1));
```

Step 3 CREATE EXTERNAL TABLE :-

```
    CREATE TABLE emp_load (employee_number CHAR(5), employee_last_name CHAR(20),
                 employee_first_name CHAR(15),
                 employee_middle_name CHAR(15))
    ORGANIZATION EXTERNAL (TYPE ORACLE_LOADER DEFAULT DIRECTORY ext_tab_dir
                ACCESS PARAMETERS (RECORDS DELIMITED BY NEWLINE FIELDS
                (employee_number CHAR(2),
                employee_dob CHAR(20),
                employee_last_name CHAR(18),
                employee_first_name CHAR(11),
                employee_middle_name CHAR(11)))
                LOCATION ('info.dat'));
```

Step 4 :- Load the data from the external table emp_load into the table emp:

```
INSERT INTO emp (emp_no, first_name, middle_initial, last_name)
(SELECT employee_number, employee_first_name, substr(employee_middle_name, 1, 1),
     employee_last_name
        FROM emp_load);
```

Step 5 :- Perform the following select operation to verify that the information in the .dat file was loaded into the emp table:

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

Analytical Functions

→Analytic Functions are commonly used to compute cumulative, moving, centered and reporting aggregates.

→Oracle provides several analytic functions that help compute an aggregate value based on a group of rows.

→Analytic Functions provided by Oracle open up a whole new way of looking at the data.

→It helps remove a lot of procedural code and complex code spec that would have taken a long time to develop, to achieve the same result.

→Whatever an analytic function does, can be done by using SQL, with the help of joins and subqueries. However, an analytic function always does it faster, when compared to native SQL.

Getting Started With Analytic Functions:-

Oracle provides the following Analytic Functions.

- AVG
- CORR
- COVAR_POP
- COVAR_SAMP
- COUNT
- CUME_DIST
- DENSE_RANK
- FIRST
- FIRST_VALUE
- LAG
- LAST
- LAST_VALUE
- LEAD
- MAX
- MIN
- NTILE
- PERCENT_RANK
- PERCENTILE_CONT
- PERCENTILE_DISC
- RANK
- RATIO_TO_REPORT
- STDDEV
- STDDEV_POP
- STDDEV_SAMP
- SUM
- VAR_POP
- VAR_SAMP
- VARIANCE

Syntax:-

Function(arg1,..., argn) OVER ([PARTITION BY <...>] [ORDER BY <....>] [<window_clause>])

Where,

Arguments: Analytic functions accept 0 to 3 arguments.

Query Partition Clause:

→The PARTITION BY clause logically breaks a single result set into N groups, according to the criteria set by the partition expressions.
→The words partition and group are used synonymously here.
→The analytic functions cane be applied to each group independently.

Order By Clause:

The ORDER BY clause specifies how the data is sorted within each group [partition]. This will definitely affects the outcome of any analytic function.

Windowing Clause:

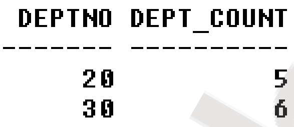
The windowing clause allows defining a sliding or anchored window of data, on which the analytic function will operate, within a group.

Windows can be used for:

→RANGES of data values
→ROWS offset from the current row
→An ORDER BY clause in an analytic function adds a default window clause of RANGE UNBOUNDED PRECEDING which means all the previous rows available in a partition are considered.

How are analytic functions different from group or aggregate functions?

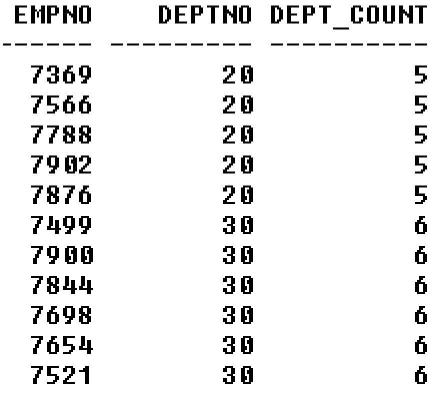
```
SQL>SELECT deptno,COUNT(*) DEPT_COUNT
        FROM emp
        WHERE deptno IN (20, 30)
        GROUP BY deptno;
```



```
 DEPTNO DEPT_COUNT
------- ----------
     20          5
     30          6
```

Consider the above and its result. The above returns departments and their employee count. Most importantly it groups the records into departments. As such any non-"group by" column is not allowed in the select clause.

SQL>SELECT empno, deptno, COUNT(*) OVER (PARTITION BY deptno) DEPT_COUNT
FROM emp
WHERE deptno IN (20, 30);



```
 EMPNO     DEPTNO DEPT_COUNT
------ ---------- ----------
  7369         20          5
  7566         20          5
  7788         20          5
  7902         20          5
  7876         20          5
  7499         30          6
  7900         30          6
  7844         30          6
  7698         30          6
  7654         30          6
  7521         30          6
```

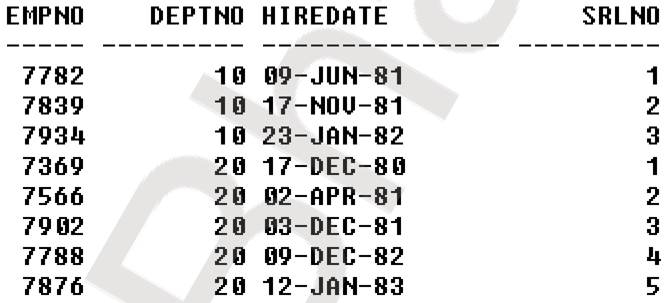
Now consider the analytic function query and its result. Note the repeating values of DEPT_COUNT column.This brings out the main difference between aggregate and analytic functions. Analytical function returns aggregate data along with detailed data.

Analytic functions are computed after WHERE clause, GROUP BY and HAVING . The ORDER BY clause of the query operates after the analytic functions. So analytic functions can only appear in the select list and in the ORDER BY clause of the query.

In absence of any PARTITION or <window_clause> inside the OVER( ) portion, the function acts on entire record set returned by the where clause.

ROW_NUMBER( ) :-
gives a running serial number to a partition of records. It is very useful in reporting, especially in places where different partitions have their own serial numbers. In below ecxmple the function ROW_NUMBER( ) is used to give separate sets of running serial to employees of departments 10 and 20 based on their HIREDATE.

SQL> SELECT empno, deptno, hiredate,ROW_NUMBER( ) OVER (PARTITION BY deptno
ORDER BY hiredate NULLS LAST) SRLNO
FROM emp
WHERE deptno IN (10, 20)
ORDER BY deptno, SRLNO;



```
EMPNO     DEPTNO HIREDATE             SRLNO
----- ---------- --------------- ----------
 7782         10 09-JUN-81                1
 7839         10 17-NOV-81                2
 7934         10 23-JAN-82                3
 7369         20 17-DEC-80                1
 7566         20 02-APR-81                2
 7902         20 03-DEC-81                3
 7788         20 09-DEC-82                4
 7876         20 12-JAN-83                5
```

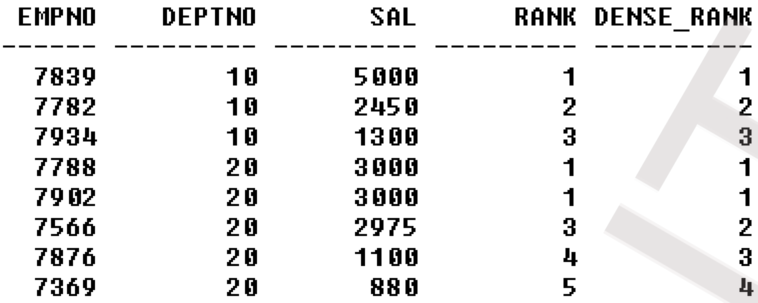
SQL>SELECT empno,ename,sal,SUM(sal) OVER (ORDER BY sal ROWS UNBOUNDED PRECEDING)
AS cum_sal
FROM emp ;

RANK () and DENSE_RANK() :-

Computes the RANK of a row in an ordered group of rows and returns rank as a number begins with 1.this function is useful to find top-N & bottom-N records.

Example :-

```
SQL> SELECT empno, deptno, sal,
RANK() OVER (PARTITION BY deptno ORDER BY sal DESC NULLS LAST) RANK,
DENSE_RANK() OVER (PARTITION BY deptno ORDER BY sal DESC NULLS LAST) DENSE_RANK
FROM emp
WHERE deptno IN (10, 20)
ORDER BY 2, RANK;
```



```
 EMPNO     DEPTNO        SAL       RANK DENSE_RANK
------ ---------- ---------- ---------- ----------
  7839         10       5000          1          1
  7782         10       2450          2          2
  7934         10       1300          3          3
  7788         20       3000          1          1
  7902         20       3000          1          1
  7566         20       2975          3          2
  7876         20       1100          4          3
  7369         20        880          5          4
```

NOTE :- The difference between RANK & DENSE_RANK is RANK function generates gaps but DENSE_RANK function doesn't generate gaps.

RANK() and DENSE_RANK() function can be used to find top-N records as follows :-

Display top 3 maximum salaries in emp table ?

```
SQL>SELECT DISTINCT sal FROM
                  (SELECT sal, DENSE_RANK() OVER (ORDER BY sal DESC) AS RNK FROM EMP)
   WHERE RNK <=3;
```

LEAD and LAG:-

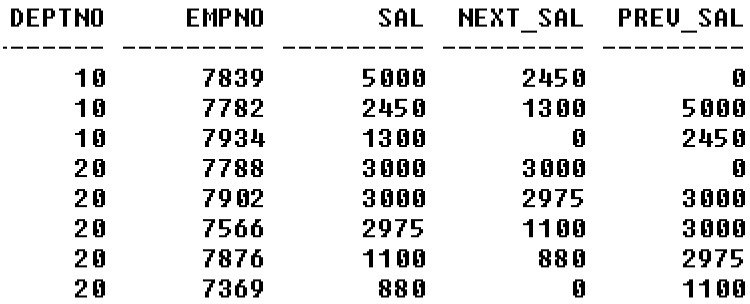
LEAD has the ability to compute an expression on the next rows (rows which are going to come after the current row) and return the value to the current row. The general syntax of LEAD is shown below:

LEAD (<sql_expr>, <offset>, <default>) OVER (<analytic_clause>)

<sql_expr> is the expression to compute from the leading row.
<offset> is the index of the leading row relative to the current row.
<offset> is a positive integer with default 1.
<default> is the value to return if the <offset> points to a row outside the partition range.

The syntax of LAG is similar except that the offset for LAG goes into the previous rows.below example and its result show simple usage of LAG and LEAD function.

```
SQL> SELECT deptno, empno, sal,
LEAD(sal, 1, 0) OVER (PARTITION BY deptno ORDER BY sal DESC NULLS LAST) NEXT_SAL,
LAG(sal, 1, 0) OVER (PARTITION BY deptno ORDER BY sal DESC NULLS LAST) PREV_SAL
FROM emp
WHERE deptno IN (10, 20)
ORDER BY deptno, sal DESC;
```



```
DEPTNO      EMPNO        SAL   NEXT_SAL   PREV_SAL
------- ---------- ---------- ---------- ----------
     10       7839       5000       2450          0
     10       7782       2450       1300       5000
     10       7934       1300          0       2450
     20       7788       3000       3000          0
     20       7902       3000       2975       3000
     20       7566       2975       1100       3000
     20       7876       1100        880       2975
     20       7369        880          0       1100
```

FIRST VALUE and LAST VALUE function:-

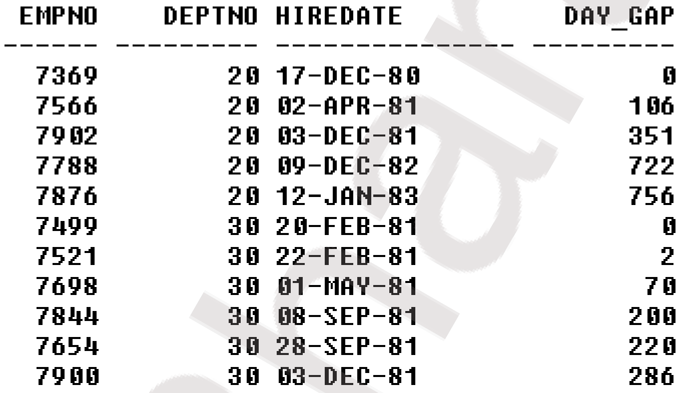
The general syntax is:

FIRST_VALUE(<sql_expr>) OVER (<analytic_clause>)

The FIRST_VALUE analytic function picks the first record from the partition after doing the ORDER BY. The <sql_expr> is computed on the columns of this first record and results are returned. The LAST_VALUE function is used in similar context except that it acts on the last record of the partition.

How many days after the first hire of each department were the next employees hired?

```
SQL>SELECT empno, deptno,hiredate, hiredate - FIRST_VALUE(hiredate)
OVER (PARTITION BY deptno ORDER BY hiredate) DAY_GAP
FROM emp
WHERE deptno IN (20, 30)
ORDER BY deptno, DAY_GAP;
```



```
 EMPNO     DEPTNO HIREDATE             DAY_GAP
------ ---------- --------------- ----------
  7369         20 17-DEC-80                0
  7566         20 02-APR-81              106
  7902         20 03-DEC-81              351
  7788         20 09-DEC-82              722
  7876         20 12-JAN-83              756
  7499         30 20-FEB-81                0
  7521         30 22-FEB-81                2
  7698         30 01-MAY-81               70
  7844         30 08-SEP-81              200
  7654         30 28-SEP-81              220
  7900         30 03-DEC-81              286
```

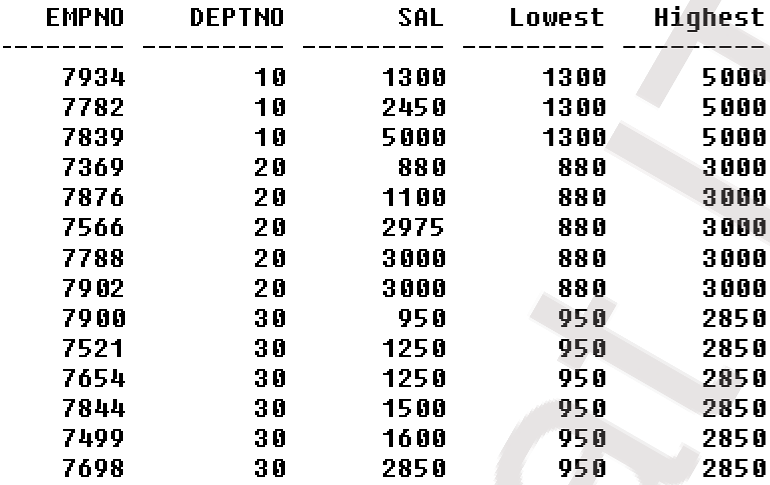
FIRST & LAST functions :-

The FIRST and LAST function can be used to return the first or last value from an ordered sequence. For example we want to display the salary of each employee , along with the lowest and highest within in their department.

The general Synatx:-

Function( ) KEEP (DENSE_RANK FIRST ORDER BY <expr>) OVER (<partitioning_clause>)

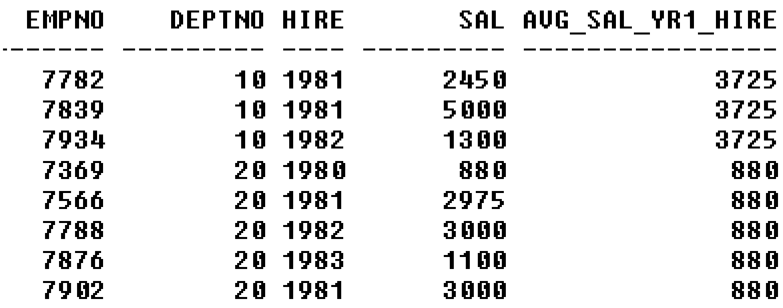
```
SQL>SELECT empno,
       deptno,
       sal,
       MIN(sal) KEEP (DENSE_RANK FIRST ORDER BY sal) OVER (PARTITION BY deptno) "Lowest",
       MAX(sal) KEEP (DENSE_RANK LAST ORDER BY sal) OVER (PARTITION BY deptno) "Highest"
FROM   emp
ORDER BY deptno, sal ;
```



```
     EMPNO     DEPTNO        SAL     Lowest    Highest
---------- ---------- ---------- ---------- ----------
      7934         10       1300       1300       5000
      7782         10       2450       1300       5000
      7839         10       5000       1300       5000
      7369         20        880        880       3000
      7876         20       1100        880       3000
      7566         20       2975        880       3000
      7788         20       3000        880       3000
      7902         20       3000        880       3000
      7900         30        950        950       2850
      7521         30       1250        950       2850
      7654         30       1250        950       2850
      7844         30       1500        950       2850
      7499         30       1600        950       2850
      7698         30       2850        950       2850
```

How each employee's salary compare with the average salary of the first year hires of their department?

```
SQL>SELECT empno, deptno, TO_CHAR(hiredate,'YYYY') HIRE_YR, sal,
TRUNC(AVG(sal) KEEP (DENSE_RANK FIRST ORDER BY TO_CHAR(hiredate,'YYYY') )
        OVER (PARTITION BY deptno)) AVG_SAL_YR1_HIRE
FROM emp
WHERE deptno IN (20, 10)
ORDER BY deptno, empno, HIRE_YR;
```



```
  EMPNO     DEPTNO HIRE        SAL AVG_SAL_YR1_HIRE
------- ---------- ---- ---------- ----------------
   7782         10 1981       2450             3725
   7839         10 1981       5000             3725
   7934         10 1982       1300             3725
   7369         20 1980        800              800
   7566         20 1981       2975              800
   7788         20 1982       3000              800
   7876         20 1983       1100              800
   7902         20 1981       3000              800
```

WINDOW clause :-

Some analytic functions (AVG, COUNT, FIRST_VALUE, LAST_VALUE, MAX, MIN and SUM among the ones we discussed) can take a window clause to further sub-partition the result and apply the analytic function. An important feature of the windowing clause is that it is dynamic in nature.

The general syntax of the <window_clause> is
[ROW or RANGE] BETWEEN <start_expr> AND <end_expr>
<start_expr> can be any one of the following
1.UNBOUNDED PECEDING
2. CURRENT ROW
3. <sql_expr> PRECEDING or FOLLOWING.
<end_expr> can be any one of the following
1.UNBOUNDED FOLLOWING or
2. CURRENT ROW or
3. <sql_expr> PRECEDING or FOLLOWING.

For ROW type windows the definition is in terms of row numbers before or after the current row. So for ROW type windows <sql_expr> must evaluate to a positive integer.

For RANGE type windows the definition is in terms of values before or after the current ORDER. We will take this up in details latter.

The ROW or RANGE window cannot appear together in one OVER clause. The window clause is defined in terms of the current row. But may or may not include the current row. The start point of the window and the end point of the window can finish before the current row or after the current row. Only start point cannot come after the end point of the window. In case any point of the window is undefined the default is UNBOUNDED PRECEDING for <start_expr> and UNBOUNDED FOLLOWING for <end_expr>.

If the end point is the current row, syntax only in terms of the start point can be can be
[ROW or RANGE] [<start_expr> PRECEDING or UNBOUNDED PRECEDING]

[ROW or RANGE] CURRENT ROW is also allowed but this is redundant. In this case the function behaves as a single-row function and acts only on the current row.

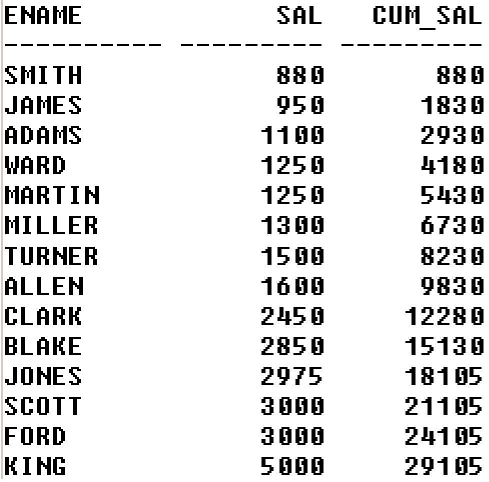
ROW Type Windows:-

For analytic functions with ROW type windows, the general syntax is:

Function( ) OVER (PARTITIN BY <expr1> ORDER BY <expr2,..> ROWS BETWEEN <start_expr> AND <end_expr>)
or
Function( ) OVER (PARTITON BY <expr1> ORDER BY <expr2,..> ROWS [<start_expr> PRECEDING or UNBOUNDED PRECEDING]

For ROW type windows the windowing clause is in terms of record numbers.

SQL>SELECT ename,sal,
 SUM(sal) over (order by sal ROWS UNBOUNDED PRECEDING)
 as cum_sal
from emp ;



```
ENAME             SAL    CUM_SAL
---------- ---------- ----------
SMITH             880        880
JAMES             950       1830
ADAMS            1100       2930
WARD             1250       4180
MARTIN           1250       5430
MILLER           1300       6730
TURNER           1500       8230
ALLEN            1600       9830
CLARK            2450      12280
BLAKE            2850      15130
JONES            2975      18105
SCOTT            3000      21105
FORD             3000      24105
KING             5000      29105
```

Optimizer Hints

Optimizer hints can be used with SQL statements to alter execution plans. This chapter explains how to use hints to force various approaches.

Hints let you make decisions usually made by the optimizer. As an application designer, you might know information about your data that the optimizer does not know. For example, you might know that a certain index is more selective for certain queries. Based on this information, you might be able to choose a more efficient execution plan than the optimizer. In such a case, use hints to force the optimizer to use the optimal execution plan.

You can use hints to specify the following:

Hints for Optimization Approaches and Goals
Access path
Join order
Join method
Parallelization

Hints apply only to the optimization of the statement block in which they appear. A statement block is any one of the following statements or parts of statements:

A simple SELECT, UPDATE, or DELETE statement.
A parent statement or subquery of a complex statement.
A part of a compound query.

Syntax:-

{DELETE|INSERT|SELECT|UPDATE} /*+ hint [text] [hint[text]]... */

Hints for Optimization Approached and Goals:-

The hints described in this section let you choose between the cost-based and the rule-based optimization approaches. With the cost-based approach, this also includes the goal of best throughput or best response time.
In ORACLE 10g rule-based optimization is obselete , but still supports for backward compatability.

ALL_ROWS
FIRST_ROWS(n)

ALL_ROWS :-

The ALL_ROWS hint explicitly chooses the cost-based approach to optimize a statement block with a goal of best throughput (that is, minimum total resource consumption).

```
SELECT /*+ ALL_ROWS */ employee_id, last_name, salary, job_id
FROM employees
WHERE employee_id = 7566;
```

FIRST_ROWS(n):-

The hints FIRST_ROWS(n) (where n is any positive integer) or FIRST_ROWS instruct Oracle to optimize an individual SQL statement for fast response. FIRST_ROWS(n) instructs Oracle to choose the plan that returns the first n rows most efficiently.

```
SELECT /*+ FIRST_ROWS(10) */ employee_id, last_name, salary, job_id
FROM employees
WHERE department_id = 20;
```

In this example each department contains many employees. The user wants the first 10 employees of department #20 to be displayed as quickly as possible.

The optimizer ignores this hint in DELETE and UPDATE statement blocks and in SELECT statement blocks that contain any of the following syntax:

Set operators (UNION, INTERSECT, MINUS, UNION ALL)
GROUP BY clause
FOR UPDATE clause
Aggregate functions
DISTINCT operator
ORDER BY clauses, when there is no index on the ordering columns

These statements cannot be optimized for best response time, because Oracle must retrieve all rows accessed by the statement before returning the first row. If you specify this hint in any of these statements, then the optimizer uses the cost-based approach and optimizes for best throughput.

Hints for Access Paths:-

FULL
ROWID
CLUSTER
HASH
INDEX
INDEX_ASC
INDEX_COMBINE
INDEX_JOIN
INDEX_DESC
INDEX_FFS
NO_INDEX
AND_EQUAL

Specifying one of these hints causes the optimizer to choose the specified access path only if the access path is available based on the existence of an index or cluster and on the syntactic constructs of the SQL statement. If a hint specifies an unavailable access path, then the optimizer ignores it.

FULL :-

The FULL hint explicitly chooses a full table scan for the specified table.

```
SELECT /*+ FULL(e) */ employee_id, last_name
  FROM employees e
```

WHERE last_name LIKE 'S%' ;

Oracle performs a full table scan on the employees table to execute this statement, even if there is an index on the last_name column that is made available by the condition in the WHERE clause.

INDEX hint:-

The INDEX hint explicitly chooses an index scan for the specified table. You can use the INDEX hint for domain, B-tree, bitmap, and bitmap join indexes. However, Oracle recommends using INDEX_COMBINE rather than INDEX for bitmap indexes, because it is a more versatile hint.

If hint specifies a single available index, then the optimizer performs a scan on this index. The optimizer does not consider a full table scan or a scan on another index on the table.

If hint specifies a list of available indexes, then the optimizer considers the cost of a scan on each index in the list and then performs the index scan with the lowest cost.

If hint specifies no indexes, then the optimizer considers the cost of a scan on each available index on the table and then performs the index scan with the lowest cost.

For example, consider this query that selects the name, job and salary all male employees

SQL>SELECT name, job,salary FROM employees WHERE sex = 'm';

Assume that there is an index on the SEX column and that this column contains the values m and f. If there are equal numbers of male and female employees, then the query returns a relatively large percentage of the table's rows, and a full table scan is likely to be faster than an index scan. However, if a very small percentage of male employees, then the query returns a relatively small percentage of the table's rows, and an index scan is likely to be faster than a full table scan.

If you know that the value in the WHERE clause of the query appears in a very small percentage of the rows, then you can use the INDEX hint to force the optimizer to choose an index scan. In this statement, the INDEX hint explicitly chooses an index scan on the sex_index, the index on the sex column:

```
SELECT /*+ INDEX(patients sex_index) use sex_index because there are few
  male patients  */ name, height, weight
FROM patients
WHERE sex = 'm';
```

INDEX_COMBINE hint:-

The INDEX_COMBINE hint explicitly chooses a bitmap access path for the table.

```
SELECT /*+INDEX_COMBINE(employees salary_bmi hire_date_bmi)*/ *
FROM employees
WHERE salary < 50000 AND hire_date < '01-JAN-1990';
```

JOIN ORDERS:-

The ORDERED hint causes Oracle to join tables in the order in which they appear in the FROM clause.

If you omit the ORDERED hint from a SQL statement performing a join, then the optimizer chooses the order in which to join the tables. You might want to use the ORDERED hint to specify a join order if you know something about the number of rows selected from each table that the optimizer does not. Such information lets you choose an inner and outer table better than the optimizer could.

The following query is an example of the use of the ORDERED hint:

```
SELECT  /*+ORDERED */ o.order_id, c.customer_id, l.unit_price * l.quantity
  FROM customers c, order_items l, orders o
 WHERE c.cust_last_name = :b1
   AND o.customer_id = c.customer_id
   AND o.order_id = l.order_id;
```

JOIN OPERATIONS:-

USE_NL
USE_MERGE
USE_HASH
DRIVING_SITE
LEADING
HASH_AJ, MERGE_AJ, and NL_AJ
HASH_SJ, MERGE_SJ, and NL_SJ

USE_NL hint :-

The USE_NL hint causes Oracle to join each specified table to another row source with a nested loops join, using the specified table as the inner table.

For example, consider this statement, which joins the accounts and customers tables. Assume that these tables are not stored together in a cluster:

```
SELECT accounts.balance, customers.last_name, customers.first_name
    FROM accounts, customers
    WHERE accounts.customer_id = customers.customer_id;
```

Because the default goal of the cost-based approach is best throughput, the optimizer chooses either a nested loops operation, a sort-merge operation, or a hash operation to join these tables, depending on which is likely to return all the rows selected by the query more quickly.

However, you might want to optimize the statement for best response time or the minimal elapsed time necessary to return the first row selected by the query, rather than best throughput. If so, then you can force the optimizer to choose a nested loops join by using the USE_NL hint. In this statement, the USE_NL hint explicitly chooses a nested loops join with the customers table as the inner table:

```
SELECT /*+ ORDERED USE_NL(customers) to get first row faster */
    accounts.balance, customers.last_name, customers.first_name
FROM accounts, customers
WHERE accounts.customer_id = customers.customer_id;
```

In many cases, a nested loops join returns the first row faster than a sort merge join. A nested loops join can return the first row after reading the first selected row from one table and the first matching row from the other and combining them, while a sort merge join cannot return the first row until after reading and sorting all selected rows of both tables and then combining the first rows of each sorted row source.

In the following statement where a nested loop is forced through a hint, orders is accessed through a full table scan and the filter condition l.order_id = h.order_id is applied to every row. For every row that meets the filter condition, order_items is accessed through the index order_id.

```
SELECT /*+ USE_NL(l h) */ h.customer_id, l.unit_price * l.quantity
  FROM orders h ,order_items l
 WHERE l.order_id = h.order_id;
```

USE_MERGE hint:-

The USE_MERGE hint causes Oracle to join each specified table with another row source, using a sort-merge join.

```
SELECT /*+USE_MERGE(employees departments)*/ *
FROM employees, departments
WHERE employees.department_id = departments.department_id;
```

USE_HASH hint:-

The USE_HASH hint causes Oracle to join each specified table with another row source, using a hash join.

```
SELECT /*+use_hash(employees departments)*/ *
FROM employees, departments
WHERE employees.department_id = departments.department_id;
```

PARALLEL

The PARALLEL hint lets you specify the desired number of concurrent servers that can be used for a parallel operation. The hint applies to the SELECT, INSERT, UPDATE, and DELETE portions of a statement, as well as to the table scan portion.

The number of servers that can be used is twice the value in the PARALLEL hint, if sorting or grouping operations also take place.If any parallel restrictions are violated, then the hint is ignored.

The PARALLEL hint must use the table alias, if an alias is specified in the query. The hint can then take two values, separated by commas after the table name. The first value specifies the degree of parallelism for the given table, and the second value specifies how the table is to be split among the Oracle Real Application Clusters instances. Specifying DEFAULT or no value signifies that the query coordinator should examine the settings of the initialization parameters to determine the default degree of parallelism. In the following example, the PARALLEL hint overrides the degree of parallelism specified in the employees table definition:

```
SELECT /*+ FULL(hr_emp) PARALLEL(hr_emp, 5) */ last_name
FROM hr.employees  hr_emp;
```

In the next example, the PARALLEL hint overrides the degree of parallelism specified in the employees table definition and tells the optimizer to use the default degree of parallelism determined by the initialization parameters. This hint also specifies that the table should be split among all of the available instances, with the of parallelism on each instance.

```
SELECT /*+ FULL(hr_emp) PARALLEL(hr_emp, DEFAULT,DEFAULT) */ last_name
FROM hr.employees hr_emp;
```

Data Pump

Oracle Data Pump (expdp and impdp) introduced in Oracle Database 10g is faster and more flexible alternative to the "exp" and "imp" utilities used in previous Oracle versions. In addition to basic import and export functionality data pump provides a PL/SQL API and support for external tables.

To use DATA PUMP utilities first we must create a directory object . The directory object is only a pointer to a physical directory, creating it does not actually create the physical directory on the file system.

```
SQL>CONN SYSTEM/MANAGER
SQL>CREATE OR REPLACE DIRECTORY test_dir AS 'C:\WINDOWS’
SQL>GRANT READ, WRITE ON DIRECTORY test_dir TO scott;
```

Existing directories can be queried using the **ALL_DIRECTORIES** view.

Table Exports/Imports :-

The TABLES parameter is used to specify the tables that are to be exported. The following is an example of the table export and import syntax and command should be executed at command prompt.

```
expdp scott/tiger@db10g tables=EMP,DEPT directory=TEST_DIR dumpfile=EMPDEPT.dmp
logfile=expdpEMP_DEPT.log
```

```
impdp scott/tiger@db10g tables=EMP,DEPT directory=TEST_DIR dumpfile=EMPDEPT.dmp
logfile=impdpEMP_DEPT.log
```

The TABLE_EXISTS_ACTION=APPEND parameter allows data to be imported into existing tables.

Schema Exports/Imports

The OWNER parameter of exp has been replaced by the SCHEMAS parameter which is used to specify the schemas to be exported. The following is an example of the schema export and import syntax.

```
expdp scott/tiger@db10g schemas=SCOTT directory=TEST_DIR dumpfile=SCOTT.dmp
logfile=expdpSCOTT.log
```

```
impdp scott/tiger@db10g schemas=SCOTT directory=TEST_DIR dumpfile=SCOTT.dmp
logfile=impdpSCOTT.log
```

Database Exports/Imports

The FULL parameter indicates that a complete database export is required. The following is an example of the full database export and import syntax.

```
expdp system/password@db10g full=Y directory=TEST_DIR dumpfile=DB10G.dmp
logfile=expdpDB10G.log
```

impdp system/password@db10g full=Y directory=TEST_DIR

INCLUDE and EXCLUDE options:-

The INCLUDE and EXCLUDE parameters can be used to limit the export/import to specific objects. When the INCLUDE parameter is used, only those objects specified by it will be included in the export/import. When the EXCLUDE parameter is used, all objects except those specified by it will be included in the export/import. The two parameters are mutually exclusive, so use the parameter that requires the least entries to give you the result you require. The basic syntax for both parameters is the same.

expdp scott/tiger@db10g schemas=SCOTT include=TABLE:"IN ('EMP', 'DEPT')" directory=TEST_DIR dumpfile=SCOTT.dmp logfile=expdpSCOTT.log

expdp scott/tiger@db10g schemas=SCOTT exclude=TABLE:"= 'BONUS'" directory=TEST_DIR dumpfile=SCOTT.dmp logfile=expdpSCOTT.log

Regular Expressions

The database provides a set of SQL functions that allow you to search and manipulate strings using regular expressions. You can use these functions on any datatype that holds character data such as CHAR, NCHAR, CLOB, NCLOB, NVARCHAR2, and VARCHAR2

A regular expression must be enclosed or wrapped between single quotes. Doing so, ensures that the entire expression is interpreted by the SQL function and can improve the readability of your code.

Regualar Expression Functions:-

REGEXP_LIKE

This function searches a character column for a pattern. Use this function in the WHERE clause of a query to return rows matching the regular expression you specify.

REGEXP_REPLACE

This function searches for a pattern in a character column and replaces each occurrence of that pattern with the pattern you specify.

REGEXP_INSTR

This function searches a string for a given occurrence of a regular expression pattern. You specify which occurrence you want to find and the start position to search from. This function returns an integer indicating the position in the string where the match is found.

REGEXP_SUBSTR

This function returns the actual substring matching the regular expression pattern you specify.

Metacharacters Supported in Regular Expressions:-

Operator Name	Description
.	Any Character
+	One or More
?	Zero or One
*	Zero or more
{m}	Interval--Exact Count
{m,}	Interval--At Least Count
{m,n}	Interval--Between Count
[...]	Matching Character List
[^ ...]	Non-Matching Character List
|	Or
(...)	Subexpression or Grouping
\n	Backreference
\	Escape Character

^	**Beginning of Line Anchor**
$	**End of Line Anchor**
[:class:]	**POSIX Character Class**
[.element.]	**POSIX Collating Sequence**
[=character=]	**POSIX Character Equivalence Class**

Constructing Regular Expressions:-

As mentioned earlier, regular expressions are constructed using metacharacters and literals. Metacharacters that operate on a single literal, such as '+' and '?' can also operate on a sequence of literals or on a whole expression. To do so, you use the grouping operator to enclose the sequence or subexpression.

Match Any Character—Dot:-

The dot operator '.' matches any single character in the current character set. For example, to find the sequence--'a', followed by any character, followed by 'c'--use the expression:

a.c

This expression matches all of the following sequences:

abc
adc
a1c
a&c

The expression does not match:

abb

One or More—Plus:-

The one or more operator '+' matches one or more occurrences of the preceding expression. For example, to find one or more occurrences of the character 'a', you use the regular expression:

a+

This expression matches all of the following:

a
aa
aaa

The expression does not match:

bbb

Zero or One--Question Mark Operator:-

The question mark matches zero or one--and only one--occurrence of the preceding character or subexpression. You can think of this operator as specifying an expression that is optional in the source text. For example, to find--'a', optionally followed by 'b', then followed by 'c'--you use the following regular expression:

ab?c

This expression matches:

abc
ac

The expression does not match:

adc

abbc

Zero or More—Star:-

The zero or more operator '*', matches zero or more occurrences of the preceding character or subexpression. For example, to find--'a', followed by zero or more occurrences of 'b', then followed by 'c'--use the regular expression:

ab*c
This expression matches all of the following sequences:
ac
abc
abbc
abbbbc

The expression does not match:
adc

Interval--Exact Count:-

The exact-count interval operator is specified with a single digit enclosed in braces. You use this operator to search for an exact number of occurrences of the preceding character or subexpression.

For example, to find where 'a' occurs exactly 5 times, you specify the regular expression:
a{5}
This expression matches:
aaaaa
The expression does not match:
aaaa

Interval--At Least Count:-

You use the at-least-count interval operator to search for a specified number of occurrences, or more, of the preceding character or subexpression. For example, to find where 'a' occurs at least 3 times, you use the regular expression:

a{3,}
This expression matches all of the following:
aaa
aaaaa

The expression does not match:
aa

Interval--Between Count:-

You use the between-count interval operator to search for a number of occurrences within a specified range. For example, to find where 'a' occurs at least 3 times and no more than 5 times, you use the following regular expression:

a{3,5}
This expression matches all of the following sequences:
aaa

aaaa
aaaaa

The expression does not match:
aa

Matching Character List:-

You use the matching character list to search for an occurrence of any character in a list. For example, to find either 'a', 'b', or 'c' use the following regular expression:

[abc]

This expression matches the first character in each of the following strings:

at
bet
cot

The expression does not match:

def

The following regular expression operators are allowed within the character list, any other metacharacters included in a character list lose their special meaning (are treated as literals):

Range operator '-'
POSIX character class [::]
POSIX collating sequence [. .]
POSIX character equivalence class [= =]
Non-Matching Character List

Non-matching character list :-
used to specify characters that you do not want to match. Characters that are not in the non-matching character list are returned as a match. For example, to exclude the characters 'a', 'b', and 'c' from your search results, use the following regular expression:

[^abc]

This expression matches
abcdef
ghi
The expression does not match:
abc

As with the matching character list, the following regular expression operators are allowed within the non-matching character list (any other metacharacters included in a character list are ignored):

For example, the following regular expression excludes any character between 'a' and 'i' from the search result:

[^a-i]
This expression matches the characters 'j' and 'l' in the following strings:

hijk
lmn

The expression does not match the characters:

abcdefghi
Or
Use the Or operator '|' to specify an alternate expression. For example to match 'a' or 'b', use the following regular expression:

a|b

Subexpression:-

You can use the subexpression operator to group characters that you want to find as a string or to create a complex expression. For example, to find the optional string 'abc', followed by 'def', use the following regular expression:

(abc)?def
This expression matches strings 'abcdef' and 'def' in the following strings:
abcdefghi
defghi

The expression does not match the string:
ghi
Backreference:-

The backreference lets you search for a repeated expression. You specify a backreference with '\n', where n is an integer from 1 to 9 indicating the nth preceding subexpression in your regular expression.

For example, to find a repeated occurrence of either string 'abc' or 'def', use the following regular expression:

(abc|def)\1
This expression matches the following strings:
abcabc
defdef
The expression does not match the following strings:
abcdef
abc

Escape Character:-

Use the escape character '\' to search for a character that is normally treated as a metacharacter. For example to search for the '+' character, use the following regular expression:

\+

This expression matches the plus character '+' in the following string:
abc+def
The expression does not match any characters in the string:
abcdef

Beginning of Line Anchor:-

Use the beginning of line anchor ^ to search for an expression that occurs only at the beginning of a line. For example, to find an occurrence of the string def at the beginning of a line, use the expression:

^def

This expression matches def in the string:
defghi
The expression does not match def in the following string:
abcdef

End of Line Anchor:-

The end of line anchor metacharacter '$' lets you search for an expression that occurs only at the end of a line. For example, to find an occurrence of def that occurs at the end of a line, use the following expression:

def$

This expression matches def in the string:
abcdef
The expression does not match def in the following string:
defghi

POSIX Character Class:-

[:alnum:]	Alphanumeric characters
[:alpha:]	Any alphabet either upper or lower case.
[:cntrl:]	Any control character. A non-printable character is called as control character.
[:digit:]	Any digit.
[:lower:]	Any lower case letter.
[:print:]	Any printable character.
[:punct:]	Any punctuation character.
[:space:]	All space characters.
[:upper:]	Any upper case letter.

Examples:-

SQL>SELECT ename FROM emp WHERE regexp_like (ename, '^S.*T$');

The above example displays enames that start with S and end with T. It may contain anything in between these two.

If you want to modify the way REGEXP_LIKE compares characters then third parameter, which contains either 'c' for case sensitive or 'i' for ignore case, can be given as shown below.

SQL>SELECT ename FROM emp WHERE regexp_like (ename, '^S.*T$','i')

Now let us see how REGEXP_SUBSTR is used to extract a substring based on regular expression.

SQL>SELECT regexp_substr('Oracle Database 10g is first grid aware database','[0-9]+') version FROM DUAL;

The above query displays 10 as it is consisting of one or more digits.

SQL>SELECT regexp_substr('Oracle Database 10g is first grid aware database','[0-9]+[a-z]') version FROM DUAL;

The following query displays the starting position of one or more digits.
SQL>SELECT regexp_instr('Oracle Database 10g is first grid aware database','[0-9]+') position FROM DUAL;

The following query returns the positon of first non-alphabet in the given string.
SQL>SELECT regexp_instr('Abc123 xyz123','[^[:alpha:]]') FROM DUAL;
The following query places a space between Oracle its version using REGEXP_REPLACE function. For example, Oracle9i will become Oracle 9i, Oracle10g will become Oracle 10g.
SQL>SELECT regexp_replace('Oracle10g','([[:alpha:]]+)([[:digit:]]+.)','\1 \2') FROM DUAL;
We extract series of alphabets and take them as group 1. Then we are looking for a group of digits followed by any character and treat it as group 2. Then we replace the original with \1 (group 1) a space and \2 (group 2).

XML

Generating XML data from ORACLE database :-

using DBMS_XMLGEN package :-

the above package contains a function called **GETXML** that takes **SELECT QUERY** and displays query result in **XML** format.

Example:-

```
SQL> SELECT DBMS_XMLGEN.getXML('SELECT * FROM DEPT') FROM DUAL;
```

Output :-

```
<?xml version="1.0"?>
<ROWSET>
   <ROW>
     <DEPTNO>10</DEPTNO>
     <DNAME>ACCOUNTING</DNAME>
     <LOC>NEW YORK</LOC>
   </ROW>
  <ROW>
     <DEPTNO>20</DEPTNO>
     <DNAME>RESEARCH</DNAME>
     <LOC>DALLAS</LOC>
  </ROW>
  <ROW>
     <DEPTNO>30</DEPTNO>
     <DNAME>SALES</DNAME>
     <LOC>CHICAGO</LOC>
  </ROW>
 <ROW>
    <DEPTNO>40</DEPTNO>
    <DNAME>OPERATIONS</DNAME>
    <LOC>BOSTON</LOC>
  </ROW>
</ROWSET>
```

Using SQL/XML functions :-

XMLELEMENT and XMLATTRIBUTES SQL Functions:-

You use SQL function XMLElement to construct XML instances from relational data. It takes as arguments an element name, an optional collection of attributes for the element, and zero or more additional arguments that make up the element content. It returns an XMLType instance.

```
SQL>SELECT XMLElement("Date", hiredate)
  FROM emp
  WHERE empno=7844 ;
```

Output :-

```
                <Date>1994-06-07</Date>
```

```
SQL>SELECT XMLElement("Date", to_char(hiredate))
  FROM emp
  WHERE empno=7844 ;
```

Output :-

```
                <Date>07-JUN-1994</Date>
```

```
SQL>SELECT  empno,XMLELEMENT("Emp",ename) as Result         FROM emp
```

```
        EMPNO           RESULT
          7369           <Emp>SMITH</Emp>

          7499           <Emp>ALLEN</Emp>

          7521          <Emp>WARD</Emp>

          7566          <Emp>JONES</Emp>

          7654           <Emp>MARTIN</Emp>

         7698           <Emp>BLAKE</Emp>

         7782           <Emp>CLARK</Emp>

         7788          <Emp>SCOTT</Emp>

         7839          <Emp>KING</Emp>

         7844           <Emp>TURNER</Emp>

         7876          <Emp>ADAMS</Emp>

         7900          <Emp>JAMES</Emp>

         7902          <Emp>FORD</Emp>

         7934          <Emp>MILLER</Emp>
```

Generating Nested XML :-

```
SQL> SELECT XMLElement("Emp",
  2                        XMLElement("name", ename),
  3                        XMLElement("hiredate", hiredate)) AS "RESULT"
  4  FROM emp;
```

```
RESULT
<Emp><name>SMITH</name><hiredate>1980-12-17</hiredate></Emp>
<Emp><name>ALLEN</name><hiredate>1981-02-20</hiredate></Emp>
<Emp><name>WARD</name><hiredate>1981-02-22</hiredate></Emp>
<Emp><name>JONES</name><hiredate>1981-04-02</hiredate></Emp>
<Emp><name>MARTIN</name><hiredate>1981-09-28</hiredate></Emp>
<Emp><name>BLAKE</name><hiredate>1981-05-01</hiredate></Emp>
<Emp><name>CLARK</name><hiredate>1981-06-09</hiredate></Emp>
<Emp><name>SCOTT</name><hiredate>1982-12-09</hiredate></Emp>
<Emp><name>KING</name><hiredate>1981-11-17</hiredate></Emp>
<Emp><name>TURNER</name><hiredate>1981-09-08</hiredate></Emp>
<Emp><name>ADAMS</name><hiredate>1983-01-12</hiredate></Emp>
<Emp><name>JAMES</name><hiredate>1981-12-03</hiredate></Emp>
<Emp><name>FORD</name><hiredate>1981-12-03</hiredate></Emp>
<Emp><name>MILLER</name><hiredate>1982-01-23</hiredate></Emp>
```

Using XMLATTRIBUTES :-

```
SQL> SELECT XMLElement("Emp",
                              XMLAttributes( empno as "ID", ename AS "name"))
  AS "RESULT"
  FROM emp
  WHERE empno in (7369,7566,7844,7902);
```

RESULT :-

```
<Emp ID="7369" name="SMITH"></Emp>
<Emp ID="7566" name="JONES"></Emp>
<Emp ID="7844" name="TURNER"></Emp>
<Emp ID="7902" name="FORD"></Emp>
```

XMLFOREST SQL Function

SQL function XMLForest produces a forest of XML elements from its arguments, which are expressions to be evaluated, with optional aliases.

```
SQL> SELECT XMLElement("Emp",
                                      XMLForest(ename as "Ename",
                                                    sal as "Salary",
                                                    deptno as "Department"))
  AS "RESULT"
  FROM emp WHERE deptno = 20;
```

Output :-

```
<Emp><Ename>SMITH</Ename><Salary>800</Salary><Department>20</Department></Emp>
<Emp><Ename>JONES</Ename><Salary>2975</Salary><Department>20</Department></Emp>
<Emp><Ename>SCOTT</Ename><Salary>3000</Salary><Department>20</Department></Emp>
<Emp><Ename>ADAMS</Ename><Salary>1100</Salary><Department>20</Department></Emp>
<Emp><Ename>FORD</Ename><Salary>3000</Salary><Department>20</Department></Emp>
```

Inserting XML data into ORACLE DB :-

Consider the following XMLType table containing an XML document with employee information:

```
SQL>CREATE TABLE emp_xml_tab OF XMLType;
Table created.
```

Method 1 :-

```
SQL>INSERT INTO emp_xml_tab VALUES(
        XMLType('<EMPLOYEES>
                    <EMP>
                        <EMPNO>112</EMPNO>
                        <EMPNAME>Joe</EMPNAME>
                        <SALARY>50000</SALARY>
                  </EMP>
                   <EMP>
                        <EMPNO>217</EMPNO>
                        <EMPNAME>Jane</EMPNAME>
                        <SALARY>60000</SALARY>
                    </EMP>
                     <EMP>
                        <EMPNO>412</EMPNO>
                        <EMPNAME>Jack</EMPNAME>
                        <SALARY>40000</SALARY>
                     </EMP>
                     </EMPLOYEES>'));
```

1 Row Created

Method 2:-

```
SQL>INSERT INTO emp_xml VALUES(
        XMLType(BFILENAME('XMLDIR','emp.xml'),  nls_charset_id('AL32UTF8')));
```

XMLDIR is a Directory Object