# Structure Query Language:-

* SQL is a structure query language using SQL user can ACCESS, RETRIVE, STORE & MANAPULATE THE DATA IN DATABASE.

|  |  |
| --- | --- |
| LANGUAGE | SQL, PLSQL, |
| TOOL | SQLPLUS, Isqlplus, Toad, SQLDeveloper, |
|  |  |

**SQL CLAUSES:-**

* Basically we use them to apply filters for queries.

|  |  |
| --- | --- |
| **Name** | **Examples** |
| CLAUSE | FROM, SELECT, WHERE, HAVING, ORDER BY, DISTINCT |
| SAPRATION OPERATOR | **,** (COMMA), semicolon |
| ALL COLUMNS | **\*** |

ORACLE SEARCH Procedure:-

Step1: SEARCH ORACLE KEYWORD(

**FROM → WHERE → GROUP BY → HAVING → SELECT → ORDER BY**

) -> SYNTACTIC CHECKING.rb

**STEP2:** USER INFORMATION (TABLE NAME, COLUMN NAME)-> SEMANTIC CHECKING.

**STEP3:** IT WILL GO TO DATABASE -> TABLE FULL SCAN.

**STEP4:** COST & cordiality -> %CPU UTILIZATION.

### Important Notes

* **JOINs**: If multiple tables are involved, joins are resolved during the FROM clause.
* **Subqueries**: Subqueries are executed first, and their results are used in the main query.
* **Aliases**: Columns or expressions aliased in SELECT cannot be used in WHERE or GROUP BY but can be used in ORDER BY.

### Cardinality: -

**Definition**: Cardinality is the estimated number of rows that the database expects to return.

#### ****Example****:

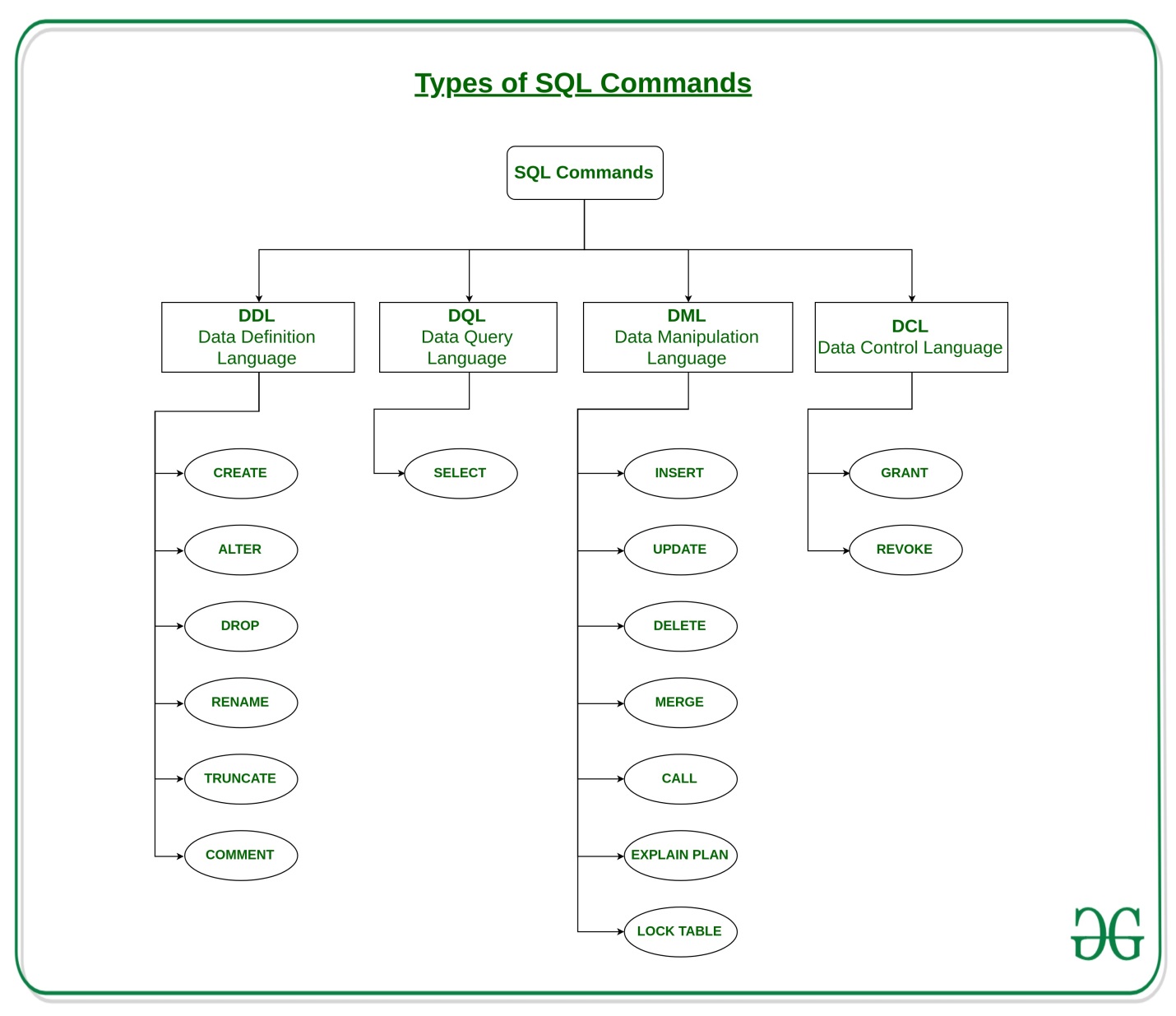
For a table with 10,000 rows, if a query includes a filter (WHERE clause) that is expected to match 500 rows, the **cardinality** of that step would be **500**.

Cost:-

**Definition**: Cost is a numerical estimate of the computational resources required to execute a query step

**Types of SQL:-**

**DQL, DML, DDL, DCL, TCL : -**



## DQL (Data query Language):-

* DQL commands are basically SELECT statements.
* SELECT statements let you query the database to find information in one or more tables, and return the query as a result set.

Select how many tables in the database:

SELECT \* FROM USER\_TABLES

SELECT \* FROM USER\_TAB\_COLUMNS

**Select all column in table :-**

SELECT \* FROM EMPLOYEES;

**To see Description of the table:-**

DESC EMPLOYEES;

**Select custom column in table:-**

SELECT FIRST\_NAME, LAST\_NAME, EMP\_ID FROM EMPLOYEES;

**Select Using where Condition (Selecting particular condition) :-**

SELECT EMPLOYEE\_ID, FIRST\_NAME, SALARY, DEPARTMENT\_ID FROM EMPLOYEES WHERE Employee\_ID = 100;

**Select Using Order By function:-**

**Order by clause always comes last only**

SELECT EMP\_ID, FIRST\_NAME, SALARY, DEPARTMENT\_ID FROM EMPLOYEES WHERE EMP\_ID

order by SALARY DESC

FETCH FIRST 2 ROWS ONLY; --- ITS fetch 2 row

SELECT EMP\_ID, FIRST\_NAME, SALARY, DEPARTMENT\_ID

FROM EMPLOYEES

WHERE EMP\_ID

order by SALARY ASC

FETCH FIRST 1 ROWS ONLY;

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

FROM EMPLOYEES

**OFFSET 2 ROWS FETCH NEXT 1 ROWS ONLY**;

## OPERATORS:-

SELECT \* FROM WORKER WHERE FIRST\_NAME in ('Vipul', 'Satish');

SELECT \* FROM WORKER WHERE FIRST\_NAME NOT IN ('Vipul', 'Satish');

It finish with a

SELECT \* FROM EMPLOYEES WHERE FIRST\_NAME LIKE '%a';

It start with A

SELECT \* FROM EMPLOYEES WHERE FIRST\_NAME LIKE 'A%';

SELECT \* FROM EMPLOYEES WHERE FIRST\_NAME NOT LIKE 'A%';

SELECT \* FROM EMPLOYEES WHERE COMMISSION\_PCT IS NULL;

SELECT \* FROM EMPLOYEES WHERE COMMISSION\_PCT IS NOT NULL;

Greater operator:-

SELECT \* FROM EMPLOYEES WHERE SALARY >1000;

SELECT \* FROM EMPLOYEES WHERE SALARY >=1000;

Less then operator:-

SELECT \* FROM EMPLOYEES WHERE SALARY <1000;

SELECT \* FROM EMPLOYEES WHERE SALARY <=1000;

SELECT EMP\_ID, FIRST\_NAME, SALARY, DEPARTMENT\_ID FROM EMPLOYEES WHERE EMP\_ID between 100 AND 110;

SELECT EMP\_ID, FIRST\_NAME, SALARY, DEPARTMENT\_ID FROM EMPLOYEES WHERE EMP\_ID Not between 100 AND 110;

## DDL (Data Definition Language) :-

### CREATE TABLE :-

We know that a table comprises of rows and columns. So while creating tables we have to provide all the information to SQL about the names of the columns, type of data to be stored in columns, size of the data etc. Let us now dive into details on how to use CREATE TABLE statement to create tables in SQL.

CREATE DATABASE DATABASE\_NAME;

CREATE TABLE TABLE1(ID NUMBER(10), NAME VARCHAR2(20), DOB DATE);

### ALTER TABLE:-

* Multiple columns can also be modified at once.
* We cannot modify number to varchar & varchar to number, only size of the variable can change that too increase not decrease.

ALTER TABLE T1 ADD(MOBILE NUMBER(10));

ALTER TABLE T1 DROP(DOB);

ALTER TABLE T1 MODIFY (MOBILE VARCHAR2(20));

ALTER TABLE T1 RENAME COLUMN MOBILE TO EMAIL;

### RENAME TABLE:-

RENAME T1 TO TABLE1;

### TRUNCATE TABLE:-

TRUNCATE TABLE TABLE1;

### DROP TABLE:-

DROP TABLE TABLE1;

## delete and truncate and drop difference:-

|  |  |  |
| --- | --- | --- |
| **Delete** | **Truncate** | **Drop** |
| * DML command. | * DDL command. | * DDL command. |
| * Delete some or all row from the table. | * Delete all row from the table. | * It removes all the indexes,privilleges,rows and frees the memory space for other objects. |
| * Use UNDO spaces. | * Don’t use UNDO Table space | * Don’t use UNDO table space. |
| * Released blocks that go to the freelist for the table, to be used for subsequent inserts/updates. Does not deallocate space. | * Deallocates all space used by the table except MINEXTENTS. | * Unless the PURGE clause is specified, does not result in space being released. |
| * Can delete data even if FKs are enabled. | * Cant TRUNCATE data FK need to disable / droped. | * Can drop the table with CASCADE CONSTRAINTS option. This will remove the associate FKs. |
| * Uncommitted DELETE can be roll back. | * Can’t be ROLL BACK. | * CAN BE roll back. |
| * Required commit. | * Need need commit after delete. | * No need commit require. |
| * Can use with triggers. | * No trigger Fired. | * No trigger fired. |
| * DELETE privilege on a specific table can be granted to another user or role. | * TRUNCATE privilege on a specific table cannot be granted to another user or role. | * DROP ANY privilege on a specific table cannot be granted to another user or role. |
| * Yes, as long as the user has the DELETE privilege on the object. | * No. A table can be truncated in one’s own schema only. | * No. A table can be dropped in one’s own schema only. |
| * Can work on a table that is part of a cluster | * No. You will have to truncate the whole cluster, or use either DELETE or DROP. | * Can work on a table that is part of a cluster |

## DML (DATA MANAPULATION LANGUAGE) :-

### INSERT :-

**TYPE 1 :-**

INSERT INTO

T1(ID, NAME, ADDRESS)

VALUES(1,’RAMESH’, ‘CHENNAI’);

**TYPE 2:-**

INSERT INTO T1 VALUES(1,’RAMESH’,‘CHENNAI’);

**TYPE 3:-**

INSERT ALL

INTO T1(ID,NAME,ADDRESS) VALUES(1,'K','AV')

INTO T1(ID,NAME,ADDRESS) VALUES(2,'M','AT')

INTO T1(ID,NAME,ADDRESS) VALUES(3,'R','AVT')

SELECT \* FROM DUAL;

### UPDATE VALUES :-

UPDATE T1

SET DEPARTMENT=’NON TRADE’

WHERE EMPID = 5986;

### MULTIPLE COLUMN UPDATE:-

UPDATE STUDENTS SET

student\_name = 'kumar',

date\_of\_birth = TO\_DATE('11-December-1992', 'DD-MON-YY'),

email = 'scott@tiger.com',

city = 'Chennai'

WHERE STUDENT\_ID = 100;

### DELETE VALUES :-

DELETE FROM EMP WHERE EMP\_ID=1265;

### MERGE TABLE:-

* The merge statement selects data from one or more source tables and updates or inserts it into a target table.

MERGE INTO TEMP\_TABLE /\*TARGET TABLE\*/

USING MASTER\_TABLE /\*SOURCE TABLE\*/

ON (TEMP.ID=MASTER.ID)

WHEN MATCHED THEN

UPDATE SET TEMP.DEPARTMENT=MASTER.DEPARTMENT, TEMP.LOCATION=MASTER.LOCATION

WHEN NOT MATCHED THEN

INSERT VALUES (TEMP.ID, TEMP.DEPART, TEMP.LOCATION);

* It will copy master to temp table.

## TCL (Transaction Control Language):-

Commit;-> save the transaction into data file from buffer cache.

SAVEPOINT A -> Temporarily save the transaction that you can roll back to that point whenever you want before commit.

ROLL BACK A; -> This command restores the database to last committed state.

It is also used with savepoint command to jump to a savepoint in a transaction.

# SINGLE ROW FUNCTION:-

These functions require one or more input arguments and operate on each row, thereby returning one output value for each row. Argument can be a column, literal or an expression. Single row functions can be used in SELECT statement, WHERE and ORDER BY clause.

## Character Function:-

Handling string function in the query.

### SUBSTR:-

Example: Find Print The First Three Characters Of FIRST\_NAME From Worker Table;

Select SUBSTR(first\_name, 1,3) from EMPLOYEES;

SELECT

FIRST\_NAME,

SUBSTR(FIRST\_NAME,-2)

FROM EMPLOYEES

WHERE ROWNUM <10;

The result will be last from 2 character.

FIRST\_NAME SU

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

Ellen en

Sundar ar

Mozhe he

David id

Hermann nn

Shelli li

Amit it

Elizabeth th

Sarah ah

### INSTR:-

Example:-v Find The Position Of The Alphabet (‘A’) In The First Name

SELECT INSTR(FIRST\_NAME, ‘a’) from EMPLOYEES;

FIND how many ‘A’ is in first\_name

SELECT

FIRST\_NAME,

(length(FIRST\_NAME) - LENGTH(replace(HIGHER(first\_NAME),'A',''))) HOW\_MANY\_L

FROM EMPLOYEES;

### TRIM:-

* To remove BOTH side SPACE values.

SELECT TRIM(FIRST\_NAME) FROM EMPLOYEES;

Trim only remove first and last character.

select

FIRST\_NAME,

TRIM('a' FROM FIRST\_NAME)

FROM EMPLOYEES;

Output:-

FIRST\_NAME TRIM('A'FROMFIRST\_NA

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

Jose Manuel Jose Manuel

Peter Peter

Clara Clar

Shanta Shant

Alana Alan

Matthew Matthew

Jennifer Jennifer

Eleni Eleni

### RTRIM:-

* To remove right side arguments values.

Example:- To remove N in first name column in right side.

SELECT RTRIM(FIRST\_NAME, 'N') from WORKER;

### LTRIM:-

* To remove LEFT side given arguments values.

Example:- To remove N in first name column in right side.

SELECT LTRIM(FIRST\_NAME, 'A') from WORKER;

SELECT LTRIM('\*\*RA\*\*JE\*\*SH\*\*','\*'), RTRIM('\*\*RA\*\*JE\*\*SH\*\*','\*') FROM DUAL;

L\_TRIM R\_TRIM

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

RA\*\*JE\*\*SH\*\* \*\*RA\*\*JE\*\*SH

Below statement is wrong

SELECT TRIM('\*\*RA\*\*JE\*\*SH\*\*','\*') FROM DUAL;

### LPAD:-

* To Add Left side arguments values.

Example:- To add N in first name column in right side.

SELECT LPAD(FIRST\_NAME, '10', '0') from EMPLOYEES;

### LENGTH:-

* Find length of the string.

Example:- find the unique value of the department & there lengths

SELECT LENGTH(DEPARTMENT), department FROM WORKER;

Find how many A or a letters are there in the columns:-

SELECT FIRST\_NAME,

(LENGTH(UPPER(FIRST\_NAME)) - LENGTH(REPLACE(UPPER(FIRST\_NAME), 'A', ''))) AS count\_of\_A

FROM EMPLOYEES;

### CONCAT:-

* combain two words or string.

SELECT CONCAT(FIRST\_NAME,LAST\_NAME) FROM EMPLOYEES;

SELECT FIRST\_NAME||'SENTHIL'|| PHONE\_NUMBER||SALARY FROM EMPLOYEES;

### CASE MANIPULULICATION:-

#### UPPER

SELECT UPPER(FIRST\_NAME) FROM EMPLOYEES;

#### LOWER

SELECT LOWER(FIRST\_NAME) FROM EMPLOYEES;

#### INITCAP

SELECT INITCAP(FIRST\_NAME) FROM EMPLOYEES;

### TRANSLATE & REPLACE:-

* Translate will change each & every variable.
* Replace will only change given EXACT variable.

SELECT

REPLACE ('INDIA', 'IN','XY'), TRANSLATE('INDIA', 'IN','XY')

FROM DUAL;

SELECT REPLACE(FIRST\_NAME, 'a', 'A') FROM WORKER;

In above query all small a will replace A

|  |  |
| --- | --- |
| **TRANSLATE('INDIA','IN','XY')** | **REPLACE('INDIA','IN','XY')** |
| XYDXA | XYDIA |

SELECT

TRANSLATE('INDIA', 'I','XY'),

REPLACE ('INDIA', 'I','XY')

FROM DUAL;

TRANS REPLACE

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

XNDXA XYNDXYA

SELECT TRANSLATE('123456789', '598','#@'),

REPLACE ('123456789', '54','@@') FROM DUAL;

|  |  |
| --- | --- |
| **TRANSLATE('123456789','59','#@')** | **REPLACE('123456789','54','@@')** |
| 1234#67@ | 123456789 |

SELECT REPLACE('INDIA','I','57'), TRANSLATE('INDIA','IN','57') FROM DUAL;



## Regular Expression :-

**Key Features of Regular Expressions in Oracle**

1. **Pattern Matching**: Search for strings matching a specific pattern.
2. **Extracting Substrings**: Extract specific parts of a string that match a pattern.
3. **Replacing Strings**: Replace parts of a string based on a pattern.
4. **Splitting Strings**: Break strings into parts using a pattern as the delimiter.

### 1. REGEXP\_LIKE:-

**Purpose**: Matches a string against a regular expression pattern (similar to LIKE, but more powerful).

**Example**: Find employees whose names start with J and end with n:

SELECT \*

FROM EMPLOYEES

WHERE REGEXP\_LIKE(NAME, '^J.\*n$');

### 2. REGEXP\_SUBSTR

**Purpose**: Extracts the substring that matches a regular expression.

**SELECT REGEXP\_SUBSTR('Address 123 Main St.', '\d+', 1, 1) AS first\_number**

**FROM DUAL;**

Extract the domain (e.g., gmail.com) from an email address:

**SELECT REGEXP\_SUBSTR(EMAIL, '@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}', 1, 1) AS domain**

**FROM USERS;**

### 3. REGEXP\_REPLACE

 **Purpose**: Replaces substrings matching a pattern with another string.

**Example**: Replace all non-numeric characters in a string with an empty string

**SELECT REGEXP\_REPLACE('Phone: (123) 456-7890', '[^0-9]', '') AS digits\_only**

**FROM DUAL;**

Replace Multiple Spaces with a Single Space

**SELECT REGEXP\_REPLACE('This is a test.', '\s+', ' ') AS normalized\_text**

**FROM DUAL;**

### 4. REGEXP\_INSTR

**Purpose**: Returns the starting position of a substring that matches a pattern.

**Example**: Find the starting position of the first numeric sequence in a string

**SELECT REGEXP\_INSTR('ID: ABC123XYZ', '\d+', 1, 1) AS position**

**FROM DUAL;**

**Regex Patterns and Metacharacters**

Regular expressions use special characters (metacharacters) to define patterns.

|  |  |  |
| --- | --- | --- |
| **Metacharacter** | **Description** | **Example** |
| . | Matches any single character. | A.B matches ACB, A9B. |
| \* | Matches 0 or more occurrences of the preceding character. | A\* matches A, AA, AAA. |
| + | Matches 1 or more occurrences of the preceding character. | A+ matches A, AA, AAA. |
| ? | Matches 0 or 1 occurrence of the preceding character. | AB?C matches AC, ABC. |
| ^ | Matches the beginning of the string. | ^A matches strings starting with A. |
| $ | Matches the end of the string. | Z$ matches strings ending with Z. |
| [ ] | Matches any one character in the set. | [A-Z] matches any uppercase letter. |
| [^ ] | Matches any character not in the set. | [^0-9] matches non-numeric characters. |
| {n,m} | Matches between n and m occurrences. | A{2,4} matches AA, AAA, AAAA. |
| \d | Matches any digit (equivalent to [0-9]). | \d+ matches 123. |
| \w | Matches any word character (letters, digits, or \_). | \w+ matches hello123. |
| \s | Matches any whitespace character. | \s matches spaces, tabs, etc. |

SELECT HIRE\_DATE, TRANSLATE(HIRE\_DATE,'-0123456789', ' ')AS ALPHA, TRANSLATE(HIRE\_DATE,'ABCDEFGHIJKLMNOPQRSTUVWXYZ-', ' ')AS "NUMBER" FROM EMPLOYEES;

**To separate each character:-**

select country\_name, REGEXP\_REPLACE(country\_name, '(.)', '\1 ') from countries

|  |  |
| --- | --- |
| **COUNTRY\_NAME** | **REGEXP\_REPLACE(COUNTRY\_NAME,'(.)','\1')** |
| Argentina | A r g e n t i n a |
| Australia | A u s t r a l i a |
| Belgium | B e l g i u m |
| Brazil | B r a z i l |

To Remove 2 or more space from the gap :-

SELECT REGEXP\_REPLACE('the web development tutorial w3resource.com', '( ){2,}', ' ') "REGEXP\_REPLACE" FROM DUAL;

Sample Output:

REGEXP\_REPLACE

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

the web development tutorial w3resource.com

SELECT HIRE\_DATE, REGEXP\_REPLACE(HIRE\_DATE, '[^A-Z]', '') ALPHA from EMPLOYEES;

|  |  |
| --- | --- |
| **HIRE\_DATE** | **ALPHA** |
| 17-JUN-87 | JUN |
| 21-SEP-89 | SEP |
| 13-JAN-93 | JAN |
| 03-JAN-90 | JAN |
| 21-MAY-91 | MAY |

**SPECIAL CHAR REMOVE:-**

^ in this setup it allows only numeric and character other then that it will skip.

REGEXP\_REPLACE(TAC\_RTA\_CODE, '[^0-9A-Za-z]', '')

## Number Function:-

### ROUND() :-

SELECT ROUND (10.1234), ROUND(10.7456), ROUND(10.2,1), ROUND(10.2876,-1), ROUND(10.2876,-2), ROUND(10.7,1),ROUND(10.7,-1) from dual;



### TRUNC() :-

SELECT TRUNC(10.2), TRUNC(10.7), TRUNC(10.2,1), TRUNC(10.2,-1), TRUNC(10.7,1), TRUNC(10.7,-1) from dual;

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **TRUNC(10.2)** | **TRUNC(10.7)** | **TRUNC(10.2,1)** | **TRUNC(10.2,-1)** | **TRUNC(10.7,1)** | **TRUNC(10.7,-1)** |
| 10 | 10 | 10.2 | 10 | 10.7 | 10 |

### CEIL():-

Take next value

SELECT CEIL (10.2), CEIL(10.7), CEIL(10.9345)from dual;



Ceil will not accept comma in within the function ex

SELECT CEIL(100.2,1),CEIL(100.7456,3) FROM DUAL ;

### MOD():-

function is used to calculate the **remainder** of the division between two numbers. Specifically, it returns the remainder when one number (the dividend) is divided by another number (the divisor).

Query a list of CITY names from STATION with even ID numbers only. You may print the results in any order, but must exclude duplicates from your answer.

SELECT DISTINCT (CITY) FROM STATION WHERE MOD(STATION.ID,2)=0 ORDER BY CITY;

SELECT MOD(7,2) FROM dual;

MOD(7,2)

----------

1

### SIGN:-

returns the sign of an argument.

SELECT SIGN(-145), SIGN(0), SIGN(145) FROM dual;

SIGN(-145) SIGN(0) SIGN(145)

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

-1 0 1

### ABS:-

function returns the **absolute value** of a number. The absolute value of a number is its non-negative value, regardless of whether it is positive or negative.

### Functionality:

* If the input is positive or zero, ABS() returns the same number.
* If the input is negative, ABS() returns its positive equivalent.

SELECT SALARY , ABS(SALARY) FROM EMPLOYEES;

SELECT SALARY , -ABS(SALARY) FROM EMPLOYEES;

## DATE\_FUNCTION:-

Oracle SQL provides a rich set of **date functions** that allow you to work with date and time values. These functions help with manipulating, formatting, and calculating dates, as well as extracting specific components of date values.

### 1. ****SYSDATE****

Returns the current system date and time (based on the database server's time zone).

SELECT SYSDATE FROM DUAL;

### 2. ****CURRENT\_DATE****

Returns the current date in the session's time zone.

SELECT CURRENT\_DATE FROM dual;

### 3. ****SYSTIMESTAMP****

Returns the current date and time, including fractional seconds and the time zone.

SELECT SYSTIMESTAMP FROM dual;

### 4. ****EXTRACT()****

Extracts specific parts of a date (e.g., year, month, day) from a DATE or TIMESTAMP.

SELECT EXTRACT(YEAR FROM SYSDATE) AS year FROM dual;

SELECT EXTRACT(MONTH FROM SYSDATE) AS month FROM dual;

SELECT EXTRACT(DAY FROM SYSDATE) AS day FROM dual;

### 5. Add\_months():-

SELECT ADD\_MONTHS(SYSDATE,3) FROM DUAL;

SELECT ADD\_MONTHS('01-JAN-2020',3) FROM DUAL;

|  |
| --- |
| **ADD\_MONTHS('01-JAN-2020',3)** |
| 01-APR-20 |

SELECT ADD\_MONTHS('01-JAN-2020',-3) FROM DUAL;

|  |
| --- |
| **ADD\_MONTHS('01-JAN-2020',-3)** |
| 01-OCT-19 |

### 6. MONTHS\_BETWEEN :-

function returns the count of months between the two dates.

SELECT MONTHS\_BETWEEN('06-NOV-18', '06-NOV-06') FROM DUAL;

|  |
| --- |
| **MONTHS\_BETWEEN('06-NOV-18','06-NOV-06')** |
| 144 |

### 7. NEXT\_DAY:-

* Find given parmeter date.

select NEXT\_DAY('01-Aug-03', 'TUESDAY') from dual;

|  |
| --- |
| **NEXT\_DAY('01-AUG-03','TUESDAY')** |
| 05-AUG-03 |

### 8. LAST\_DAY:-

* Find last day of the month.

select last\_DAY(ADD\_MONTHS(SYSDATE,-1)) from dual;

|  |
| --- |
| **LAST\_DAY(ADD\_MONTHS(SYSDATE,-1))** |
| 30-NOV-19 |

### 9. TRUNC() IN DATE:-

* It will print 1st day of the month or year. Ex: 10.2 means it will print 10.

SELECT TRUNC(SYSDATE, 'YY')FROM DUAL;

|  |
| --- |
| **TRUNC(SYSDATE,'YY')** |
| 01-JAN-19 |

SELECT HIRE\_DATE, TRUNC(HIRE\_DATE,'MM') FROM EMPLOYEES;

SELECT HIRE\_DATE, TRUNC(HIRE\_DATE,'YY') FROM EMPLOYEES;

SELECT TRUNC(SYSDATE,'DAY')FROM DUAL;

### 10. ROUND() IN DATE:-

* If the system date after june means it will print next year 1st date.
* If the system date before june it will print current year 1st date.

SELECT round(SYSDATE, 'YY')FROM DUAL;

* IN translate single character/int satisfied it will change.
* But replace all argument should be satisfied.
* If the translate argument is NOT available, it consider as a NULL in display. (see example).

## Conversion Function:-

### To\_char:-

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

|  |
| --- |
| **TO\_CHAR(SYSDATE,'DD-MM-YYYY')** |
| 26-02-2020 |

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

|  |
| --- |
| **TO\_CHAR(SYSDATE,'DD-MON-YYYY')** |
| 26-FEB-2020 |

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

|  |
| --- |
| **TO\_CHAR(SYSDATE,'DD-MONTH-YYYY')** |
| 26-FEBRUARY -2020 |

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

|  |
| --- |
| **TO\_CHAR(SYSDATE,'DAY-MONTH-YYYY')** |
| WEDNESDAY-FEBRUARY -2020 |

### To\_date:-

SELECT TO\_DATE('01012020','DD/MM/YYYY') FROM DUAL;

### 3.To\_number:-

SELECT SALARY, TO\_NUMBER(SALARY) FROM EMPLOYEEES;

### 4. ****NEW\_TIME()****

Converts a date from one time zone to another.

SELECT NEW\_TIME(SYSDATE, 'PST', 'EST') AS est\_time FROM dual; -- Converts from Pacific to Eastern Standard Time

### 5. ****FROM\_TZ()****

Converts a timestamp to a specific time zone.

SELECT FROM\_TZ(TIMESTAMP '2024-10-21 12:00:00', 'America/New\_York') FROM dual;

## NULL functions:-

Usually contains NULL handling functions. The functions under the category are NVL, NVL2, NULLIF, COALESCE, CASE, DECODE.

#### NVL:-

It accepts 2 arguments.

/\* If 1st argument is NULL it will display 2nd argument\*/

/\* if 1st argument is NOTNULL it will display 1st argument\*/

Example:-

SELECT NVL(NULL, 2)FROM DUAL;

|  |
| --- |
| **NVL(NULL,2)** |
| 2 |

SELECT EMPLOYEE\_ID, FIRST\_NAME, NVL(COMMISSION\_PCT,0) FROM EMPLOYEES

SELECT NVL(3,4) FROM DUAL;

|  |
| --- |
| **NVL(2,3)** |
| 3 |

SELECT NVL('A','B') FROM DUAL;

|  |
| --- |
| **NVL('A','B')** |
| A |

SELECT EMPLOYEE\_ID, FIRST\_NAME, SALARY, NVL(COMMISSION\_PCT,0) FROM EMPLOYEES WHERE EMPLOYEE\_ID BETWEEN 140 AND 150;

|  |  |  |  |
| --- | --- | --- | --- |
| **EMPLOYEE\_ID** | **FIRST\_NAME** | **SALARY** | **NVL(COMMISSION\_PCT,0)** |
| 140 | Joshua | 2500 | 0 |
| 141 | Trenna | 3500 | 0 |
| 142 | Curtis | 3100 | 0 |
| 143 | Randall | 2600 | 0 |
| 144 | Peter | 2500 | 0 |
| 145 | John | 14000 | .4 |
| 146 | Karen | 13500 | .3 |
| 147 | Alberto | 12000 | .3 |
| 148 | Gerald | 11000 | .3 |
| 149 | Eleni | 10500 | .2 |
| 150 | Peter | 10000 | .3 |

#### NVL2:-

It accepts 3 arguments.

/\* If 1st argument is NULL it will display 3nd argument\*/

/\* if 1st argument is NOTNULL it will display 2st argument\*/

SELECT NVL2(NULL,'A','B') FROM DUAL;

|  |
| --- |
| **NVL2(NULL,'A','B')** |
| B |

SELECT NVL2(1,3,5) FROM DUAL;

|  |
| --- |
| **NVL2(1,3,5)** |
| 3 |

#### NULLIF:-

It will accept 2 arguments.

/\* If both same it will display NULL\*/

/\* If both not same it will display 1st argument\*/

SELECT NULLIF(2,2)FROM DUAL;

|  |
| --- |
| **NULLIF(2,2)** |
| - |

SELECT NULLIF(5,2)FROM DUAL;

|  |
| --- |
| **NULLIF(5,2)** |
| 5 |

#### COALESCE:-

Different between nvl & coalesce is nvl accepts 2 arguments but coalesce accepts multiple arguments (based on the memory)

SELECT COALESCE(NULL,NULL,2,4, NULL, NULL, 5)FROM DUAL;

|  |
| --- |
| **COALESCE(NULL,NULL,2,4,NULL,NULL,5)** |
| 2 |

Real time example find salary & salary + commission in different column from employees table.

# CONTROL STATEMENT:-

#### CASE:-

The CASE function in Oracle SQL is used to implement conditional logic within SQL queries. It allows you to perform **if-then-else** logic directly in SQL, returning different values based on specified conditions. It's similar to the IF or SWITCH statements found in other programming languages.

* When the condition is satisfied then given statement is display.
* You can use RELATIONAL operator in CASE.
* You can use UNLIMITED arguments in CASE.

SELECT

employee\_id, first\_name, SALARY,

(CASE

WHEN SALARY >=20000 THEN 'TOP GRADE'

WHEN SALARY >=15000 AND SALARY <20000 THEN 'MANAGER GRADE'

WHEN SALARY >=10000 AND SALARY <15000 THEN 'TEAM LEADERS'

WHEN SALARY BETWEEN 5000 AND 10000 THEN 'TEAM LEADERS'

ELSE 'CONTRACTERS'

END) AS GRADE

FROM EMPLOYEES;

One more example:-

UPDATE TABLENAME

SET GENDER = (CASE WHEN GENDER='MALE' THEN 'FEMALE'

WHEN GENDER='FEMALE' THEN 'MALE' END)

UPDATE EMPLOYEES

SET GENDER =

(CASE

WHEN 'M' THEN 'F'

WHEN 'F' THEN 'M'

END);

#### DECODE:-

* IT will replace selected code to given values
* You can pass 1 to 255 more arguments in decode.
* you can’t use relational operator in Decode.

SELECT employee\_id, first\_name, SALARY,

DECODE(DEPARTMENT\_ID, 90, 'SUPERBOSS',

60,'BOSS',

100,'MANAGER',

'EMP') AS GRADE

FROM EMPLOYEES;

|  |  |  |  |
| --- | --- | --- | --- |
| **EMPLOYEE\_ID** | **FIRST\_NAME** | **SALARY** | **GRADE** |
| 100 | Steven | 24000 | SUPERBOSS |
| 101 | Neena | 17000 | SUPERBOSS |
| 102 | Lex | 17000 | SUPERBOSS |
| 103 | Alexander | 9000 | BOSS |
| 104 | Bruce | 6000 | BOSS |
| 105 | David | 4800 | BOSS |
| 106 | Valli | 4800 | BOSS |
| 107 | Diana | 4200 | BOSS |
| 108 | Nancy | 12000 | MANAGER |
| 109 | Daniel | 9000 | MANAGER |
| 110 | John | 8200 | MANAGER |
| 111 | Ismael | 7700 | MANAGER |
| 112 | Jose Manuel | 7800 | MANAGER |
| 113 | Luis | 6900 | MANAGER |
| 114 | Den | 11000 | EMP |
| 115 | Alexander | 3100 | EMP |
| 116 | Shelli | 2900 | EMP |
| 117 | Sigal | 2800 | EMP |

SELECT LAST\_NAME, HIRE\_DATE, TO\_CHAR(HIRE\_DATE,'Day'), DECODE(TO\_CHAR(HIRE\_DATE,'D'),1,8, (TO\_CHAR(HIRE\_DATE, 'D'))) DAY FROM EMPLOYEES ORDER BY DAY;

|  |  |  |  |
| --- | --- | --- | --- |
| **LAST\_NAME** | **HIRE\_DATE** | **TO\_CHAR(HIRE\_DATE,'DAY')** | **DAY** |
| Kaufling | 01-MAY-95 | Monday | 2 |
| OConnell | 21-JUN-99 | Monday | 2 |
| Patel | 06-APR-98 | Monday | 2 |
| Errazuriz | 10-MAR-97 | Monday | 2 |
| Bernstein | 24-MAR-97 | Monday | 2 |
| Olsen | 30-MAR-98 | Monday | 2 |
| Sully | 04-MAR-96 | Monday | 2 |
| Smith | 10-MAR-97 | Monday | 2 |

MULTIPLE ROW INTO SINGLE ROW:-

|  |  |
| --- | --- |
| **CASE** | **DECODE** |
| DML operation can perform. (update operation) | Can’t perform. |
| It is a expression. | It is a function. |
| Can perform relation operator like <, <=. | Can’t perform relation operatior. |
| UNLIMITED argument can pass | Can pass 255 auguments. |
| ANSI standard. | ORACLE standard. |

# GROUP FUNCTIONS:-

* To do mathematical functions to operate on sets of rows.
* to give one result per set.

The types of group functions (also called *aggregate functions*) are:

* AVG, that calculates the average of the specified columns in a set of rows,
* COUNT, calculating the number of rows in a set.
* MAX, calculating the maximum,
* MIN, calculating the minimum,
* STDDEV, calculating the standard deviation,
* SUM, calculating the sum,
* VARIANCE, calculating the variance.
* DISTINCT, To Remove duplicate value.
* MEDIAN()
* MODE()
* GROUPING()
* GROUPING\_ID()
* CORR()
* COVAR\_POP()
* COVAR\_SAMP()
* **REGR\_ Functions** (Regression Analysis)

## AVG :-

Calculates the average (mean) value of a numeric column.

SELECT

department\_id,

ROUND(min(SALARY)) AVG\_SAL,

MAX(SALARY) MAX\_SAL, MIN(SALARY) MIN\_SAL

FROM EMPLOYEES

GROUP BY department\_id

ORDER BY department\_id;

DEPARTMENT\_ID AVG\_SAL MAX\_SAL MIN\_SAL

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

10 4400 4400 4400

20 6000 13000 6000

30 2500 11000 2500

40 6500 6500 6500

50 2100 8200 2100

60 4200 9000 4200

70 10000 10000 10000

80 6100 14000 6100

90 17000 24000 17000

100 6900 12008 6900

110 8300 12008 8300

SELECT AVG(employee\_id), AVG(SALARY) FROM EMPLOYEES;

|  |  |
| --- | --- |
| **AVG(EMPLOYEE\_ID)** | **AVG(SALARY)** |
| 153 | 6461.68224299065420560747663551401869159 |

## MAX / MIN :-

SELECT

MAX(employee\_id),

MIN(employee\_id),

MAX(SALARY),

MIN(SALARY)

FROM EMPLOYEES;

|  |  |  |  |
| --- | --- | --- | --- |
| **MAX(EMPLOYEE\_ID)** | **MIN(EMPLOYEE\_ID)** | **MAX(SALARY)** | **MIN(SALARY)** |
| 206 | 100 | 24000 | 2100 |

## Count:-

SELECT \* FROM

(SELECT

department\_id, COUNT(1)

FROM EMPLOYEES

GROUP BY department\_id

ORDER BY department\_id)

WHERE ROWNUM <=10;

DEPARTMENT\_ID COUNT(1)

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

10 1

20 2

30 6

40 1

50 45

60 5

70 1

80 34

90 3

100 6

The below example is different column to row approach:-

SELECT COUNT(employee\_id), SUM(employee\_id), COUNT(SALARY), SUM(SALARY) FROM EMPLOYEES;

SELECT

JOB\_ID,

COUNT(DECODE(DEPARTMENT\_ID,10,DEPARTMENT\_ID)) AS DEPT\_10,

COUNT(DECODE(DEPARTMENT\_ID,20,DEPARTMENT\_ID)) AS DEPT\_20,

COUNT(DECODE(DEPARTMENT\_ID,30,DEPARTMENT\_ID)) AS DEPT\_30,

COUNT(DECODE(DEPARTMENT\_ID,40,DEPARTMENT\_ID)) AS DEPT\_40,

COUNT(DECODE(DEPARTMENT\_ID,50,DEPARTMENT\_ID)) AS DEPT\_50,

COUNT(DECODE(DEPARTMENT\_ID,60,DEPARTMENT\_ID)) AS DEPT\_60,

COUNT(DECODE(DEPARTMENT\_ID,70,DEPARTMENT\_ID)) AS DEPT\_70,

COUNT(DECODE(DEPARTMENT\_ID,80,DEPARTMENT\_ID)) AS DEPT\_80,

COUNT(DECODE(DEPARTMENT\_ID,90,DEPARTMENT\_ID)) AS DEPT\_90,

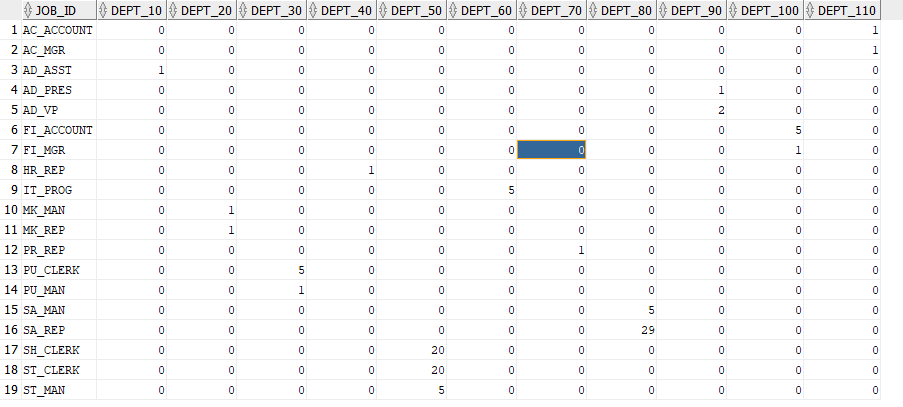
COUNT(DECODE(DEPARTMENT\_ID,100,DEPARTMENT\_ID)) AS DEPT\_100,

COUNT(DECODE(DEPARTMENT\_ID,110,DEPARTMENT\_ID)) AS DEPT\_110

FROM EMPLOYEES

GROUP BY JOB\_ID

ORDER BY JOB\_ID;



## SUM :-

TO calculate sum of group.

SELECT COUNT(employee\_id), SUM(employee\_id), COUNT(SALARY), SUM(SALARY) FROM EMPLOYEES;

|  |  |  |  |
| --- | --- | --- | --- |
| **COUNT(EMPLOYEE\_ID)** | **SUM(EMPLOYEE\_ID)** | **COUNT(SALARY)** | **SUM(SALARY)** |
| 107 | 16371 | 107 | 691400 |

**ROW CONVERTED INTO COLUMN** (OR)

Sum of salary group by job\_id showed by depart\_id

SELECT job\_id,

SUM(DECODE(DEPARTMENT\_ID, 10, SALARY) ) AS DEPT\_10,

SUM(DECODE(DEPARTMENT\_ID, 20, SALARY) ) AS DEPT\_20,

SUM(DECODE(DEPARTMENT\_ID, 30, SALARY) ) AS DEPT\_30,

SUM(DECODE(DEPARTMENT\_ID, 40, SALARY) ) AS DEPT\_40,

SUM(DECODE(DEPARTMENT\_ID, 50, SALARY) ) AS DEPT\_50,

SUM(DECODE(DEPARTMENT\_ID, 60, SALARY) ) AS DEPT\_60,

SUM(DECODE(DEPARTMENT\_ID, 70, SALARY) ) AS DEPT\_70,

SUM(DECODE(DEPARTMENT\_ID, 80, SALARY) ) AS DEPT\_80,

SUM(DECODE(DEPARTMENT\_ID, 90, SALARY) ) AS DEPT\_90,

SUM(DECODE(DEPARTMENT\_ID, 100, SALARY) ) AS DEPT\_100,

SUM(DECODE(DEPARTMENT\_ID, 110, SALARY) ) AS DEPT\_110

FROM EMPLOYEES

GROUP BY JOB\_ID ORDER BY JOB\_ID

Output will be

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **JOB\_ID** | **DEPT\_10** | **DEPT\_20** | **DEPT\_30** | **DEPT\_40** | **DEPT\_50** | **DEPT\_60** | **DEPT\_70** | **DEPT\_80** | **DEPT\_90** | **DEPT\_100** | **DEPT\_110** |
| AC\_ACCOUNT | - | - | - | - | - | - | - | - | - | - | 8300 |
| AC\_MGR | - | - | - | - | - | - | - | - | - | - | 12000 |
| AD\_ASST | 4400 | - | - | - | - | - | - | - | - | - | - |
| AD\_PRES | - | - | - | - | - | - | - | - | 24000 | - | - |
| AD\_VP | - | - | - | - | - | - | - | - | 34000 | - | - |
| FI\_ACCOUNT | - | - | - | - | - | - | - | - | - | 39600 | - |
| FI\_MGR | - | - | - | - | - | - | - | - | - | 12000 | - |
| HR\_REP | - | - | - | 6500 | - | - | - | - | - | - | - |
| IT\_PROG | - | - | - | - | - | 28800 | - | - | - | - | - |
| MK\_MAN | - | 13000 | - | - | - | - | - | - | - | - | - |
| MK\_REP | - | 6000 | - | - | - | - | - | - | - | - | - |
| PR\_REP | - | - | - | - | - | - | 10000 | - | - | - | - |
| PU\_CLERK | - | - | 13900 | - | - | - | - | - | - | - | - |
| PU\_MAN | - | - | 11000 | - | - | - | - | - | - | - | - |
| SA\_MAN | - | - | - | - | - | - | - | 61000 | - | - | - |
| SA\_REP | - | - | - | - | - | - | - | 243500 | - | - | - |
| SH\_CLERK | - | - | - | - | 64300 | - | - | - | - | - | - |
| ST\_CLERK | - | - | - | - | 55700 | - | - | - | - | - | - |
| ST\_MAN | - | - | - | - | 36400 | - | - | - | - | - | - |

## **DISTINCT :-**

* TO REMOVE DUPLICATE VALUES

SELECT DISTINCT(DEPARTMENT\_ID) FROM EMPLOYEES;

* It will print the DEPARTMENT\_ID result with out duplicate

|  |
| --- |
| **DEPARTMENT\_ID** |
| 100 |
| 30 |
| - |
| 90 |
| 20 |
| 70 |
| 110 |
| 50 |
| 80 |
| 40 |
| 60 |
| 10 |

SELECT DEPARTMENT\_ID, MAX(SALARY), MIN(SALARY)

FROM EMPLOYEES

GROUP BY DEPARTMENT\_ID;

|  |  |  |
| --- | --- | --- |
| **DEPARTMENT\_ID** | **MAX(SALARY)** | **MIN(SALARY)** |
| 100 | 12000 | 6900 |
| 30 | 11000 | 2500 |
| - | 7000 | 7000 |
| 90 | 24000 | 17000 |
| 20 | 13000 | 6000 |
| 70 | 10000 | 10000 |
| 110 | 12000 | 8300 |
| 50 | 8200 | 2100 |
| 80 | 14000 | 6100 |
| 40 | 6500 | 6500 |
| 60 | 9000 | 4200 |
| 10 | 4400 | 4400 |

find the highest purchase amount with their ID and order date, for only those customers who have a higher purchase amount in a day is within the list 2000, 3000, 5760 and 6000.

SELECT customer\_id,ord\_date,MAX(purch\_amt)

FROM orders

GROUP BY customer\_id,ord\_date

HAVING MAX(purch\_amt) IN(2000 ,3000,5760, 6000);

## Rollup & cube:-

It will group up and give total values of the result.

select department\_id, sum(salary) from employees group by rollup (department\_id);

|  |  |
| --- | --- |
| **DEPARTMENT\_ID** | **SUM(SALARY)** |
| 10 | 4400 |
| 20 | 19000 |
| 30 | 24900 |
| 40 | 6500 |
| 50 | 156400 |
| 60 | 28800 |
| 70 | 10000 |
| 80 | 304500 |
| 90 | 58000 |
| 100 | 51600 |
| 110 | 20300 |
| - | 7000 |
| - | 691400 |

# Analytic Functions :-

* Oracle analytic functions calculate an aggregate value based on a group of rows and return multiple rows for each group.

## Rank():-

* Calculate the rank of a value in a set of values.
* If the duplicate available The **ranks may not be consecutive** numbers.
* function returns the same rank for the rows with the same values. It adds the number of tied rows to the tied rank to calculate the next rank.

Example:-

SELECT

FIRST\_NAME,

SALARY,

RANK() OVER(ORDER BY SALARY ASC)

FROM EMPLOYEES ORDER BY SALARY ASC;

|  |  |  |
| --- | --- | --- |
| **FIRST\_NAME** | **SALARY** | **RANK()OVER(ORDERBYSALARYASC)** |
| TJ | 2100 | 1 |
| Steven | 2200 | 2 |
| Hazel | 2200 | 2 |
| James | 2400 | 4 |
| Ki | 2400 | 4 |
| Karen | 2500 | 6 |
| James | 2500 | 6 |

SELECT

EMPLOYEE\_ID, DEPARTMENT\_ID,SALARY,

RANK() OVER(PARTITION BY DEPARTMENT\_ID ORDER BY SALARY DESC) AS RK\_1

FROM EMPLOYEES;

EMPLOYEE\_ID DEPARTMENT\_ID SALARY RK\_1

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

200 10 4400 1

201 20 13000 1

202 20 6000 2

114 30 11000 1

115 30 3100 2

116 30 2900 3

117 30 2800 4

118 30 2600 5

119 30 2500 6

203 40 6500 1

121 50 8200 1

## Dense\_Rank() :-

* returns rank values as consecutive integers. It does not skip rank in case of ties. Rows with the same values for the rank criteria will receive the same rank values.

SELECT

FIRST\_NAME,

SALARY,

DENSE\_RANK() OVER(order BY SALARY DESC)

FROM EMPLOYEES;

|  |  |  |
| --- | --- | --- |
| **FIRST\_NAME** | **SALARY** | **DENSE\_RANK()OVER(ORDERBYSALARYASC)** |
| TJ | 2100 | 1 |
| Steven | 2200 | 2 |
| Hazel | 2200 | 2 |
| James | 2400 | 3 |
| Ki | 2400 | 3 |
| Karen | 2500 | 4 |
| James | 2500 | 4 |
| Joshua | 2500 | 4 |

SELECT

EMPLOYEE\_ID, SALARY, DEPARTMENT\_ID,

DENSE\_RANK() OVER(PARTITION BY DEPARTMENT\_ID ORDER BY SALARY DESC) AS RK\_2

FROM EMPLOYEES;

EMPLOYEE\_ID SALARY DEPARTMENT\_ID RK\_2

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

200 4400 10 1

201 13000 20 1

202 6000 20 2

114 11000 30 1

115 3100 30 2

116 2900 30 3

117 2800 30 4

118 2600 30 5

119 2500 30 6

203 6500 40 1

121 8200 50 1

SELECT

E.\*,

DENSE\_RANK() OVER(ORDER BY SALARY DESC)AS RK

FROM EMPLOYEES E;

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **EMPLOYEE\_ID** | **FIRST\_NAME** | **LAST\_NAME** | **EMAIL** | **PHONE\_NUMBER** | **HIRE\_DATE** | **JOB\_ID** | **SALARY** | **COMMISSION\_PCT** | **MANAGER\_ID** | **DEPARTMENT\_ID** | **RK** |
| 201 | Michael | Hartstein | MHARTSTE | 515.123.5555 | 17-FEB-96 | MK\_MAN | 13000 | - | 100 | 20 | 5 |

Questions:-

Retrieve the **top 2 highest-paid employees** in each department.

SELECT \* FROM

(SELECT

EMPLOYEE\_ID, SALARY, DEPARTMENT\_ID,

DENSE\_RANK() OVER(PARTITION BY DEPARTMENT\_ID ORDER BY SALARY DESC) AS RK\_2

FROM EMPLOYEES)

WHERE RK\_2 <=2;

EMPLOYEE\_ID SALARY DEPARTMENT\_ID RK\_2

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

200 4400 10 1

201 13000 20 1

202 6000 20 2

114 11000 30 1

115 3100 30 2

203 6500 40 1

121 8200 50 1

120 8000 50 2

103 9000 60 1

104 6000 60 2

204 10000 70 1

## Row Number() :-

* that assigns a sequential unique integer to each row to which it is applied, either each row in the partition or each row in the result set.

SELECT FIRST\_NAME, SALARY,

ROW\_NUMBER() OVER(order BY SALARY ASC)

FROM EMPLOYEES;

|  |  |  |
| --- | --- | --- |
| **FIRST\_NAME** | **SALARY** | **ROW\_NUMBER()OVER(ORDERBYSALARYASC)** |
| TJ | 2100 | 1 |
| Steven | 2200 | 2 |
| Hazel | 2200 | 3 |
| James | 2400 | 4 |
| Ki | 2400 | 5 |
| Karen | 2500 | 6 |

## Lead():-

* it will print next row in the table (means 2nd coloumn value will come argument column 1st value). Final value filled with passed argument now 0

Example:-

SELECT FIRST\_NAME, SALARY,

LEAD(SALARY,1, 0) OVER(ORDER BY FIRST\_NAME DESC)

FROM EMPLOYEES WHERE ROWNUM <5;

|  |  |  |
| --- | --- | --- |
| **FIRST\_NAME** | **SALARY** | **LEAD(SALARY,1,0)OVER(ORDERBYFIRST\_NAMEDESC)** |
| Steven | 24000 | 17000 |
| Neena | 17000 | 17000 |
| Lex | 17000 | 9000 |
| Alexander | 9000 | 0 |

## Lag():-

* it will print previous ROW IN the table (means 1nd coloumn value will come argument column 2st value).

select first\_name,salary, Lag(salary,1, 0) over(order by first\_name)from employees;

Example:-

select first\_name,salary,

Lag(salary,1, 0) over(order by first\_name)

from employees;

|  |  |  |
| --- | --- | --- |
| **FIRST\_NAME** | **SALARY** | **LAG(SALARY,1,0)OVER(ORDERBYFIRST\_NAME)** |
| Adam | 8200 | 0 |
| Alana | 3100 | 8200 |
| Alberto | 12000 | 3100 |
| Alexander | 9000 | 12000 |
| Alexander | 3100 | 9000 |
| Alexis | 4100 | 3100 |
| Allan | 9000 | 4100 |
| Alyssa | 8800 | 9000 |

## Listagg()

* it will convert a column into single row (will work from 11g onwords).

select listagg(department\_id, '-') within group(order by department\_id) from employees

output:-

10-20-20-30-30-30-30-30-30-40-50-50-50-50-50-50-50-50-50-50-50-50-50-50-50-50-50-50-50-50-50-50-50-50-50-50-50-50-50-50-50-50-50-50-50-50-50-50-50-50-50-50-50-50-50-60-60-60-60-60-70-80-80-80-80-80-80-80-80-80-80-80-80-80-80-80-80-80-80-80-80-80-80-80-80-80-80-80-80-80-80-80-80-80-80-90-90-90-100-100-100-100-100-100-110-110

Listagg working example :-

SELECT

ERROR\_DESC,

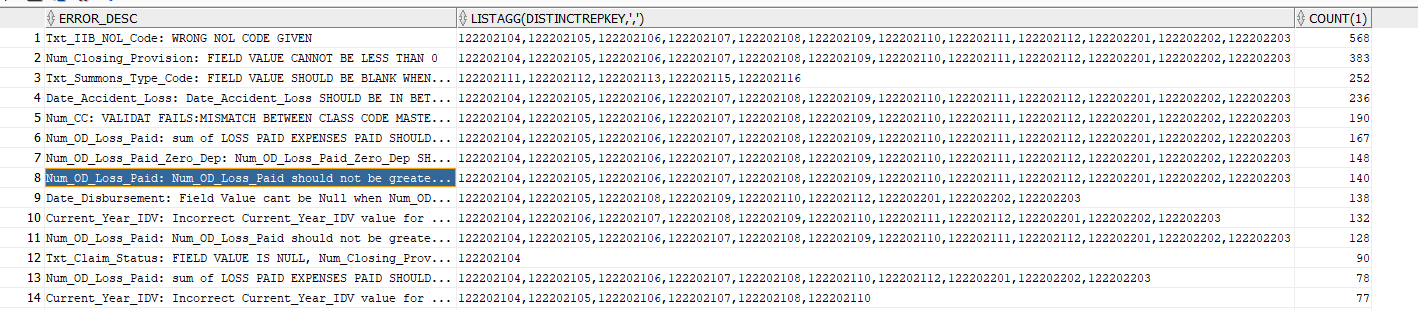
LISTAGG(DISTINCT REPKEY,','),

COUNT(1)

FROM IIB\_REUPLOAD

GROUP BY ERROR\_DESC

ORDER BY COUNT(1) DESC;



## ****NTILE()****

Divides rows into a specified number of buckets (n) and assigns a bucket number to each row.

# PSEUDOCOLUMN:-

* is an Oracle assigned value
* A pseudo-column behaves like a table column but is not actually stored in the table.
* You can select from pseudo-columns, but you cannot insert, update, or delete their values.
* A pseudo-column is also similar to a function without arguments.

1. SYSDATE.
2. TIMESTAMP.
3. ROWNUM.
4. ROWID.
5. NEXTVAL
6. CURRVAL.
7. LEVEL FUNCTION.

## SYSDATE:-

SELECT SYSDATE FROM DUAL;

|  |
| --- |
| **SYSDATE** |
| 22-NOV-19 |

## SYSTIMESTAMP :-

SELECT SYSTIMESTAMP FROM DUAL;

|  |
| --- |
| **SYSTIMESTAMP** |
| 22-NOV-19 02.51.02.964000 PM +05:30 |
|  |

select SYSTIMESTAMP, TO\_CHAR(SYSTIMESTAMP,'HH-MI') FROM DUAL;

## USER :-

SELECT USER FROM DUAL;

|  |
| --- |
| **USER** |
| ANONYMOUS |

## UID :-

SELECT UID FROM DUAL;

|  |
| --- |
| **UID** |
| 28 |

## ROWNUM:-

SELECT ROWID, EMPLOYEE\_ID, ROWNUM FROM EMPLOYEES;

## ROWID:-

* ROWID is the physical location of a row.
* it is the fastest way of locating a row, faster even than a primary key lookup.

SELECT ROWID, EMPLOYEE\_ID, ROWNUM FROM EMPLOYEES;

|  |  |  |
| --- | --- | --- |
| **ROWID** | **EMPLOYEE\_ID** | **ROWNUM** |
| AAAC9EAAEAAAABXAAA | 100 | 1 |
| AAAC9EAAEAAAABXAAB | 101 | 2 |
| AAAC9EAAEAAAABXAAC | 102 | 3 |

**Most efficient method to deleting duplicate:-**

SELECT \* FROM EMPLOYEES E1

WHERE ROWID <

(SELECT MAX(E2.ROWID) FROM EMPLOYEES E2

WHERE E2.EMPLOYEE\_ID = E1.EMPLOYEE\_ID);

## LEVEL FUNCTION:-

SELECT LEVEL FROM DUAL CONNECT BY LEVEL<=5;

|  |
| --- |
| **LEVEL** |
| 1 |
| 2 |
| 3 |
| 4 |
| 5 |

Working Example:-

SELECT

LEVEL AS EMP\_LEVEL,

EMPLOYEE\_ID,

FIRST\_NAME||' ' ||LAST\_NAME,

MANAGER\_ID

FROM EMPLOYEES

START WITH MANAGER\_ID IS NULL

CONNECT BY PRIOR EMPLOYEE\_ID = MANAGER\_ID

ORDER BY EMP\_LEVEL, EMPLOYEE\_ID;

EMP\_LEVEL EMPLOYEE\_ID FIRST\_NAME||''||LAST\_NAME MANAGER\_ID

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

1 100 Steven King

2 101 Neena Kochhar 100

2 102 Lex De Haan 100

2 114 Den Raphaely 100

2 120 Matthew Weiss 100

2 121 Adam Fripp 100

2 122 Payam Kaufling 100

2 123 Shanta Vollman 100

2 124 Kevin Mourgos 100

2 145 John Russell 100

2 146 Karen Partners

Adding Indentation for Visual Representation:-

SELECT

LPAD(' ',3 \* (LEVEL -1 ) ) || FIRST\_NAME AS NAME,

EMPLOYEE\_ID, MANAGER\_ID,LEVEL AS EMP\_LEVEL

FROM EMPLOYEES

START WITH MANAGER\_ID IS NULL

CONNECT BY PRIOR EMPLOYEE\_ID=MANAGER\_ID;

|  |  |  |  |
| --- | --- | --- | --- |
| **NAME** | **EMPLOYEE\_ID** | **MANAGER\_ID** | **EMP\_LEVEL** |
| Steven | 100 |  | 1 |
| Neena | 101 | 100 | 2 |
| Nancy | 108 | 101 | 3 |
| Daniel | 109 | 108 | 4 |
| John | 110 | 108 | 4 |
| Ismael | 111 | 108 | 4 |
| Jose Man | 112 | 108 | 4 |
| Luis | 113 | 108 | 4 |
| Jennifer | 200 | 101 | 3 |
| Susan | 203 | 101 | 3 |
| Hermann | 204 | 101 | 3 |
| Shelley | 205 | 101 | 3 |
| William | 206 | 205 | 4 |
| Lex | 102 | 100 | 2 |
| Alexander | 103 | 102 | 3 |
| Bruce | 104 | 103 | 4 |
| David | 105 | 103 | 4 |
| Valli | 106 | 103 | 4 |

# SEQUENCES

AUTO INCREMENT Field Sequence is a set of integers 1, 2, 3, … that are generated and supported by some database systems to produce unique values on demand.

## CREATE SEQUENCE :-

To increment the column value by default script:

* If don’t mention maxvalue means it is unlimited.
* After created sequence currval not work. After nextval only currval work.
* We can’t alter start value after sequence created.
* NOCYCLE means if reaching max value will throw the error, DEFAULT value is **NOCYCLE.**
* We have to use NOCA

CREATE SEQUENCE IDCREATE

START WITH 1002

INCREMENT BY 1

MINVALUE 1002

MAXVALUE 2000

--cycle nocache; ----it repert the values from start values.

NOCYCLE;

INSERT INTO TEST VALUES(**IDCREATE.NEXTVAL,** 'PAPAA');

* WHEN WE ALTER THE SEQUENCE ONLY MAX VALUE, & INCREMENTED BY

## ALTER SEQUENCE :-

ALTER SEQUENCE IDCREATE

INCREMENT BY 1

MINVALUE 1000

MAXVALUE 1015;

## CURRENT & NEXT SEQUENCE VALUES:-

## CURRVAL:-

SELECT empseq.CURRVAL

FROM DUAL;

## NEXTVAL:-

SELECT empseq.NEXTVAL

FROM DUAL;

SELECT IDCREATE.CURRVAL FROM DUAL;

SELECT IDCREATE.NEXTVAL FROM DUAL;

select TO\_CHAR(SYSDATE,'MON') || '0000' ||SQ1.nextval

from dual;

## DROP SEQUENCE :-

DROP SEQUENCE IDCREATE;

# Constraints:-

* Constraints are the rules enforced on the data columns of a table
* SQL constraints are used to specify rules for the data in a table.

Constraints are used to limit the type of data that can go into a table. This ensures the accuracy and reliability of the data in the table. If there is any violation between the constraint and the data action, the action is aborted.

Constraints can be column level or table level. Column level constraints apply to a column, and table level constraints apply to the whole table.

* [**PRIMARY KEY**](https://www.w3schools.com/sql/sql_primarykey.asp) - A combination of a NOT NULL and UNIQUE. Uniquely identifies each row in a table.
* [**FOREIGN KEY**](https://www.w3schools.com/sql/sql_foreignkey.asp) - Uniquely identifies a row/record in another table.
* The **foreign key** establishes a link to the primary key

A table can have **multiple foreign keys**.

* [**NOT NULL**](https://www.w3schools.com/sql/sql_notnull.asp) - Ensures that a column cannot have a NULL value.
* [**UNIQUE**](https://www.w3schools.com/sql/sql_unique.asp) - Ensures that all values in a column are different, it accepts NULL VALUE
* [**CHECK**](https://www.w3schools.com/sql/sql_check.asp) - Ensures that values in a column satisfy a specific condition.

Example : CONSTRAINT C6 CHECK(GENDER IN('M','F'))

## **Create constraints:-**

Create table cource

(

cid number,

cname varchar(30) NOT NULL,

CONSTRAINT C1 PRIMARY KEY(CID)

);

CREATE TABLE STUDENTS

(

SID NUMBER,

CID NUMBER(30),

SNAME VARCHAR(30) NOT NULL,

PHONE NUMBER,

DOB DATE NOT NULL,

EMAIL VARCHAR2(30),

DOJ TIMESTAMP,

GENDER CHAR(1),

ADDRESS VARCHAR(200),

RESUME LONG,

CONSTRAINT C2 PRIMARY KEY(SID),

CONSTRAINT C3 UNIQUE(PHONE),

CONSTRAINT C4 UNIQUE(EMAIL),

CONSTRAINT C5 CHECK(LENGTH(PHONE)=10),

CONSTRAINT C6 CHECK(GENDER IN('M','F')),

CONSTRAINT C7 FOREIGN KEY(CID) REFERENCES COURCE(CID)

);

## Drop a constraint:-

ALTER TABLE STUDENDS DROP CONSTRAINT C2;

## Add a Constraint into existing table:-

ALTER TABLE STUDENTS ADD CONSTRAINT C2 PRIMARY KEY(CID);

SELECT r.table\_name

FROM user\_constraints t

JOIN user\_constraints r ON t.r\_constraint\_name = r.constraint\_name

WHERE t.constraint\_type = 'R' AND t.table\_name = 'STUDENTS';

SELECT \* FROM USER\_TABLES

SELECT \* FROM USER\_TAB\_COLUMNS

SELECT \* FROM USER\_CONSTRAINTS

# SET OPERATORS:-

* The set operators are availed to combine information of similar type from one or more than one table.

Types of operators:-

* **Union**
* **Union all**
* **Intersect**
* **Minus**

## Union:-

* It will print common value & unique values. Or (Select all distinct rows by either query.)

|  |  |  |
| --- | --- | --- |
| **T1** |  | **T2** |
| 1 |  | 6 |
| 2 |  | 1 |
| 3 |  | 2 |
| 4 |  | 3 |
| 5 |  | 7 |
|  |  | 8 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  | **NO** |  |
|  |  |  |  | 1 |  |
| SELECT \* FROM T1 | | |  | 2 |  |
| UNION |  |  | OUT PUT | 3 |  |
| SELECT \* FROM T2; | | |  | 4 |  |
|  |  |  |  | 5 |  |
|  |  |  |  | 6 |  |
|  |  |  |  | 7 |  |
|  |  |  |  | 8 |  |

## UNION ALL:-

* Print all values in both tables column, include duplicate.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  | **NO** |
| SELECT \* FROM T1 | |  |  |  | 1 |
| UNION all | |  |  |  | 2 |
| SELECT \* FROM T2 | |  |  |  | 3 |
|  |  |  |  |  | 4 |
|  |  |  |  |  | 5 |
|  |  |  |  |  | 6 |
|  |  |  |  |  | 1 |
|  |  |  |  |  | 2 |
|  |  |  |  |  | 3 |
|  |  |  |  |  | 7 |
|  |  |  |  |  | 8 |

## Intersect:-

* Print only common values in both table columns.

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | **NO** |
| SELECT \* FROM T1 | |  | 1 |
| INTERSECT | |  | 2 |
| SELECT \* FROM T2 | |  | 3 |
|  |  |  |  |

## MINUS:-

* It prints REMAIN T1 ONLY IT WILL MINUS T1- T2 TABLES.

T1 T2

1 1

2 2

3 4

5

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| SELECT \* FROM T2 | | |  | **NO** |
| MINUS |  |  |  | 4 |
| SELECT \* FROM T1 | | |  | 5 |

## COMBINATION OF UNION:-

SELECT \* FROM T1

UNION ALL

SELECT \* FROM T2

MINUS

SELECT \* FROM T2

**Summary of Set Operators:-**

|  |  |
| --- | --- |
| **Operator** | **Description** |
| UNION | Combines results, removing duplicates |
| UNION ALL | Combines results, keeping all duplicates |
| INTERSECT | Returns only rows present in both queries |
| MINUS | Returns rows from the first query that are not in the second |

# SQL LODER:-

* It is a bulk loader utility used for moving data from external files into the oracle database.

**TYPES OF LODER:-**

1. Conventional path loader:
2. Direct path loader:

## Conventional path loader:-

* first it will constrains in the table.
* Next it will check empty blocks if available it will fill empty block first.
* It will save log file, then it will save the data file.
* Direct = false

## Direct path loader:-

* It is fast loader.
* Not check any condition it will directly save to the data file.

STEP 1: FIND SOURCE FILE (MODIFY BY CSV FILE)

STEP 2: create control file as below

LOAD DATA INFILE 'D:\sql\marks.csv' /\*Data File Path\*/

Append

INTO TABLE marks

fields TERMINATED BY ','

(ID,TAMIL,ENGLISH,MATHS,SCIENCE,SOCIAL) /\*Column Names\*/

Step 3: Run SQL LODER QUERY in CMD PROMPT(not in sqlplus)

SQLLDR HR/ADMIN CONTROL=D:\NEW\1.CTL

SKIP=1

BAD= D:\NEW\BADFILE.BAD

LOG= D:\NEW\LOGFILE.LOG

ERROR=5

DIRECT = FALSE

# SUBQUERY:-

* Generally query within a query call sub query,
* It may occur SELECT, FROM, WHERE clause.
* Sub query can be nested inside SELECT, INSERT, UPDATE, DELETE inside another subquery.
* You can use comparison operator such as > < or =

TYPES OF SUBQUERYS:

1. Single Row Sub Query.
2. Multiple row sub query.
3. Nested row sub query.
4. Inline view sub query.
5. Scalar row sub query.
6. Correlated Sub Query.

## Single Row Sub Query:-

* It will display a single row as a output then it is single row subquery.
* Can use ONLY THIS EXPRESSIONS (=, <, >, <=, >=, )

Example:-

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

FROM EMPLOYEES

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

|  |  |  |  |
| --- | --- | --- | --- |
| **EMPLOYEE\_ID** | **FIRST\_NAME** | **SALARY** | **DEPARTMENT\_ID** |
| 132 | TJ | 2100 | 50 |

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

FROM EMPLOYEES E JOIN DEPARTMENTS D

ON E.DEPARTMENT\_ID = D.DEPARTMENT\_ID

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

|  |  |  |  |
| --- | --- | --- | --- |
| **FIRST\_NAME** | **SALARY** | **DEPARTMENT\_ID** | **DEPARTMENT\_NAME** |
| Steven | 24000 | 90 | Executive |

## Multiple Row Sub Query:-

* Nested queries that can return more than one row of results to the parent query.
* Multiple-row subqueries are used most commonly in WHERE and HAVING clauses.
* Since it returns multiple rows,it must be handled by set comparison operators (IN, ALL, ANY).
* [> ALL] More than the highest value
* [< ALL] Less than the lowest value
* [> ANY] More than the lowest value
* [< ANY] Less than the highest value
* [= ANY] Equal to any value

Example:-

SELECT \* FROM EMPLOYEES

WHERE SALARY >ANY(SELECT SALARY FROM EMPLOYEES WHERE SALARY IN(17000,9000));

## NESTED SUBQUERY:-

Subquery inside a subquery is called nested subquery.

Example:- (find out 3rd maximum salary in employees table)

Select max(salary)

from employees where salary <(Select max(salary)

from employees where salary < (select max(salary)from employees));

## INLINE VIEW SUBQUERY :-

Subquery written on the from clause is called inline view subquery.

Example1:-

SELECT \* FROM

(SELECT FIRST\_NAME, LAST\_NAME, SALARY\*12 AS ANNSAL FROM EMPLOYEES);

Example2:-(using ROWNUM in where clause)

SELECT \* FROM (SELECT E.\*, ROWNUM AS RN FROM EMPLOYEES E) WHERE RN=3;

SELECT MIN(SALARY) FROM

(SELECT UNIQUE SALARY FROM EMPLOYEES ORDER BY SALARY DESC)

WHERE ROWNUM <=5;

## Multiple Column Sub Query:-

* more than one column to the outer query and can be listed in the outer query's FROM, WHERE, or HAVING clause.

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

WHERE (DEPARTMENT\_ID, SALARY)

IN (SELECT DEPARTMENT\_ID, MAX(SALARY)FROM EMPLOYEES

GROUP BY DEPARTMENT\_ID);

## SCALAR SUBQUERY:-

It is used on select clause is called scalar subquery.

SELECT 2+(SELECT 5+6 FROM DUAL)FROM DUAL;

select

(SELECT COUNT(DEPARTMENT\_ID) FROM EMPLOYEES WHERE DEPARTMENT\_ID = 90) AS D90,

(SELECT COUNT(DEPARTMENT\_ID) FROM EMPLOYEES WHERE DEPARTMENT\_ID = 60) AS D60,

(SELECT COUNT(\*) FROM EMPLOYEES) AS TOTAL

.

FROM DUAL;

|  |  |  |
| --- | --- | --- |
| **D90** | **D60** | **TOTAL** |
| 3 | 5 | 107 |

SELECT name, salary, (SELECT AVG(salary) FROM employees) as avg\_salary

FROM employees

WHERE department = 'Sales';

## CORELATED SUBQUERY:-

First outer query has to be executed & generates some out put values.

Second based on this output values, subquery has to be executed.

Third subquery we have some output and it is passed to outer query again.

Forth outer query has to be executed.

display the employee\_id, manager\_id, first\_name and last\_name of those employees who manage other employees:-

SELECT employee\_id, manager\_id, first\_name, last\_name

FROM employees a

WHERE EXISTS (SELECT employee\_id FROM employees b WHERE b.manager\_id = a.employee\_id);

SELECT department\_id, department\_name

FROM departments d

WHERE NOT EXISTS (SELECT ’X’

FROM employees

WHERE department\_id= d.department\_id);

SELECT

E.\*

FROM EMPLOYEES E

WHERE E.DEPARTMENT\_ID=(SELECT D.DEPARTMENT\_ID

FROM DEPARTMENT D

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

Example:- update the column values by matching cid.

UPDATE STUDENTS SET CNAME=COURSE.CNAME WHERE CID IN (SELECT CID FROM COURSE WHERE STUDENTS.CID=COURSE.CID);

Students course

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **SID** | **SNAME** | **CID** | **CNAME** |  | **CID** | **CNAME** |  |
| 1 | stevel | 20 | - |  | 10 | sql |  |
| 3 | lex | 20 | - |  | 20 | unix |  |
| 2 | neena | 10 | - |  | 30 | WINDOWS |  |

After execution Students table CNAME will updated the values.

# JOINS :

* A JOIN clause is used to combine rows from two or more tables, based on a related column between them.
* JOINS are used to retrieve data from multiple tables based on a relationship between specified columns.
* An Oracle JOIN is performed whenever two or more tables are joined in a SQL statement.

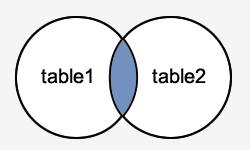
**Types of joins:-**

* INNER JOIN (or sometimes called simple join). Return matching Records in both tables.
* LEFT OUTER JOIN (or sometimes called LEFT JOIN)Returns all records from the left table, and the matched records from the right table.
* RIGHT OUTER JOIN (or sometimes called RIGHT JOIN)Returns all records from the right table, and the matched records from the left table
* FULL OUTER JOIN (or sometimes called FULL JOIN) Returns all records when there is a match in either left or right table
* CROSS JOIN returns all combinations of rows from each table
* SELF JOIN. Retruns the matching records from the same table.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| SUPPLIER | |  | ORDERS | | |
| **SUPPLIER\_ID** | **SUPPLIER\_NAME** |  | **ORDER\_ID** | **ORDER\_DATE** | **SUPPLIER\_ID** |
| 10001 | HP |  | 500125 | 12-May-03 | 10000 |
| 10002 | MICROSOFT |  | 500126 | 13-May-03 | 10001 |
| 10003 | NIVEDA |  | 500127 | 14-May-03 | 10004 |
| 10000 | IBM |  | 500129 | 16-May-03 | 10006 |

## INNER / NATURAL/ SIMPLE JOIN : -

INNER JOINS return all rows from multiple tables where the join condition is met.



Example 1:-

SELECT \* FROM

EMPLOYEES E inner join DEPARTMENTS D

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

EXAMPLE 2: -

SELECT S.SUPPLIER\_ID, O.ORDER\_ID, S.SUPPLIER\_NAME, O.ORDER\_DATE FROM

ORDERS O, SUPPLIER S

WHERE O.SUPPLIER\_ID = S.SUPPLIER\_ID;

EXAMPLE 3:-

SELECT \*FROM

SUPPLIER S, ORDERS O

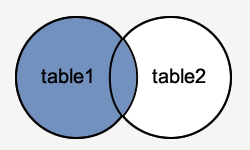
WHERE S.SUPPLIER\_ID = O.SUPPLIER\_ID;

SAME OUTUPT FOR BOTH:-

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **SUPPLIER\_ID** | **SUPPLIER\_NAME** | **ORDER\_ID** | **ORDER\_DATE** | **SUPPLIER\_ID** |
| 10000 | IBM | 500125 | 12-MAY-03 | 10000 |
| 10001 | HP | 500126 | 13-MAY-03 | 10001 |

## LEFT OUTER JOIN : -

A LEFT JOIN returns all rows from the left (first) table and the matching rows from the right (second) table. If no match is found, NULL values are returned for columns from the right table.



EXAMPLE 1: -

SELECT \*

FROM SUPPLIER S, ORDERS O

WHERE S.SUPPLIER\_ID = O.SUPPLIER\_ID (+);

Example 2:-

SELECT

S.SUPPLIER\_ID, S.SUPPLIER\_NAME, O.ORDER\_ID, O.ORDER\_DATE

FROM SUPPLIER S LEFT OUTER JOIN ORDERS O

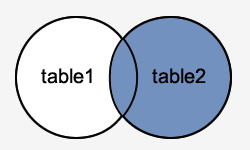
ON S.SUPPLIER\_ID = O.SUPPLIER\_ID;

SAME OUTUPT FOR BOTH:-

|  |  |  |  |
| --- | --- | --- | --- |
| **SUPPLIER\_ID** | **SUPPLIER\_NAME** | **ORDER\_ID** | **ORDER\_DATE** |
| 10000 | IBM | 500125 | 12-MAY-03 |
| 10001 | HP | 500126 | 13-MAY-03 |
| 10002 | MICROSOFT | - | - |
| 10003 | NIVEDA | - | - |

## RIGHT OUTER JOIN : -

A RIGHT JOIN is similar to a left join but includes all rows from the right (second) table and matching rows from the left (first) table. If no match is found, NULL values are returned for columns from the left table.



EXAMPLE 1:-

SELECT S.SUPPLIER\_ID, S.SUPPLIER\_NAME, O.ORDER\_ID, O.ORDER\_DATE

FROM SUPPLIER S, ORDERS O

WHERE S.SUPPLIER\_ID(+) = O.SUPPLIER\_ID;

EXAMPLE 2:-

SELECT \*

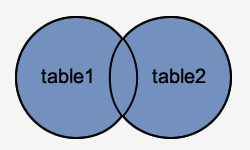
FROM SUPPLIER S RIGHT OUTER JOIN ORDERS O

ON S.SUPPLIER\_ID = O.SUPPLIER\_ID;

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **SUPPLIER\_ID** | **SUPPLIER\_NAME** | **ORDER\_ID** | **ORDER\_DATE** | **SUPPLIER\_ID** |
| 10001 | HP | 500126 | 13-MAY-03 | 10001 |
| 10000 | IBM | 500125 | 12-MAY-03 | 10000 |
| - | - | 500127 | 14-MAY-03 | 10004 |
| - | - | 500129 | 16-MAY-03 | 10006 |

## FULL OUTER JOIN : -

A FULL JOIN returns all rows when there is a match in one of the tables. If there’s no match, NULL values are included from the non-matching side.



EXAMPLE:-

SELECT \*FROM

SUPPLIER S FULL OUTER JOIN ORDERS O

ON S.SUPPLIER\_ID = O.SUPPLIER\_ID;

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **SUPPLIER\_ID** | **SUPPLIER\_NAME** | **ORDER\_ID** | **ORDER\_DATE** | **SUPPLIER\_ID** |
| 10000 | IBM | 500125 | 12-MAY-03 | 10000 |
| 10001 | HP | 500126 | 13-MAY-03 | 10001 |
| 10002 | MICROSOFT | - | - | - |
| 10003 | NIVEDA | - | - | - |
| - | - | 500127 | 14-MAY-03 | 10004 |
| - | - | 500129 | 16-MAY-03 | 10006 |

## CROSS JOIN :-

The CROSS JOIN is used to generate a paired combination of each row of the first table with each row of the second table.

Cross joins are used to return every combination of rows from two tables,



SELECT \*

FROM SUPPLIER S CROSS JOIN ORDERS O;

OUTUPT:-

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **SUPPLIER\_ID** | **SUPPLIER\_NAME** | **ORDER\_ID** | **ORDER\_DATE** | **SUPPLIER\_ID** |
| 10001 | HP | 500125 | 12-MAY-03 | 10000 |
| 10002 | MICROSOFT | 500125 | 12-MAY-03 | 10000 |
| 10003 | NIVEDA | 500125 | 12-MAY-03 | 10000 |
| 10000 | IBM | 500125 | 12-MAY-03 | 10000 |
| 10001 | HP | 500126 | 13-MAY-03 | 10001 |
| 10002 | MICROSOFT | 500126 | 13-MAY-03 | 10001 |
| 10003 | NIVEDA | 500126 | 13-MAY-03 | 10001 |
| 10000 | IBM | 500126 | 13-MAY-03 | 10001 |
| 10001 | HP | 500127 | 14-MAY-03 | 10004 |
| 10002 | MICROSOFT | 500127 | 14-MAY-03 | 10004 |
| 10003 | NIVEDA | 500127 | 14-MAY-03 | 10004 |
| 10000 | IBM | 500127 | 14-MAY-03 | 10004 |
| 10001 | HP | 500129 | 16-MAY-03 | 10006 |
| 10002 | MICROSOFT | 500129 | 16-MAY-03 | 10006 |
| 10003 | NIVEDA | 500129 | 16-MAY-03 | 10006 |
| 10000 | IBM | 500129 | 16-MAY-03 | 10006 |

## SELF JOIN :-

A self join is joined with itself, especially when the table has a FOREIGN KEY which references its own PRIMARY KEY. To join a table itself means that each row of the table is combined with itself and with every other row of the table.

Example :-

SELECT E1.EMPLOYEE\_ID, E1.FIRST\_NAME, E1.SALARY, E2.FIRST\_NAME AS MANAGER\_NAME

FROM EMPLOYEES E1 INNER JOIN EMPLOYEES E2

ON E2.EMPLOYEE\_ID = E1.MANAGER\_ID;

## Un matched Records:-

SELECT \*

FROM EMPLOYEES E FULL JOIN DEPARTMENTS D

ON E.DEPARTMENT\_ID = D.DEPARTMENT\_ID

WHERE E.DEPARTMENT\_ID IS NULL OR D.DEPARTMENT\_ID IS NULL;

## MULTI JOINS :-

* Multi joins will joins the 3 or more tables

SELECT \* FROM

EMPLOYEES E FULL JOIN DEPARTMENTS D

ON E.DEPARTMENT\_ID = D.DEPARTMENT\_ID

JOIN LOCATIONS L

ON D.LOCATION\_ID = L.LOCATION\_ID;

SELECT \* FROM

EMPLOYEES E JOIN DEPARTMENTS D

ON E.DEPARTMENT\_ID = D.DEPARTMENT\_ID

JOIN LOCATIONS L

ON D.LOCATION\_ID = L.LOCATION\_ID

WHERE SALARY >15000;

# SYNONYM:-

* It is used to create Alice name of an table name.
* by using synonym we can perform DML operation.
* Synonyms make the referencing of objects easier.
* we can create a multiple synonym for a table.
* It will be stored parentally in data base,

SELECT \* FROM all\_synonyms;

There are 2 basic categories of synonyms :  
1. Public synonyms  
2. Private synonyms

## Public synonyms Example:-

CREATE PUBLIC SYNONYM EMP FOR EMPLOYEES;

## Private synonyms EXAMPLE:-

CREATE SYNONYM EMP FOR EMPLOYEES;

INSERT USING SYNONYM:

INSERT INTO EMP VALUES(102, ‘RAK’, 85);

DROP PUBLIC SYNONYM EMP;

DROP SYNONYM EMP;

# VIEWS :-

* Views are virtual table with display the subject of one (or) more tables.
* It does not physically exist.
* It is stored in Oracle data dictionary and do not store any data.
* It can be executed when called.

1. It stores only the query.
2. It is not stored in database.
3. Mostly use for security (if have salary column we can hide and create the views).
4. Mostly use for simplicity complex query can combain and create view.
5. Views are mainly created for security purpose in order to restrict user access to specific columns.

TO SEE already created VIEWS IN DB

SELECT \* FROM user\_VIEWS

## Simple view :-

1. It is created on single base table.
2. You can do DML operation in simple table.
3. When a simple view contains GROUP BY, GROUP FUNCTION, DISTINCT CLASE, SET OPERATION, JOINS, ROWNUM, Then we cannot perform DML operation on view.
4. If you want use GROUP FUNCTION then you must use alias names for them.

Example:-

CREATE VIEW V1 AS SELECT \* FROM EMP;

## Complex view:-

1. It is created on multiple base table.
2. We can use insert values in complex view by using TRIGGERS.

Example:-

CREATE OR REPLACE VIEW MGR\_VIEW AS

SELECT E1.EMPLOYEE\_ID, E1.FIRST\_NAME, E1.SALARY, E2.FIRST\_NAME AS MANAGER\_NAME, E1.MANAGER\_ID

FROM EMPLOYEES E1 INNER JOIN EMPLOYEES E2

ON E2.EMPLOYEE\_ID = E1.MANAGER\_ID;

## With read only :-

User can only read the data cannt do insert OR update.

Example:-

CREATE VIEW V1 AS

SELECT

FIRST\_NAME

FROM EMP

WITH READ ONLY;

## With check option :-

row insertion is allowed based on some condition. It is used for restricted DML operation.

Example:-

CREATE VIEW V1

AS

SELECT EMP\_ID, FIRST\_NAME

FROM EMPLOYEES

WHERE BETWEEN(100 TO 110) WITH CHECK OPTION;

## Force View:-

If you want create views With out a base table than use force view.

Example:-

CREATE OR REPLACE FORCE VIEW V1

AS

SELECT \* FROM WELCOME;

After we can create table with views name it will be automatically connected to view.

Create table welcome(id number, name varchar);

To drop views :-

DROP VIEW V1;

# MATERIALIZED VIEW:-

* It is a database object that, It will store the query O/P as a snapshot.
* It is having separate memory location.
* While create MV should have a **primary key**.
* In replication environments, the materialized views commonly created are primary key, rowid, and subquery materialized views.

To see already created MATERIALIZED VIEW MV

SELECT \* FROM USER\_MVIEWS;

Example:- (create MV without refresh)

CREATE TABLE TEST6(ID NUMBER(5), NAME VARCHAR2(10));

ALTER TABLE TEST6 ADD PRIMARY KEY(ID);

INSERT INTO TEST6 VALUES(1,'SRK');

INSERT INTO TEST6 VALUES(2,'RAJ');

COMMIT;

CREATE MATERIALIZED VIEW MV1

AS

SELECT \* FROM TEST6;

Insert some values in base table,

* If you do any dml operation in base table it will not automatically update in MATERIALIZED VIEW.
* For that we have to do some refresh command in SQL.

Manual refresh:- (By default it is complete refresh)

BEGIN

DBMS\_MVIEW.REFRESH('MV1');

END;

Or

EXEC DBMS\_MVIEW.REFRESH(‘MV1’);

There is two type of Refresh:

* Complete Refresh.
* Fast Refresh.
* Force Refresh.

## COMPLETE REFRESH:-

* By default materialized view are complete refresh.
* The rowid of MV has frequently changed when we do DML operation on base table.

Example:-

CREATE MATERIALIZED VIEW MV1 REFRESH COMPLETE

AS

SELECT \* FROM TEST6;

BEGIN

DBMS\_MVIEW.REFRESH('MV1');

END;

BEGIN

DBMS\_MVIEW.REFRESH('MV1','F');

END;

## Fast refresh:-

* Before creating fast refresh we have to create a log file for a table.
* While creating fast refresh materialized view primary key column should be there in MV.
* It is faster then complete refresh because this rowid will not change in fast refresh.
* It is also called **(incremental) refresh**

Example :-

CREATE MATERIALIZED VIEW LOG ON TEST6;

CREATE MATERIALIZED VIEW MV1 REFRESH FAST

AS

SELECT \* FROM TEST6;

On commit refresh :- (auto commit when commit)

When you commit it will refresh

CREATE MATERIALIZED VIEW MV1 REFRESH FAST ON COMMIT

AS

SELECT \* FROM TEST6;

## FORCE REFRESH :-

* When we do force refresh will check first Log file it is available it will perform fast refresh if log file not available it will perform complete refresh.

CREATE MATERIALIZED VIEW MV1 REFRESH FORCE

AS

SELECT \* FROM TEST6;

### Primary Key Materialized Views

The following statement creates the primary-key materialized view on the table emp located on a remote database.

SQL> CREATE MATERIALIZED VIEW mv\_emp\_pk  
 REFRESH FAST START WITH SYSDATE   
 NEXT SYSDATE + 1/48  
 WITH PRIMARY KEY   
 AS SELECT \* FROM emp@remote\_db;  
  
Materialized view created.

Note: When you create a materialized view using the FAST option you will need to create a view log on the master tables(s) as shown below:

SQL> CREATE MATERIALIZED VIEW LOG ON emp;  
Materialized view log created.

### Rowid Materialized Views

The following statement creates the rowid materialized view on table emp located on a remote database:

SQL> CREATE MATERIALIZED VIEW mv\_emp\_rowid   
 REFRESH WITH ROWID   
 AS SELECT \* FROM emp@remote\_db;   
  
Materialized view log created.

### Subquery Materialized Views

The following statement creates a subquery materialized view based on the emp and dept tables located on the remote database:

SQL> CREATE MATERIALIZED VIEW mv\_empdept  
AS SELECT \* FROM emp@remote\_db e  
WHERE EXISTS  
 (SELECT \* FROM dept@remote\_db d  
 WHERE e.dept\_no = d.dept\_no)

### Timing the refresh

The START WITH clause tells the database when to perform the first replication from the master table to the local base table. It should evaluate to a future point in time. The NEXT clause specifies the interval between refreshes

SQL> CREATE MATERIALIZED VIEW mv\_emp\_pk

REFRESH FAST

START WITH SYSDATE

NEXT SYSDATE + 1

WITH PRIMARY KEY

AS SELECT \* FROM emp@remote\_db;

CREATE MATERIALIZED VIEW MV1

REFRESH FORCE ON DEMAND

START WITH TO\_DATE('19-NOV-2021 19:52:00', 'DD-MM-YYYY HH24:MI:SS') NEXT SYSDATE + 1/1152

As select \* from TEST6;

Materialized view created.

In the above example, the first copy of the materialized view is made at SYSDATE and the interval at which the refresh has to be performed is every days.

## Different b/w VIEWS MATERIALIZED VIEW:-

|  |  |
| --- | --- |
| **VIEWS** | **MATERIALIZED VIEW** |
| 1. It stores the query only not output. | * It stores the query output. |
| 1. Does not require primary key. | * Base table should have a primary key. |
| 1. Does not require refresh. | * When we do DML operation we must do refresh for updation in MV table |
| 1. Data’s are not stored in data base because views & base table ROWID are SAME. | * Data stored in data base in different location because MV & base table ROWID are different. |
| 1. Can do DML operation. | * DML operation can’t do. |
| 1. When we drop base table views data are not available. | * When we drop base table the data in MV is available. |

# INDEX:-

* Index is a database object.
* It will rearrange the indexed values by ascending order and stored into the separate table. (it not visible to user).
* It reduce the CPU Utilization when use of INDEX.
* Which is used to retrieve the data very fast from the database.
* It will store the data separately.
* It reduces number of comparisons to make to fetch the required data.
* When we create primary key constraint or unique constraint in a column, automatically a b-tree index are created.

IMPORTANCE NOTES:

* It contains 2part which is data part & address(ROWID) part.
* Index will work only on when we use where clause fetch the data, if not use where clause Index will NOT work.
* When ever where class contain NOT equal & NOT NULL then also it will not search for index.
* We can’t create index for views & can create for MV.
* If you create index for firstname & lastname there is no issue in index, problem is how you write a query, if you search by only firstname will use firstname index, if you search by only last name it will use lastname index, if you use both name index will not work.
* Index taken LESS size of APPRX 1/3 OF THE BASE TABLE the original table.

### ****Why Use Indexes?****

Indexes are essential for **query performance** and **data retrieval speed**. Here’s why you might want to create indexes:

* **Fast Data Retrieval**: Indexes allow Oracle to find rows faster, making queries quicker.
* **Reduced Disk I/O**: Oracle can retrieve fewer rows from the disk by narrowing down the search using an index, reducing disk access.
* **Improved Query Performance**: Indexes can significantly improve the performance of SELECT statements with WHERE clauses, JOINs, and aggregate functions.

### ****When to Use Indexes****

Indexes are beneficial in cases like:

* **Frequent Searches**: For columns often used in WHERE clauses, JOIN conditions, or filtering criteria.
* **Large Tables**: Indexes are most effective in large tables, as they reduce the need to scan all rows.
* **High Cardinality Columns**: For example, indexing columns with many unique values, like SSN or email.

Find index in the table:-

SELECT \*

FROM all\_indexes

WHERE table\_name = UPPER('PGIW\_TAC\_HEALTH\_RID');

To see already created index:-

SELECT \* FROM user\_indexes;

SELECT \* FROM ALL\_IND\_COLUMNS WHERE TABLE\_NAME ='EMPLOYEES';

**Types of Index:-**

* B-tree INDEX
* BITMAP INDEX (NOT B-tree Index)
* Simple INDEX
* Composite Index
* Unique Index
* Reverse key Index
* Function based Index.

**Cardinality = number of distinct values / total no of rows.**

**To see index’s in DB:-**

Select \* from user\_indexes

## B-Tree Index:-

* It is created on high cardinality columns.
* When we create primary key By default B-tree index created.
* That is Less number of duplicate columns (ex: employee\_id).
* By Default index is B-Tree Index

Employee id = 107 (distinct) / 107 rows

That is high cardinality column.

## Bit map Index:-

* It is created on low cardinality columns, that is having more duplicates.
* Salary = 58(distinc)/107 rows -> it is low cardinality column.

Example:-

Create bitmap index ind1

On

Employees(salary);

## Simple Index:-

* Index created on single column is called simple index.

Example:-

CREATE INDEX IND1

ON

EMPLOYEES(FIRST\_NAME);

SELECT index\_name

FROM user\_indexes

WHERE table\_name = :table

Explain plan for fetching details:-

EXPLAIN PLAN FOR SELECT EMPLOYEE\_ID

FROM EMPLOYEES WHERE EMPLOYEE\_ID=110;

TO SEE THE REPORT:

SELECT \* FROM TABLE(DBMS\_XPLAN.DISPLAY);

## Composite Index:-

* It is created on multiple columns are called composite index.

Example:-

CREATE INDEX IND1

ON EXPLOYEES(FIRST\_NAME, SALARY);

## Unique Index:-

* It is created on unique columns are called unique index.
* Example mobile number, email id, ipaddress.

CREATE INDEX IND1

ON

EMPLOYEES(EMAIL);

## Reverse Index:-

* By default Index is Ascending.
* Reverse index orders are created by reverse called reverse index.

CREATED INDEX IND1

ON

EMP(SALARY)REVERSE;

## Function Based Index:-

* It is extension of b-tree index.
* We can create index on column along with function (or) expressions.

CREATE INDEX IND1

ON

EMPLOYEES(UPPER(FIRST\_NAME));

# Index organized table:-

* If you are using IOT it is automatically arranged ascending order.
* You have to use primary key for organization index.

Example:-

CREATE TABLE TEST6 (ID NUMBER(5) PRIMARY KEY , NAME VARCHAR2(10)) organization index;

**When we insert normal column data will be insert like below:-**

|  |  |  |
| --- | --- | --- |
| into TEST1(ID) VALUES(10) |  | **ID** |
| INTO TEST1(ID) VALUES(7) |  | 10 |
| into TEST1(ID)VALUES(9) |  | 7 |
| INTO TEST1(ID) VALUES(2) |  | 9 |
| into TEST1(ID) VALUES(1) |  | 2 |
| INTO TEST1(ID) VALUES(5) |  | 1 |
| SELECT \* FROM DUAL; |  | 5 |

**When we insert IOT column data will be insert like below:-**

|  |  |  |
| --- | --- | --- |
| into TEST1(ID) VALUES(10) |  | **ID** |
| INTO TEST1(ID) VALUES(7) |  | 1 |
| into TEST1(ID)VALUES(9) |  | 2 |
| INTO TEST1(ID) VALUES(2) |  | 5 |
| into TEST1(ID) VALUES(1) |  | 7 |
| INTO TEST1(ID) VALUES(5) |  | 9 |
| SELECT \* FROM DUAL; |  | 10 |

# HINT’S

* Hints will instruct or force to the DATABASE Engine how to execute the query.
* Hits will instruct to the DATABASE to use or not to use the Index.
* OPTIMIZER is the decision maker wheather to read the index or not.
* Optimizer is the Inbuild query plan generation.

Example:-

SELECT /\*+FULL (EMPLOYEES)\*/ FIRST\_NAME FROM EMPLOYEES WHERE EMPLOYEE\_ID =109;

EXPLAIN PLAN FOR SELECT /\*+FULL (EMPLOYEES)\*/ FIRST\_NAME FROM EMPLOYEES WHERE EMPLOYEE\_ID = 109;

SELECT \* FROM TABLE (DBMS\_XPLAN.DISPLAY())

SELECT /\*+INDEX (EMPLOYEES)\*/ FIRST\_NAME FROM EMPLOYEES WHERE EMPLOYEE\_ID =109;

SELECT /\*+INDEX (EMPLOYEES FNAMEINDX)\*/ FIRST\_NAME FROM EMPLOYEES WHERE FIRST\_NAME =’Neena’;

|  |  |
| --- | --- |
| **Hints for Optimization Approaches and Goals** | |
|
| **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). |
| **FIRST\_ROWS** | The FIRST\_ROWS hint explicitly chooses the cost-based approach to optimize a statement block with a goal of best response time (minimum resource usage to return first row). In newer Oracle version you should give a parameter with this hint: FIRST\_ROWS(n) means that the optimizer will determine an executionplan to give a fast response for returning the first n rows. |
| **CHOOSE** | The CHOOSE hint causes the optimizer to choose between the rule-based approach and the cost-based approach for a SQL statement based on the presence of statistics for the tables accessed by the statement |
| **RULE** | The RULE hint explicitly chooses rule-based optimization for a statement block. This hint also causes the optimizer to ignore any other hints specified for the statement block. The RULE hint does not work any more in Oracle 10g. |

|  |  |
| --- | --- |
| **Hints for Access Paths** | |
|
| **FULL** | The FULL hint explicitly chooses a full table scan for the specified table. The syntax of the FULL hint is FULL(table) where table specifies the alias of the table (or table name if alias does not exist) on which the full table scan is to be performed. |
| **ROWID** | The ROWID hint explicitly chooses a table scan by ROWID for the specified table. The syntax of the ROWID hint is ROWID(table) where table specifies the name or alias of the table on which the table access by ROWID is to be performed. (This hint depricated in Oracle 10g) |
| **CLUSTER** | The CLUSTER hint explicitly chooses a cluster scan to access the specified table. The syntax of the CLUSTER hint is CLUSTER(table) where table specifies the name or alias of the table to be accessed by a cluster scan. |
| **HASH** | The HASH hint explicitly chooses a hash scan to access the specified table. The syntax of the HASH hint is HASH(table) where table specifies the name or alias of the table to be accessed by a hash scan. |
| **HASH\_AJ** | The HASH\_AJ hint transforms a NOT IN subquery into a hash anti-join to access the specified table. The syntax of the HASH\_AJ hint is HASH\_AJ(table) where table specifies the name or alias of the table to be accessed.(depricated in Oracle 10g) |
| **INDEX** | The INDEX hint explicitly chooses an index scan for the specified table. The syntax of the INDEX hint is INDEX(table index) where:table specifies the name or alias of the table associated with the index to be scanned and index specifies an index on which an index scan is to be performed. This hint may optionally specify one or more indexes: |
| **NO\_INDEX** | The NO\_INDEX hint explicitly disallows a set of indexes for the specified table. The syntax of the NO\_INDEX hint is NO\_INDEX(table index) |
| **INDEX\_ASC** | The INDEX\_ASC hint explicitly chooses an index scan for the specified table. If the statement uses an index range scan, Oracle scans the index entries in ascending order of their indexed values. |
| **INDEX\_COMBINE** | If no indexes are given as arguments for the INDEX\_COMBINE hint, the optimizer will use on the table whatever boolean combination of bitmap indexes has the best cost estimate. If certain indexes are given as arguments, the optimizer will try to use some boolean combination of those particular bitmap indexes. The syntax of INDEX\_COMBINE is INDEX\_COMBINE(table index). |
| **INDEX\_JOIN** | Explicitly instructs the optimizer to use an index join as an access path. For the hint to have a positive effect, a sufficiently small number of indexes must exist that contain all the columns required to resolve the query. |
| **INDEX\_DESC** | The INDEX\_DESC hint explicitly chooses an index scan for the specified table. If the statement uses an index range scan, Oracle scans the index entries in descending order of their indexed values. |
| **INDEX\_FFS** | This hint causes a fast full index scan to be performed rather than a full table. |
| **NO\_INDEX\_FFS** | Do not use fast full index scan (from Oracle 10g) |
| **INDEX\_SS** | Exclude range scan from query plan (from Oracle 10g) |
| **INDEX\_SS\_ASC** | Exclude range scan from query plan (from Oracle 10g) |
| **INDEX\_SS\_DESC** | Exclude range scan from query plan (from Oracle 10g) |
| **NO\_INDEX\_SS** | The NO\_INDEX\_SS hint causes the optimizer to exclude a skip scan of the specified indexes on the specified table. (from Oracle 10g) |

|  |  |
| --- | --- |
| **Hints for Query Transformations** | |
|
| **NO\_QUERY\_TRANSFORMATION** | Prevents the optimizer performing query transformations. (from Oracle 10g) |
| **USE\_CONCAT** | The USE\_CONCAT hint forces combined OR conditions in the WHERE clause of a query to be transformed into a compound query using the UNION ALL set operator. Normally, this transformation occurs only if the cost of the query using the concatenations is cheaper than the cost without them. |
| **NO\_EXPAND** | The NO\_EXPAND hint prevents the optimizer from considering OR-expansion for queries having OR conditions or IN-lists in the WHERE clause. Usually, the optimizer considers using OR expansion and uses this method if it decides that the cost is lower than not using it. |
| **REWRITE** | The REWRITE hint forces the optimizer to rewrite a query in terms of materialized views, when possible, without cost consideration. Use the REWRITE hint with or without a view list. If you use REWRITE with a view list and the list contains an eligible materialized view, then Oracle uses that view regardless of its cost. |
| **NOREWRITE / NO\_REWRITE** | In Oracle 10g renamed to NO\_REWRITE. The NOREWRITE/NO\_REWRITE hint disables query rewrite for the query block, overriding the setting of the parameter QUERY\_REWRITE\_ENABLED. |
| **MERGE** | The MERGE hint lets you merge views in a query. |
| **NO\_MERGE** | The NO\_MERGE hint causes Oracle not to merge mergeable views. This hint is most often used to reduce the number of possible permutations for a query and make optimization faster. |
| **FACT** | The FACT hint indicated that the table should be considered as a fact table. This is used in the context of the star transformation. |
| **NO\_FACT** | The NO\_FACT hint is used in the context of the star transformation to indicate to the transformation that the hinted table should not be considered as a fact table. |
| **STAR\_TRANSFORMATION** | The STAR\_TRANSFORMATION hint makes the optimizer use the best plan in which the transformation has been used. Without the hint, the optimizer could make a query optimization decision to use the best plan generated without the transformation, instead of the best plan for the transformed query. |
| **NO\_STAR\_TRANSFORMATION** | Do not use star transformation (from Oracle 10g) |
| **UNNEST** | The UNNEST hint specifies subquery unnesting. |
| **NO\_UNNEST** | Use of the NO\_UNNEST hint turns off unnesting for specific subquery blocks. |

|  |  |
| --- | --- |
| **Hints for Join Orders** | |
|
| **LEADING** | Give this hint to indicate the leading table in a join. This will indicate only 1 table. If you want to specify the whole order of tables, you can use the ORDERED hint. Syntax: LEADING(table) |
| **ORDERED** | 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 , the optimizer chooses the order in which to join the tables. You may 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 would allow you to choose an inner and outer table better than the optimizer could. |

# TABLE BACKUP:-

* While taking backup these 2 type only NULL constraints only copy other Constraints will not copy.
* Index will not copy these 2 type.
* When use data pump copy all data & constraint & index will copy.

## Backup table with data:-

CREATE TABLE BACKUP AS SELECT \* FROM EMP;

## Backup table structure only:-

CREATE TABLE BACKUP AS SELECT \* FROM EMP WHERE 1=0;

## Data pump export:-

EXPDP HR/hr@XE TABLES=EMPLOYEES DIRECOTRY=NEW DUMPFILE=EMPLOYEES.DMP LOGFILE=EMPLOYEES.LOG

## Data Pump Import:-

IMPDP RAJESH/ADMIN@XE TABLES=EMPLOYEES DIRECTORY=NEW DUMPFILE=EMP.DMP LOGFILE=DEPRT.LOG

## Column name find DB:-

select table\_name from all\_tab\_columns where column\_name = 'PICK\_COLUMN';

## Table Name find in DB:-

select table\_name from all\_TABLES where TABLE\_name like '%FIRST%';

# VIRTUAL COLUMNS:-

* The values of the virtual column are not stored in the DATA BASE rather its computed at run time when you run QUERIES.
* You can’t update the virtual column values.
* Virtual column can be used in where clause of update and delete statement.

CREATE TABLE EMPLOYEE

(

empl\_id NUMBER,

empl\_nm VARCHAR2(50),

monthly\_sal NUMBER(10,2),

bonus NUMBER(10,2),

total\_sal NUMBER(10,2)

GENERATED ALWAYS AS (monthly\_sal\*12 + bonus)

);

SELECT column\_name, data\_type, data\_length, data\_default, virtual\_column

FROM user\_tab\_cols

WHERE table\_name = 'EMPLOYEE';

COLUMN\_NAME | DATA\_TYPE | DATA\_LENGTH | DATA\_DEFAULT | VIRTUAL\_COLUMN

EMPL\_ID | NUMBER | 22 | null | NO

EMPL\_NM | VARCHAR2 | 50 | null | NO

MONTHLY\_SAL | NUMBER | 22 | null | NO

BONUS | NUMBER | 22 | null | NO

TOTAL\_SAL | NUMBER | 22 | "MONTHLY\_SAL"\*12+"BONUS" | YES

# GLOBAL TEMPORARY TABLE - (GTT):-

* It holds data till a session / Transaction duration only.
* Oracle introduced the global temporary table concept since version 8i.
* the data stored in the global temporary table is **private** to the session.

Types of GTT:-

* ON COMMIT DELETE ROWS:- DEFAULT
* ON COMMIT PRESERVE ROWS :-

## On Commit delete rows :-

* When you commit the transaction the GTT table data will removed table structure remain same.

Private

CREATE GLOBAL TEMPORARY TABLE T1 (NO NUMBER) ON COMMIT DELETE ROWS;

INSERT INTO T1 VALUES (10);

INSERT INTO T1 VALUES (20);

SELECT \* FROM T1;

COMMIT;

## ON COMMIT PRESERVE ROWS :-

* IN this session when you commit data will not lose.
* When you exit the session

CREATE GLOBAL TEMPORARY TABLE T1 (NO NUMBER) ON COMMIT PRESERVE ROWS;

COMMIT; /\*WHEN you commit data will not loss\*/

EXIT; /\*when you exit session will loss\*/

## Things to consider before creating a global temporary table:

These are the most important points to consider before you create a global temporary table.

### 1) DDL operation on global temporary tables

It is not possible to perform a DDL operation (except [TRUNCATE](https://www.oracletutorial.com/oracle-basics/oracle-truncate-table/)) on an existing global temporary table if one or more sessions are currently bound to that table.

First, create a temporary table named temp3:

|  |  |
| --- | --- |
| 1  2  3 | CREATE GLOBAL TEMPORARY TABLE temp3(      id INT  ) ON COMMIT DELETE ROWS; |

Next, insert a new row into the temp3 table:

|  |  |
| --- | --- |
| 1 | INSERT INTO temp3(id) VALUES(1); |

Then, log in to the database in a separate session e.g., using SQL\*Plus and add a column to the temp3 table:

|  |  |
| --- | --- |
| 1  2 | ALTER TABLE temp3  ADD description VARCHAR2(100); |

Oracle issued the following error:

|  |  |
| --- | --- |
| 1 | ORA-14450: attempt to access a transactional temp table already in use |

After that, commit the transaction in the first session:

|  |  |
| --- | --- |
| 1 | COMMIT; |

Finally, perform the DDL operation in the second transaction:

|  |  |
| --- | --- |
| 1  2 | ALTER TABLE temp3  ADD description VARCHAR2(100); |

This time it worked because no session is currently bound to the temp3 table.

### 2) Transactions on transaction-specific global temporary tables

Oracle only allows one transaction at a time on a transaction-specific temporary table.

If you have several autonomous transactions in a single transaction scope, you must commit the previous autonomous transaction before the next transaction can use the table.

### 3) Rollback on transaction-specific global temporary tables

Rolling back (ROLLBACK) on the global temporary table will cause all data entered lost.

### 4) Backup & recovery on global temporary tables

Due to the nature of temporary tables, backup and recovery are not available in case of a system failure.

# --TABLE PARTITION:-

* A single big table divided into smaller pieces is called table partition.
* Each partition is stored separately and can be queried or maintained individually, which significantly improves performance, manageability, and availability of data.
* Partitioning allows tables, indexes, and index-organized tables to be subdivided into smaller pieces,
* enabling these database objects to be managed and accessed at a finer level of granularity.

## 1. **What is Table Partitioning?**

This is particularly beneficial for handling large tables, making queries faster and more efficient by allowing operations to access only relevant partitions rather than scanning the entire table.

Each partition acts like a smaller table with its own indexing and storage. However, from a user's perspective, the table still appears as a single entity.

**When to Partition a Table??**

* Tables greater than 2 GB should always be considered as candidates for partitioning.
* Tables containing historical data, in which new data is added into the newest partition. A typical example is a historical table where only the current month's data is updatable and the other 11 months are read only.
* When the contents of a table need to be distributed across different types of storage devices.

**ADVANTAGES OF PARTITIONS**

* Reducing downtime for scheduled maintenance, which allows maintenance operations to be carried out on selected partitions while other partitions are available to users.
* Reducing downtime due to data failure, failure of a particular partition will no way affect other partitions.
* Partition independence allows for concurrent use of the various partitions for various purposes.
* **Increases performance** by only working on the data that is relevant.
* **Improves availability** through individual partition manageability.
* **Decreases costs** by storing data in the most appropriate manner.
* Is **easy as to implement** as it requires no changes to applications and queries.
* Is a **mature, well proven feature** used by thousands of Oracle customer

To see partition :-

select \* from USER\_TAB\_PARTITIONS;

select \* from DBA\_PART\_KEY\_COLUMNS;

<https://www.complexsql.com/table-partitioning/>

**TYPES**

1. Range partitions.
2. List partitions.
3. Hash partitions.
4. **Composite Partitioning**:

## RANGE PARTITIONS :-

Create table Employee

(emp\_no number(2),

emp\_name varchar(2))

partition by range(emp\_no) /\* partition TYPE\*/

(

partition p1 values less than(100), /\*Partition Name\*/

partition p2 values less than(200),

partition p3 values less than(300),

partition p4 values less than(maxvalue)

);

1.Selecting records from partitioned tables.

Select \* from Employee;

Select \* from Employee partition(p1);

2.Adding new Table partition:

Alter table Employee

add partition p5 values less than(50000);

3.Drop Table partition:

Alter table Employee

drop partition p1;

4.Rename Table partition:

Alter table Employee

Rename partition p1 to p6;

5.Truncate partition:

Alter table Employee

Truncate partition p1;

6.Split partition:

Alter table Employee

Split partition p1 at (5000)

into (partition p10,partition p11);

7.Moving partition:

Alter table Employee

move partition p1 to tablespace ABCD;

## LIST PARTITION:-

Create table Employee

(Emp\_no number(2),

Emp\_name varchar(2)

Emp\_city varchar(10))

partition by list(Emp\_city)

(

partition p1 values(‘CHENNAI’),

partition p2 values(‘MADURAI’),

partition p3 values(‘TRICHY’),

partition p4 values(‘COVAI’,’ THIRUPUUR’),

partition p5 values(default)

);

1.Selecting records from partitioned tables.

Select \* from Employee;

Select \* from Employee partition(p1\_Maharashtra);

2.Adding new partition:

Alter table Employee

add partition p5\_Kerala values(‘Kerala’);

3.Drop partition:

Alter table Employee

drop partition p1\_Maharashtra;

4.Rename  partition:

Alter table Employee

Rename partition p1\_Maharashra to p6\_Maha;

5.Truncate partition:

Alter table Employee

6.Moving partition:

Alter table Employee

move partition p1\_Maharashtra to tablespace ABCD;

## HASH PARTITIONS

* Hash partitioning maps data to partitions based on a hashing algorithm that Oracle applies to the partitioning key that you identify.

Create table Employee

(emp\_no number(2),

emp\_name varchar(2))

partition by hash(emp\_no) partitions 5;

Here oracle automatically gives partition names like

SYS\_P1

SYS\_P2

SYS\_P3

SYS\_P4

SYS\_P5

## EXCHANGE PARTITION

We now switch the segments associated with the source table and the partition in the destination table using the EXCHANGE PARTITION syntax.

ALTER TABLE big\_table2

EXCHANGE PARTITION big\_table\_2007

WITH TABLE big\_table

WITHOUT VALIDATION

UPDATE GLOBAL INDEXES;

The exchange operation should not be affected by the size of the segments involved.

Once this is complete we can drop the old table and rename the new table and all it's constraints.

DROP TABLE big\_table;

RENAME big\_table2 TO big\_table;

ALTER TABLE big\_table RENAME CONSTRAINT big\_table\_pk2 TO big\_table\_pk;

ALTER TABLE big\_table RENAME CONSTRAINT bita\_look\_fk2 TO bita\_look\_fk;

ALTER INDEX big\_table\_pk2 RENAME TO big\_table\_pk;

ALTER INDEX bita\_look\_fk\_i2 RENAME TO bita\_look\_fk\_i;

ALTER INDEX bita\_created\_date\_i2 RENAME TO bita\_created\_date\_i;

## --Sub Partationing:-

* Partition inside a partition called sub partitioning.

create table sub\_pat\_test(emp\_name varchar2(30),job\_id varchar2(30),hire\_date date)

partition by range(hire\_date)

subpartition by list(job\_id)(partition p1 values less than(to\_date('01-01-2003','dd-mm-yyyy'))

(subpartition sp1 values('HR\_REP','PU\_MAN'),subpartition sp11 values(default)),

partition p2 values less than(to\_date('01-01-2004','dd-mm-yyyy'))(

subpartition sp2 values('AC\_ACCOUNT','FI\_ACCOUNT')

subpartition sp22 values(default)

)

partition p3 values less than(to\_date('01-01-2005','dd-mm-yyyy'))(

subpartition sp3 values('SH\_CLERK','ST\_CLERK')

subpartition sp33 values(default)

))

partition p4 values less than(to\_date('01-01-2006','dd-mm-yyyy'))(

subpartition sp4 values('SA\_MAN','PU\_MAN')

subpartition sp44 values(default)

)

partition p5 values less than(maxvalues)(

subpartition sp5 values(default)

)) ;

SELECT \* FROM USER\_TAB\_SUBPARTITIONS WHERE TABLE\_NAME=’EMP’;

# WITH CLAUSES :-

* It was introduced by ORACLE 9i release 2.
* WITH clause allows you to give a sub-query block a name, which can be reffered in several place within the main query.
* When we replace it ‘WITH’ clause the execution time will be decreased and the performance will be increased.

A perfect example for With clause:-

Print depart name last name depart id department id =90;

WITH TABLE1 AS

(SELECT E.LAST\_NAME, d.department\_id, D.DEPARTMENT\_NAME

FROM EMPLOYEES E, DEPARTMENTS D

WHERE e.department\_id = d.department\_id)

SELECT \* FROM TABLE1 WHERE DEPARTMENT\_ID = 90;

|  |  |  |
| --- | --- | --- |
| **LAST\_NAME** | **DEPARTMENT\_NAME** | **DEPARTMENT\_ID** |
| King | Executive | 90 |
| Kochhar | Executive | 90 |
| De Haan | Executive | 90 |

RANK BY SALARY WITH DEPARTMENT\_NAME (deficult question)

WITH TABLE1 AS

(SELECT E.DEPARTMENT\_ID, D.DEPARTMENT\_NAME, E.FIRST\_NAME, E.SALARY

FROM EMPLOYEES E INNER JOIN DEPARTMENTS D

ON E.DEPARTMENT\_ID = D.DEPARTMENT\_ID),

TABLE2 AS

(SELECT \* FROM (SELECT DEPARTMENT\_ID, DEPARTMENT\_NAME, FIRST\_NAME, SALARY, DENSE\_RANK() OVER(PARTITION BY DEPARTMENT\_ID ORDER BY SALARY)AS RK FROM TABLE1)WHERE RK <=3)

SELECT DEPARTMENT\_ID, DEPARTMENT\_NAME, FIRST\_NAME, SALARY FROM TABLE2

# Cluster :-

* It is a data base object that stores the data related to two or more table in a single DISKSPACE or SINGLE memory location.
* The tables which are frequency used for join operation which require more I/O operations we use cluserts so that performance Increases.
* FROM 11G ONWORDS.
* The point being, the tables in a cluster are co-located. This is a physical arrangement. So, for the database to cluster existing tables **we must drop and re-create them**. It is possible to minimise the downtime by building the clustered table under a different name

### ****What is a Cluster?****

A **cluster** is a database storage structure that allows you to store rows from multiple tables together based on a common column (known as the **cluster key**). Instead of storing each table separately, Oracle stores rows with the same cluster key value together in the same physical data block. This helps reduce I/O operations and improves the performance of join operations and queries involving these related tables.

### ****How Does a Cluster Work?****

When tables are joined frequently on a specific column, clustering those tables ensures that rows with the same cluster key value are stored together. This reduces the amount of data Oracle needs to read from disk, as related data is **grouped physically.**

#### Example:

Consider a scenario with two tables, employees and departments, that are often joined by department\_id. By clustering these tables based on department\_id, Oracle can fetch rows from both tables more efficiently when they are queried together.

### ****Creating a Cluster****

#### Syntax for Creating an Indexed Cluster

CREATE CLUSTER emp\_dept\_cluster (department\_id NUMBER) SIZE 512;

* **cluster\_name**: The name of the cluster.
* **cluster\_key\_column**: The column that serves as the cluster key.
* **SIZE**: The estimated size of the data for a given cluster key.

CREATE TABLE employees (

employee\_id NUMBER,

name VARCHAR2(100),

department\_id NUMBER

) CLUSTER emp\_dept\_cluster (department\_id);

CREATE TABLE departments (

department\_id NUMBER,

department\_name VARCHAR2(100)

) CLUSTER emp\_dept\_cluster (department\_id);

## Dcl Data Control Language:-

### Create user:-

Create user userid identified by password;

### Reset password:-

Alter user userid identified by password password expire;

### Lock & unlock :-

ALTER USERID ACCOUNT LOCK;

ALTER USERID ACCOUNT UNLOCK;

### GIVING PRIVLIAGE TO USER:-

GRANT INSERT ON TABLENAME TO USERID;

GRANT SELECT ON TABLENAME TO USERID;

|  |  |
| --- | --- |
| **ROLE** | **PROFILE** |
| USER PRIVILAGES LIKE SELECT INSERT DELETE | PASSWORD TIME, IDEAL TIME, CONNECT TIME. |
|  |  |

#### CREATE USER :

create user john identified by john123;

#### Force User to Change the Password

Alter user john identified by john@123 password expire;

#### LOCK / UNLOCK :

SQL> ALTER USER JOHN ACCOUNT LOCK;

SQL> ALTER USER JOHN ACCOUNT UNLOCK;

ALTER USER hr ACCOUNT UNLOCK;

Alter user hr identified by hr;

#### USER PRIVILEGES CHECKING:

Select \* from session\_privs;

#### GRANT SELECT, INSERT, UPDATE TO USER

SQL > GRANT SELECT, INSERT, UPDATE ON JOHN.TABLENAME to john;

SQL> grant create table to john;

#### REVOKE SYSTEM PRIVILEGES:

REVOKE CREATE TABLE FROM JOHN;

#### REVOKE OBJECT PRIVILEGES:

REVOKE SELECTON DW.SALES FROM JOHN;

#### DELETE USER :

Drop user john;

Drop user john caseced; (caseced means delete all information )

#### USERS AVALIABLE IN DBA:

Select username, account\_status from dba\_users;

#### CREATING ROLE;

SQL> CREATE ROLE MANAGER\_ROLE;

SQL> CREATE ROLE OPERATION\_ROLE;

#### GRANT PRIVILEGES TO ROLE:

grant insert, update, delete on dw.sales\_history to manager\_role;

#### Role privileges to user

SQL> GRANT MANAGER\_ROLE TO PETER;

#### PROFILE CREATIONS :

CREATE PROFILE new\_profile LIMIT PASSWORD\_REUSE\_MAX 10 PASSWORD\_REUSE\_TIME 30;

#### USER BELONG TO WHICH GROUP:-

select default\_tablespace, temporary\_tablespace from dba\_users where username='JOHN';

#### USER BELONGS TO WHICH PROFILE:-

select username, profile from dba\_users where username='JOHN';