Aditya College of Engineering & Technology

Aditya Nagar, ADB Road, Surampalem - 533437

**DATABASE MANAGEMENT SYSTEMS**

**UNIT III: Entity Relationship Model (PART *–* 1)**

**Syllabus:**

**SQL:** basic SQL querying (select and project) using where clause, arithmetic & logical operations, SQL functions (Date and Time, Numeric, String conversion).

Creating tables with relationship, implementation of key and integrity constraints,

nested queries, sub queries, grouping, aggregation, ordering, implementation of different types of joins, view(updatable and non-updatable), relational set operations.

## Basic SQL querying (select and project) using where clause:

* + - The following are the various SQL clauses:

SQL Clause

Group by clause having clause Order by clause

## Group by:

* + SQL group by statement is used to arrange identical data into groups.
  + The group by statement is used with the SQL select statement.
  + The group by statement follows the WHERE clause in a SELECT statement and precedes the ORDER BY clause.

Syntax:

**Select column from table\_name where column group by column, order by column;**

**Sample table: product**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| PRODUCT | COMPANY | QTY | RATE | COST |
| Item 1 | Com 1 | 2 | 10 | 20 |
| Item 2 | Com 2 | 3 | 25 | 75 |
| Item 3 | Com 1 | 2 | 30 | 60 |
| Item 4 | Com 3 | 5 | 10 | 50 |
| Item 5 | Com 2 | 2 | 20 | 40 |

**Example:**

* + Select company count (\*) from product group by company;

**Output:**

|  |  |
| --- | --- |
| Com 1 | 2 |
| Com 2 | 3 |
| Com 3 | 5 |

## Having clause:

* + Having clause is used to specify a search condition for a group or an aggregate.

Having clause is used in a group by clause, if you are not using group by clause then you can use having function like a where clause.

**Syntax:**

**Select column1, column2 from table\_name**

**Where conditions**

**Group by column1, column2 Having conditions**

**Order by column1, column2;**

**Example:**

* + **select company count (\*) from product**

**Group by company Having count (\*) > 2;**

**Output:**

|  |  |
| --- | --- |
| Com 3 | 5 |
| Com 2 | 2 |

## Order by clause:

The order by clause sorts the result \_set in ascending or descending order.

### Syntax:

Select column1, column2, from table\_name

Where condition

Order by column1, column2…asc;

### Sample table:

Take a student table

**Example:**

**Select \* from student order by name;**

**Output:**

|  |  |  |
| --- | --- | --- |
| NAME | ID | CITY |
| Alekhya | 501 | Hyderabad |
| Deepti | 502 | Guntur |
| Rasi | 503 | Nellore |

## ​ SQL Where clause:

* + - A where clause in SQL is a data manipulation language statement.
    - Where clauses are not mandatory clauses of SQL DML statements but it can be used to limit the number of rows affected by a SQL DML statement or returned by query.
    - Actually, it follows the records.it returns only those queries which the specific conditions.

### Syntax:

**Select column1, column2, …………column from table\_name where[condition];**

* + - Where clause uses same conditional selection.

|  |  |
| --- | --- |
| = | Equal to |
| > | Greater than |
| < | Less than |
| > = | Greater than or equal to |
| < = | Less than or equal to |
| < > | Not equal to |

## Arithmetic and logical operations:

**SQL operators:**

* + - SQL statements generally contain some reserved words or characters that are used to perform operations such as arithmetic and logical operations etc. Their reserved words are known as operators.

**SQL arithmetic operator:**

* + - We can use various arithmetic operators on the data stored in tables.
    - Arithmetic operators are:

|  |  |
| --- | --- |
| + | Addition |
| - | Subtraction |
| / | Division |
| \* | Multiplication |
| % | modulus |

1. **Addition (+):**

It is used to perform addition operation on data items.

**Sample table:**

|  |  |  |  |
| --- | --- | --- | --- |
|  | EMP\_ID | EMP\_NAME | SALARY |
| 1 | Alex | 25000 |
| 2 | John | 55000 |
| 3 | Daniel | 52000 |
| 4 | Sam | 12312 |

* + select emp id, emp\_name, salary, salary+100 as “salary +100” from

addition;

**Output:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | EMP\_ID | EMP\_NAME | SALARY | SALARY+100 |
| 1 | Alex | 25000 | 25100 |
| 2 | John | 55000 | 55100 |
| 3 | Daniel | 52000 | 52100 |
| 4 | Sam | 12312 | 12412 |

* + Here we have done addition of 100 to each emp‟s salary.

1. **Subtraction (-):**
   * It is used to perform subtraction on the data items.

**Example:**

Select emp\_id, emp\_name, salary, salary-100 as “salary-100” from

subtraction;

|  |  |  |  |
| --- | --- | --- | --- |
| EMP\_ID | EMP\_NAME | SALARY | SALARY-100 |
| 1 | Alex | 25000 | 24900 |
| 2 | John | 55000 | 54900 |
| 3 | Daniel | 52000 | 51900 |
| 4 | Sam | 90000 | 89900 |

Here we have done subtraction of 100 for each emp‟s salary.

## Division (/):

* + The division function is used to integer division (x is divided by y).an integer value is returned.

**Example:**

* + **Select emp\_id, emp\_name, salary, salary/100 as “salary/100” from division;**

|  |  |  |  |
| --- | --- | --- | --- |
| EMP\_ID | EMP\_NAME | SALARY | Salary/100 |
| 1 | Alex | 25000 | 250 |
| 2 | John | 55000 | 550 |
| 3 | Daniel | 52000 | 520 |
| 4 | Sam | 90000 | 900 |

## Multiplication (\*):

* + It is used to perform multiplication of data items.
  + **Select emp\_id, emp\_name, salary, salary\*100 as “salary\*100” from multiplication;**

|  |  |  |  |
| --- | --- | --- | --- |
| EMP\_ID | EMP\_NAME | SALARY | SALARY\*100 |
| 1 | Alex | 25000 | 2,500,000 |
| 2 | John | 55000 | 5,500,000 |
| 3 | Daniel | 52000 | 5,200,000 |
| 4 | Sam | 90000 | 9,000,000 |

* + Here we have done multiplication of 100 to each emp‟s salary.

## Modulus (%):

* + It is used to get remainder when one data is divided by another.
  + **Select emp\_id, emp\_name, salary, salary%25000 as “salary%25000” from modulus;**

**Output:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | EMP\_ID | EMP\_NAME | SALARY | SALARY%25000 |
| 1 | Alex | 25000 | 0 |
| 2 | John | 55000 | 5000 |
| 3 | Daniel | 52000 | 2000 |
| 4 | Sam | 90000 | 15000 |

* + Here we have done modulus operation to each emp‟s salary.

## ​ Logical operations:

* + - Logical operations allow you to test for the truth of a condition.
    - The following table illustrates the SQL logical operator.

|  |  |
| --- | --- |
| OPERATOR | MEANING |
| ALL | Returns true if all comparisons are true |
| AND | Returns true if both expressions are true |
| ANY | Returns true if any one of the comparisons is  true |
| BETWEEN | Return true if the operand is within a range |
| IN | Return true if the operand is equal to one of  the values in a list |
| EXISTS | Return true if the sub query contains any rows |

# AND:

The AND operator allows you to construct multiple condition in the WHERE clause of an SQL statement such as select.

* + The following example finds all employees where salaries are greater than the 5000 and less than 7000.
    - **Select first\_name, last\_name, salary from employees where salary>5000 AND salary<7000 order by salary;**

**Output:**

|  |  |  |
| --- | --- | --- |
| FIRST\_NAME | LAST\_NAME | SALARY |
| John | Wesley | 6000 |
| Eden | Daniel | 6000 |
| Luis | Popp | 6900 |
| Shanta | Suji | 6500 |

# ALL:

The ALL operator compares a value to all values in another value set.

* + The following example finds all employees whose salaries are greater than all salaries of employees.

**EX:**

**select first\_name, last\_name, salary from employees where salary>=ALL (select salary from employees where department\_id =8) order by salary DESC;**

**Output:**

|  |  |  |
| --- | --- | --- |
| FIRST\_NAME | LAST\_NAME | SALARY |
| Steven | King | 24000 |
| John | Russel | 17000 |
| Neena | Kochhar | 14000 |

# ANY:

The ANY operator compares a value to any value in a set ascending to condition.

The following example statement finds all employees whose salaries are greater than the average salary of every department.

EX:

**select first\_name, last\_name, salary from employees where salary >ANY (select avg (salary) from employees‟ group by department\_id) order by first\_name, last\_name;**

**Output:**

|  |  |  |
| --- | --- | --- |
| FIRST\_NAME | LAST\_NAME | SALARY |
| Alexander | Hunold | 9000.00 |
| Charles | Johnson | 6200.00 |
| David | Austin | 4800.00 |
| Eden | Flip | 9000.00 |

## Between:

* + The between operator searches for values that are within a set of values.
  + For example, the following statement finds all employees where salaries are between 9000 and 12000.

EX:

**select first\_name, last\_name, salary from employees where salary between 9000 AND 12000 order by salary;**

**Output:**

|  |  |  |
| --- | --- | --- |
| FIRST\_NAME | LAST\_NAME | SALARY |
| Alexander | Hunold | 9000.00 |
| Den | Richards | 10000.00 |
| Nancy | Prince | 12000.00 |

# IN:

* + The IN operator compares a value to list of specified values. The IN operator return true if compared value matches at least one value in the list.
  + The following statement finds all employees who work in department \_id 8 or 9. EX:

**select first\_name, last\_name, department\_id from employees where department\_id IN (8,9) order by department\_id;**

**Output:**

|  |  |  |
| --- | --- | --- |
| FIRST\_NAME | LAST\_NAME | DEPARTMENT\_ID |
| John | Russel | 8 |
| Jack | Livingstone | 8 |
| Steven | King | 9 |
| Neena | Kochhar | 9 |

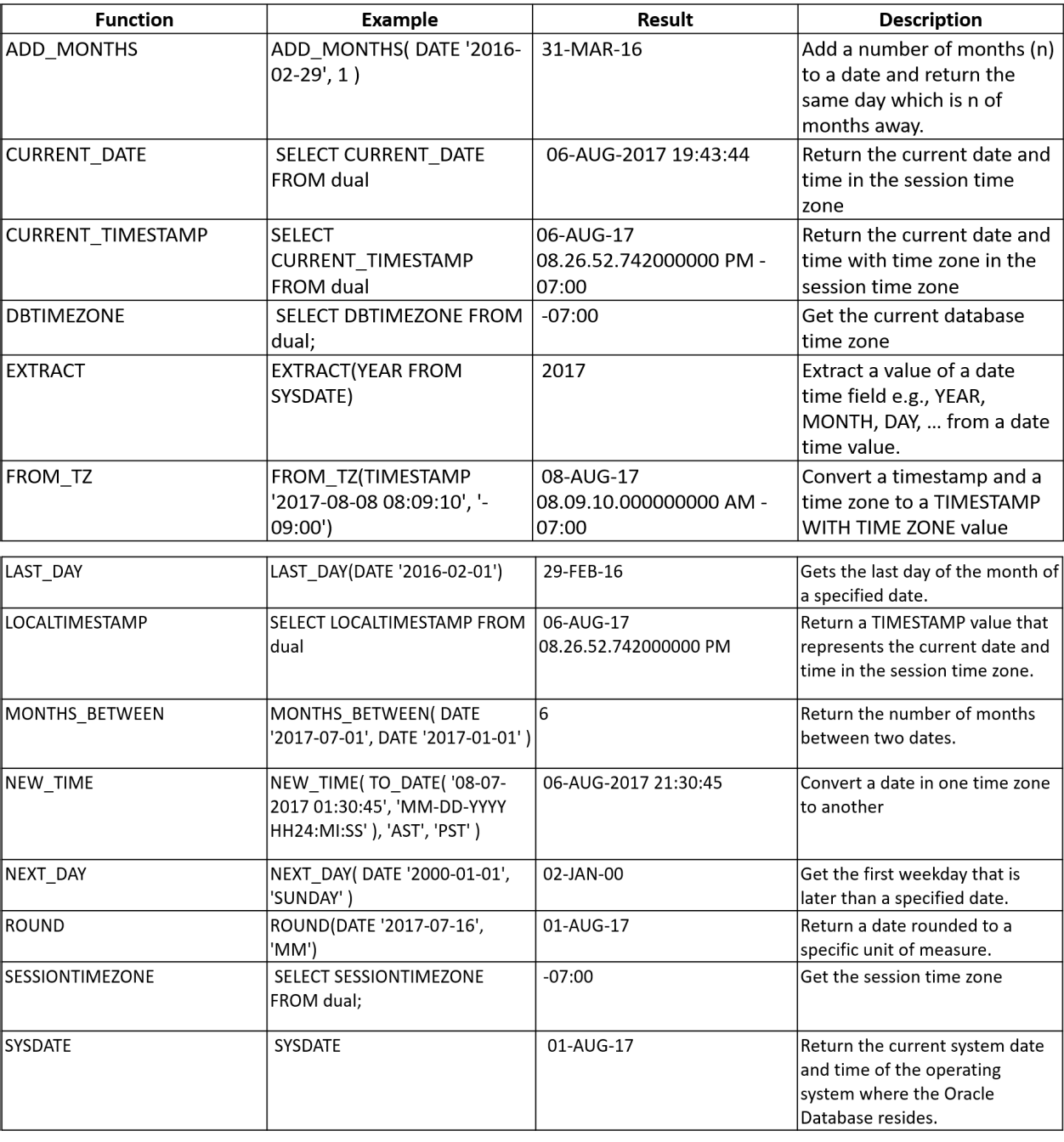
## Exists:

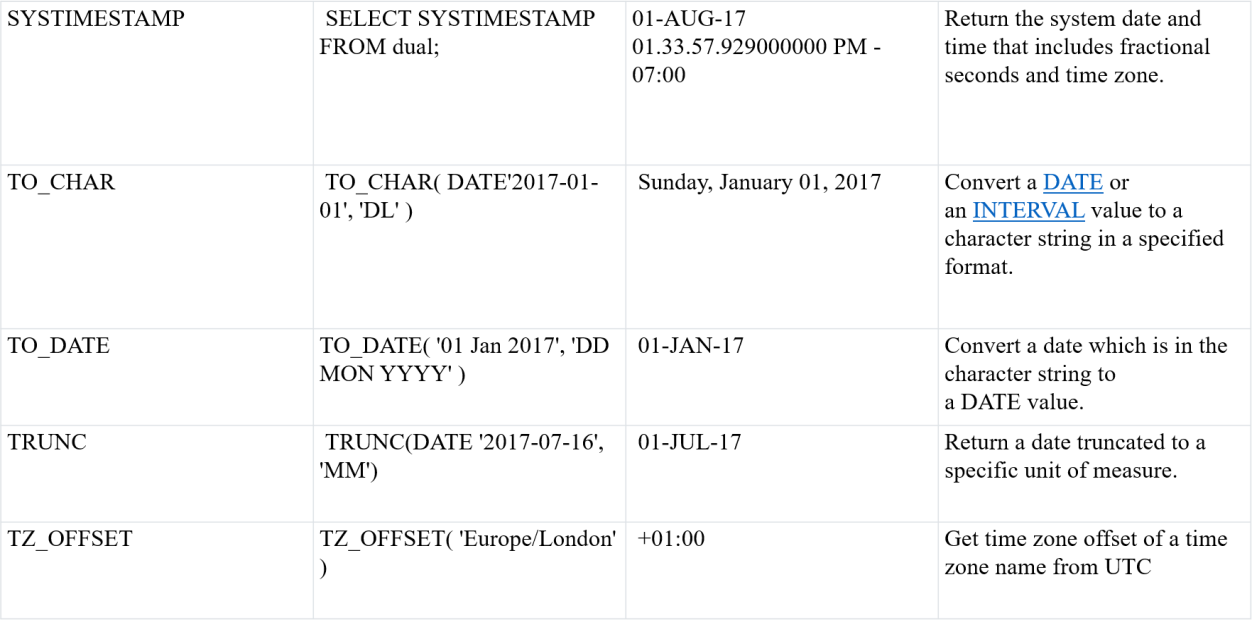
* + The EXISTS operator tests if a sub query contains any rows.
  + For example, the following statement finds all employees who have dependents.
  + select first\_name, last\_name from employees where EXISTS (select 1 from dependent d where d.employee\_id=e.employee\_id);

|  |  |
| --- | --- |
| FIRST\_NAME | LAST\_NAME |
| Steven | King |
| Neena | Kochhar |
| Alexander | Hunold |

## ​ SQL FUNCTIONS (Date & Time, Numeric, Aggregate, String conversions):

**DATE & TIME FUNCTIONS:**



****

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

### Some important date and time functions are below:

**Sysdate:** It generates the system date**. Ex: Select sysdate from dual;**

Output: 05-DEC-2021.

**ADD\_MONTHS:** This function returns a date after adding data with specified no of months. EX: **Select ADD\_MONTHS („2017-02-29‟,1) from dual;**

Output: 31-MAR-17.

**Select add\_months(sysdate,3) from dual;**

Output: 05-MAR-22.

**CURRENT\_DATE:** This function displays the current date.

**Ex: Select CURRENT\_DATE from dual;**

Output: 05-DEC-2021.

**NEXT\_DAY:** This function represents both day and date and returns the day of the next given day.

EX: **Select NEXT\_DAY(SYSDATE,‟MONDAY‟) from dual;**

Output: 07-DEC-21.

**LAST\_DAY:** This function returns a day corresponding last day of months.

EX: **Select LAST\_DAY (sysdate) from dual;**

Output: 31-DEC-21.

**MONTHS\_BETWEEN:** It is used to find no of months between two given dates.

EX: **Select MONTHS\_BETWEEN(‟16-APRIL-2021‟,‟16-AUGUST-2021) from dual;**

**Output: -4.**

**ROUND:** It gives the nearest value or round off value for the argument pass. (or) It returns a date rounded to a specific unit of measure.

EX: **Select ROUND(‟26-NOV-21‟,‟YYYY‟) from dual;**

**Output**: 01-JAN-22.

**TRUNC:** This function returns the date with the time(co-efficient) portion of the date truncated to the unit specified.

EX: **Select TRUNC (sysdate, ‟MM‟) from dual;**

**Output:** 01-DEC-21.

**TO\_DATE:** This function converts date which is in the character string to a date value.

EX: **Select TO\_DATE (‟01 jan 2017‟,‟DD MM YYYY‟) from dual;**

**Output:** 01-JAN-17.

**TO\_CHAR:** This function converts DATE or an INTERVAL value to a character string in a specified format.

EX: **Select TO\_CHAR (sysdate,‟DD MM YYYY‟) from dual;**

**Output:** 05 12 2021.

**LEAST:** This function displays the oldest date present in the argument list.

EX: **Select LEAST(‟01-march-2021‟,‟16-feb-2021‟,‟28-dec-2021‟) from dual;**

**Output:** 01-MAR-21.

**GREATEST:** This function displays the latest date present in the argument list.

EX: **Select GREATEST (‟01-march-2021‟,‟16-feb-2021‟,‟28-dec-2021‟) from dual;**

**Output:** 28-DEC-21.

## Aggregate Functions:

Aggregate Functions take a collection of values as input and returns a single value.

1. Count ()
2. Sum ()
3. Avg ()
4. Max ()
5. Min ()
6. **Count ():** This function returns number of rows returned by a query.

**Syntax: Select count(column\_name)**

**From table\_name Where condition);**

**Example: Select count (distinct manager\_id) from employees;**

1. **Sum ():** It will add/ sum all the column values in the query.

**Syntax: Select sum (column\_name)**

**From table\_name Where condition);**

**Example: Select sum(salaries) from employees;**

1. **Avg ():** Avg function used to calculate average values of the set of rows.

**Syntax: Select avg (column\_name)**

**From table\_name Where condition);**

**Example: Select avg(salary) from employees;**

1. **Max ():** This function is used to find maximum value from the set of values.

**Syntax: Select max (column\_name)**

**From table\_name Where condition);**

**Example: Select max (salary) from employees;**

1. **Min ():** This function is used to find minimum value from the set of values.

**Syntax: Select min (column\_name)**

**From table\_name**

**Where condition);**

**Example: Select min (salary) from employees;**

# SQL NUMERIC FUNCTIONS:

Numeric functions are used to perform operations on numbers and return numbers. Following are some of the Numeric functions

1. **ABS ():** It returns the absolute value of a number. EX: **select ABS (-243.5) from dual;** OUTPUT: 243.5
2. **ACOS ():** It returns the cosine of a number. EX: **select ACOS (0.25) from dual;**

OUTPUT: 1.318116071652818

1. **ASIN ():** It returns the arc sine of a number. EX: **select ASIN (0.25) from dual**;

OUTPUT: 0.253680255142

1. **CEIL ():** It returns the smallest integer value that is a greater than or equal to a number. EX: **select CEIL (25.77) from dual;**

OUTPUT: 26

1. **FLOOR ():** It returns the largest integer value that is a less than or equal to a number. EX: **select FLOOR (25.75) from dual;**

OUTPUT: 25

1. **TRUNCATE ()**: This does not work for SQL server. It returns the truncated to 2 places right of the decimal point.

EX: **select TRUNCATE (7.53635, 2) from dual;**

OUTPUT: 7.53

1. **MOD ():** It returns the remainder when two numbers are divided. EX: **select MOD (55,2) from dual;**

OUTPUT: 1.

1. **ROUND ():** This function rounds the given value to given number of digits of precision. EX: **select ROUND (14.5262,2) from dual;**

OUTPUT: 14.53.

1. **POWER ():** This function gives the value of m raised to the power of n. EX: **select POWER (4,9) from dual;**

OUTPUT: 262144.

1. **SQRT ():** This function gives the square root of the given value n. EX: **Select SQRT (576) from dual;**

OUTPUT: 24.

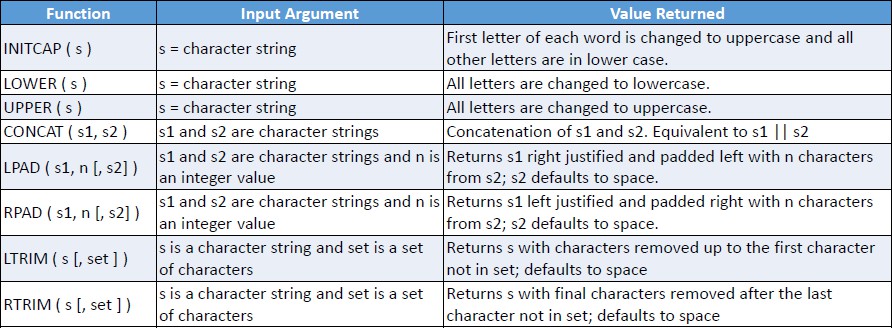
1. **LEAST ():** This function returns least integer from given set of integers. EX: **select LEAST (1,9,2,4,6,8,22) from dual;**

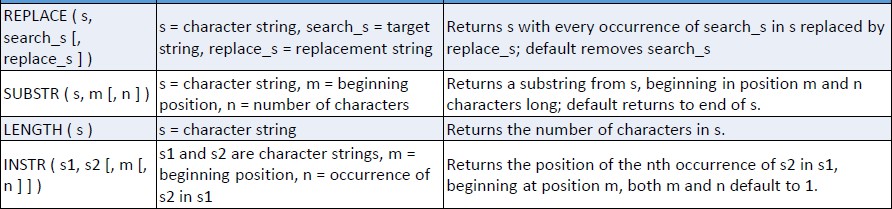
OUTPUT: 1.

1. **GREATEST ():** This function returns greatest integer from given set of integers. EX: **select GREATEST (1,9,2,4,6,8,22) from dual;**

OUTPUT: 22

# STRING CONVERSION FUNCTIONS OF SQL:

****



**String Functions are used to perform an operation on input string and return the output string. Following are the string functions**

1. **CONCAT ():** This function is used to add two words (or) strings.

EX: **select „database‟ ||‟ „|| „management system‟ From dual;**

OUTPUT: „database management system‟

1. **INSTR ():** This function is used to find the occurrence of an alphabet. EX: **instr („database system‟,‟ a‟) from dual;** OUTPUT: 2 (the first occurrence of „a‟)
2. **LOWER ():** This function is used to convert the given string into lowercase. EX: **select lower („DATABASE‟) from dual;**

OUTPUT: database

1. **UPPER ():** This function is used to convert the lowercase string into uppercase. EX: **select upper („database „) from dual;**

OUTPUT: DATABASE

1. **LPAD ():** This function is used to make the given string of the given size by adding the given symbol. EX: > **lpad („system „, 8, „0‟) from dual;**

OUTPUT: 00system

1. **RPAD ():** This function is used to make the given string as long as the given size by adding the given symbol on the right.

EX: **rpad („system „,8, „0„) from dual;**

OUTPUT: system00

1. **LTRIM ():** This function is used to cut the given substring from the original string. EX: **ltrim („database „, „data „) from dual;**

OUTPUT: base

1. **RTRIM ():** This function is used to cut the given substring from the original string. EX: **rtrim („database „, „base „) from dual;**

OUTPUT: data.

1. **INITCAP ():** This function returns the string with first letter of each word starts with uppercase.

EX: **Select INITCAP („data base management system‟) from dual;**

OUTPUT: Data Base Management System.

1. **LENGTH ():** Tis function returns the length of the given string.

EX: **select LENGTH („SQ LANGUAGE‟) from dual;**

OUTPUT: 11.

1. **SUBSTR ():** This function returns a portion of a string beginning at the character position. EX: **select SUBSTR („MY WORLD IS AMAZING‟,12,3) from dual;**

OUTPUT: AM.

1. **TRANSLATE ():** This function returns a string after replacing some set of characters into another set. EX: **select TRANSLATE („Delhi is the capital of India‟,‟i‟,‟a‟) from dual;** OUTPUT: Delha as the capatal andaa.

## Review Questions:

* 1. Why NULL values are needed in databases?
  2. Discuss GROUPBY and HAVING clauses with an example. And also give the constraints related to their usage.
  3. Consider the SAILOR DATABASE

Sailors (Sid: string, sname: string, rating: integer, age: real) Boats (bid: integer, bname: string, colour: string)

Reserves (Sid: integer, bid: integer, day: date)

Based on the above schema, write the corresponding SQL queries for the following? Find the colours of boats reserved by Lubber.

Find the names of sailors who have reserved at least one boat. Find the names of sailors who have reserved a red or green boat.

Find the names of the sailors who have reserved both a red boat and a green boat.

Find names of sailors who have reserved all boats.

* 1. Write about the usability of „group by‟ and „having‟ clauses In SQL.
  2. How would you use the operators IN, EXISTS, UNIQUE, ANY and ALL in writing nested queries? Why are they useful? Explain with an example.
  3. Explain commands with respect to SQL: (i) Rename (ii) Alter (iii) View
  4. Explain two aggregate functions of SQL.
  5. Explain the following SQL constructs with examples:

(1) order by (2) group by and having (3) as select (4) schema

* 1. Write SQL Queries for following set of tables:

EMPLOYEE (EmpNo, Name, DoB, Address, Gender, Salary, DNumber) DEPARTMENT (DNumber, Dname, ManagerEmpNo, ManagerStartDate). Display the Age of „male‟ employees.

Display all employees in Department named „Marketing‟. Display the name of highest salary paid „female‟ employee. Which employee is oldest manger in company?

Display the name of department of the employee „SMITH.

* 1. Describe creating and modifying relations using SQL. Give examples for each.
  2. Explain about integrity constraints over relations.
  3. Consider the following insurance database, where the primary keys are underlined.

Construct the following SQL queries for this relational database. person (driver-id#, name, address)

car (license, model, year)

accident (report-number, date, location) owns (driver-id#, license)

participated (driver-id, car, report-number, damage-amount)

* + 1. Find the total number of people who owned cars that were involved in accidents in 1989.
    2. Find the number of accidents in which the cars belonging to “John Smith” were involved.
    3. Add a new accident to the database; assume any values for required attributes.
    4. Delete the Mazda belonging to “John Smith”.
    5. Update the damage amount for the car with license number “AABB2000” in the accident with report number “AR2197” to $3000.
  1. List the SQL functions for string conversions.
  2. Write SQL statements for following:

Student (Enrno, name, courseId, emailId, cellno) Course (courseId, course\_nm, duration)

1. Add a column city in student table.
2. Find out list of students who have enrolled in “computer” course.
3. List name of all courses with their duration.
4. List name of all students start with „a‟.
5. List email Id and cell no of all mechanical engineering students.
   1. How to define a domain constraint? Give an example.
   2. What is an integrity constraint? Explain its enforcement by DBMS with illustrative
   3. List the data types supported by SQL.
   4. Demonstrate the use of DISTINCT keyword in SQL select statement.
   5. Consider the following database schema to write nested queries in SQL Supplier (id, name, city)

Parts (pno, pname, pdescription) Supply (id, pno, cost)

1. Find the names of the parts supplied by “RamRaj”
2. Find the names of the suppliers who supply “Nuts”
3. Find the cost of bolts being supplied by Nagpur suppliers.

## References:

* Raghurama Krishnan, Johannes Gehrke, *Database Management Systems*, 3rd Edition, Tata McGraw Hill.
* C.J. Date, *Introduction to Database Systems*, Pearson Education.
* Elmasri Navrate, *Fundamentals of Database Systems*, Pearson Education.

DATABASE MANAGEMENT SYSTEMS UNIT – III : ER MODEL

**UNIT-3 PART- II SQL**

* + **Creating Tables with relationship**
  + **Implementation of key and integrity constraints**
  + **Relational Set Operations**
  + **Implementation of different types of Joins**
  + **Views(Updatable and Non Updatable)**
  + **Grouping, Aggregation, Ordering**
  + **Nested queries, Sub queries**

**I.INTRODUCTION TO CREATING TABLES WITH RELATIONSHIP**

**Creating tables using CREATE command:**

This command is used to create a database and its objects such as Tables, Views, Procedures, Triggers etc. It defines each column of the table uniquely. Each column has minimum of **three** attributes, a column name , data type and size.

**Syntax: CREATE TABLE <table\_name>**

**(**

**column \_name 1 DATATYPE 1 (SIZE),**

**column \_name 2 DATATYPE 2 (SIZE),**

**:**

**column \_name n DATATYPE N (SIZE) );**

**CREATING TABLES WITH RELATIONSHIP**

When we want to create tables with relationship , we need to use **Referential integrity constraints**. The referential integrity constraint **enforces relationship** between tables.

-It **designates** a column or combination of columns as a **Foreign key**.

-The foreign key establish a **relationship** with a specified primary or unique key in another table called the **Referenced key**.

* When referential integrity is enforced , it prevents from..
  1. Adding records to a related table if there is no associated record in the primary table.
  2. Changing values in a primary table that result in orphaned records in a related table.
  3. Deleting records from a primary table if there are matching related records.

Note: The table containing the foreign key is called the child table and the table containing the referenced key is called the Parent table.

**SYNTAX: CREATE TABLE <tablename>(**

**col\_name1 datatype[size] , col\_name2 datatype[size] ,**

**:**

**col\_name n datatype[size],**

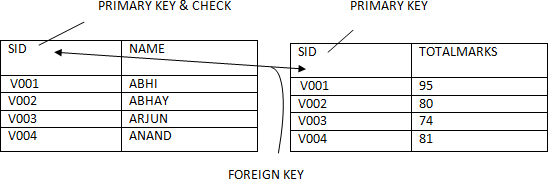
**FOREIGN KEY(column\_name) REFERENCES <parent\_table\_name>(column\_name));**

**EX:** SQL> CREATE TABLE marks(

sid VARCHAR2(4),

marks NUMBER(3), PRIMARY KEY(sid),

FOREIGN KEY(sid) REFERENCES student1(sid));



**II.IMPLEMENTATION OF KEY AND INTEGRITY CONSTRAINTS**

**Data constraints:** All business of the world run on business data being gathered, stored and analyzed. Business managers determine a set of business rules that must be applied to their data prior to it being stored in the database/table of ensure its integrity.

For instance , no employee in the sales department can have a salary of less than Rs.1000/- . Such rules have to be enforced on data stored. If not, inconsistent data is maintained in database.

Note: It is used to impose business rules on DBs.

It allows to enter only valid data.

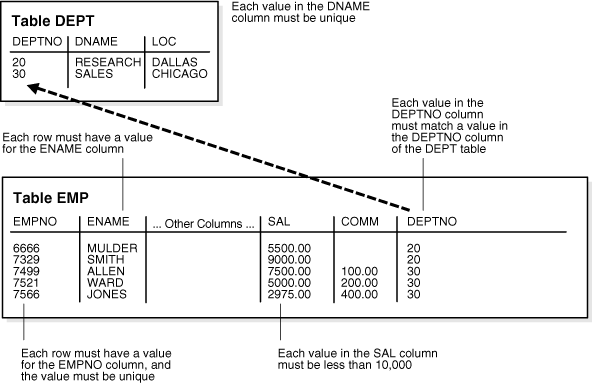
**Various types of Integrity Constraints:**

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

**You can use constraints to do the following:**

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

**Example for Integrity Constraints :-**



**Constraints are categorized as follows.**

1. **Domain integrity constraints** - A domain means a set of values assigned to a column. i.e A set of permitted values. Domain constraints are handled by

Defining proper **data type**

Specifying **not null** constraint

Specifying **check** constraint.

Specifying **default** constraint

**Not null –**indicates that a column cannot store NULL value.

**Check** – Ensures that the value in column meets a specific condition.

**Default**- prevents **null** value in column when value is not provided in column by user. So, it assigns default value or globally assigned value.

1. **Entity integrity constraints – are TWO types**

**Unique constraint** –It defines a Entity or column as a UNIQUE for particular table.

And ensures that each row of a column must have a UNIQUE value

or name.

**Primary key constraint** – This avoids **duplicate** and **null** values. It combination of a NOT NULL and UNIQUE.

1. **Referential integrity constraints**

**Foreign key** –indicates the relationship between child and parent tables.

This Constraint are always attached to a column not a table.

We can add constraints in **two** ways.

**Column level :-**

 Constraint is declared immediately declaring column.

 Define with each column

 Use column level to declare constraint for single column.

 Composite key cannot be defined at column level.

**Table level :-**

constraint is declared after declaring all columns.

use table level to declare constraint for combination of columns.(i.e composite key)

 **not null** cannot be defined. Another type is possible at **Alter level**

 constraint is declared with ALTER command.

When we use this , make sure that the table should not contain data.

To add these constraints , we can use constraint with label or with out label. TWO BASIC TYPES ------- 1. Constraint WITH NAME

2.constraints WITHOUT NAME.

* 1. **​Declaring Constraint at “TABLE” level (Constraints with label) Syntax :- CREATE TABLE <table name>**

**(**

**col\_name1 DATATYPE(SIZE) ,**

**…..,**

**col\_nameN DATATYPE(SIZE) ,**

**CONSTRAINT <cons\_lable> NAME \_OF\_ THE\_CONSTRAINT [column\_list] );**

* 1. **Declaring Constraint at “Column” level (Constraints with label) Syntax :-**

**col\_name DATATYPE(SIZE) CONSTRAINT <cons\_lable> NAME \_OF\_ THE\_CONSTRAINT**

* 1. **​Declaring Constraint at “TABLE” level (Constraints without label) Syntax :-**

**CREATE TABLE <table name> (**

**col\_name1 DATATYPE(SIZE) ,**

**…..,**

**col\_nameN DATATYPE(SIZE) ,**

**NAME \_OF\_ THE\_CONSTRAINT [column\_list] );**

* 1. **​Declaring Constraint at “Column” level (Constraints without label) Syntax :-**

**col\_name DATATYPE(SIZE) NAME \_OF\_ THE\_CONSTRAINT**

* 1. **​Adding Constraint to a table at “Alter” level (Constraints with label)**

A constraint can be added to a table at any time after the table was created by using ALTER TABLE statement , using **ADD** clause.

Syntax:

**ALTER TABLE <table\_name> ADD CONSTRAINT cont\_label NAME \_OF\_ THE\_CONSTRAINT (column);**

* 1. **​Declaring Constraint at “Alter” level (Constraints without label)**

Syntax:

**ALTER TABLE <table\_name> ADD NAME \_OF\_ THE\_CONSTRAINT (column); Note:’ Constraint ‘ clause is not required when constraints declared without a label.**

* 1. **​NOT NULL:**
     + It ensures that a table column cannot be left empty.
     + Column declared with NOT NULL is a mandatory column i.e data must be entered.
     + The NOT NULL constraint can only be applied at column level.
     + It allows DUPLICATE data.
     + Used to avoid null values into columns.

**NOTE: I**t is applicable at **COLUMN LEVEL only.**

**SYNTAX: column\_name DATATAYPE[SIZE] NOT NULL**

**EX** 1 : CREATE TABLE table\_notnull(

sid NUMBER(4) **NOT NULL**, // COLUMN LEVEL sname VARCHAR2(10));

SQL> SELECT \*FROM table\_notnull; SID SNAME

501 GITA

502 RAJU

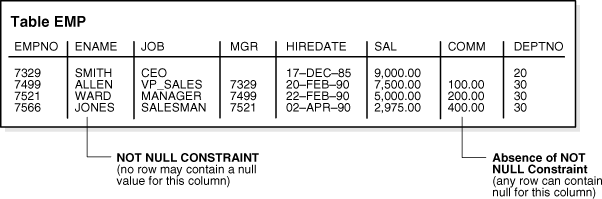
503

503

504

Here, SID column not allowed any null values and it can allow duplicate values , but sname can allows it.

**Ex 2:**



* 1. **CHECK :**
* Used to impose a conditional rule a table column.
* It defines a condition that each row must satisfy.
* Check constraint **validates data** based on a condition .

**Limitations :-**

* Value entered in the column **should not violate the condition**.
* Check constraint allows **null** values.
* Check constraint can be declared at **table** level or **column** level.
* There is no limit to the number of CHECK constraints that can be defined on a condition.
  + Conditions should not contain/not applicable to pseudo columns like ROWNUM, SYSDATE etc.
  + Condition should not access columns of another table

**//CONSTRAINT @ COLUMN LEVEL**

**SYNTAX**: column\_name DATATYPE (SIZE) **CHECK** (condition) // without label Here, we are creating a table with two columns such as Sid, sname.

Ex: CREATE TABLE check\_column(

sid VARCHAR2(4) **CHECK** (sid LIKE 'C%' AND LENGTH(sid)=4), // **without label**

sname VARCHAR2(10));

Here, **sid** should start with **‘C** ‘and a length of sid is exactly **4** characters.

SQL> SELECT \*FROM check\_column; SID SNAME

C501 MANI C502 DHANA C503 RAVI C504 RAJA

**// with label**

**Syntax**: Column\_name DATATYPE(SIZE) CONSTRAINT constaint\_label CHECK(condtion)

CREATE TABLE check\_column (

sid VARCHAR2(4) CONSTRAINT ck **CHECK** (sid LIKE 'C%' AND LENGTH(sid)=4), sname VARCHAR2(10) );

**//CHECK @TABLE LEVEL**

CREATE TABLE check\_table (

sid VARCHAR2(4) , sname VARCHAR2(10),

CHECK (sid LIKE'C%' AND LENGTH(sid)=4), CHECK(sname LIKE '%A'));

Here, **sid** should start with **‘C** ‘and a length of sid is exactly **4** characters. And **sname** should ends with letter ‘ A’

SQL> SELECT \*FROM check\_table;

SID SNAME

C401 ABHIL**A** C401 ANITH**A** C403 NANDHITHA C522 LOHITHA

// **with label**

CONSTRAINT ck1 CHECK (sid LIKE'C%' AND LENGTH(sid)=4), CONSTRAINT ck2 CHECK(sname LIKE '%A'));

**@ ALTER LEVEL**

**Here, we add check constraint to new table with columns.**

SQL> CREATE TABLE check\_alter(sid VARCHAR2(4),

Sname VARCHAR2(10));

**//CHECK @ ALTER LEVEL: / / CONSTRAINT WITHOUT NAME**

**SQL> ALTER TABLE <table name>ADD CHECK (condition ); SQL> ALTER TABLE check\_alter ADD CHECK ( sid like ‘C%’ );**

SYNTAX: **ALTER TABLE <table\_name> ADD CONSTRAINT cont\_name CHECK(cond);**

SQL> ALTER TABLE check\_alter ADD CONSTRAINT ck CHECK ( sid LIKE 'C%');

**ANOTHER EXAMPLE FOR TABLE LEVEL CONSTRAINT**

**Here, We create table with THREE columns**

**ADD CHECK CONSTRAINT @ TABLE LEVEL (AT THE END OF TABLE DEFINITION)**

**MARKS IN BETWEEN 0 AND 100.**

**SQL> CREATE TABLE marks2 ( sid VARCHAR2(4),**

**sec VARCHAR2(2),**

**marks NUMBER(3),**

**CHECK(marks>0 AND marks<=100) ); DROP @ CHECK CONSTRAINT**

**SYNTAX: ALTER TABLE <table\_name> DROP CONSRAINT cont\_name;**

SQL> ALTER TABLE check\_table DROP CONSTRAINT ck;

**DEFAULT**

**-**If values are not provided for table column , default will be considered.

-This prevents NULL values from entering the columns , if a row is inserted without a value for a column.

-The default value can be a literal, an expression, or a SQL function.

-The default expression must match the data type of the column.

- The DEFAULT constraint is used to provide a default value for a column.

-The default value will be added to all new records IF no other value is specified. **Syntax**: Column\_name datatype (size) **DEFAULT** <value/expression/function> Ex: MIDDLENAME VARCHAR(10) **DEFAULT** 'UNAVAILABLE'

CONTACTNO NUMBER(10) **DEFAULT** 9999999999

This defines what value the column should use when no value has been supplied explicitly when inserting a record in the table.

**CREATE TABLE tab\_default( sid NUMBER(10),**

**contactno number(10) DEFAULT 9999999999);**

**Add data to table:**

Insert into tab\_default (sid,contactno) values(501,9493949312); Insert into tab\_student(sid) values(502);

Insert into tab\_student(sid) values(503);

Insert into tab\_student(sid,sname) values(504,9393949412);

**Select \* from tab\_default; SID CONTACTNO**

**---------- ------------------**

**501 9493949312**

**502 9999999999**

**503 9999999999**

**504 9393949412**

* + 1. **UNIQUE**
       - Columns declared with UNIQUE constraint **does not accept duplicate values**.
       - One table can have a number of unique keys.
       - Unique key can be defined on more than one column i.e composite unique key
       - A composite key UNIQUE key is always defined at the table level only.
       - By default UNIQUE columns accept null values unless declared with NOT NULL constraint
       - Oracle automatically creates UNIQUE index on the column declared with UNIQUE constraint
       - UNIQUE constraint can be declared at column level and table level.

**UNIQUE@ COLUMN LEVEL**

**SYNTAX: column\_name DATA\_TYPE(SIZE) UNIQUE**

CREATE TABLE table\_unique(

**//UNIQUE @ TABLE LEVEL**

sid NUMBER(4) **UNIQUE**, sname VARCHAR2(10));

**SYNTAX: UNIQUE(COLUMN\_LIST);**

CREATE TABLE table\_unique2(

sid NUMBER(4),

sname VARCHAR2(10) ,

**UNIQUE(sid,sname));**

SQL> SELECT \*FROM TABLE\_UNIQUE2; SID SNAME

401 RAMU

**402 SITA // Here , these two records are distinct not the same.**

**402 GITHA**

403 GITHA

404 RAMU

**Unique @ ALTER level:**

Alter table table\_unique ADD UNIQUE (sid) // with out label

Alter table table\_unique ADD CONSTRAINT uq UNIQUE(sid) // with label

**DROP UNIQUE @ TABLE LEVEL**

**SQL> ALTER TABLE table\_unique2 DROP UNIQUE(sid,sname);**

**Now , we removed unique constraint , so now this table consists duplicate data.**

**//UNIQUE@ ALTER LEVEL (here, the table contains duplicates, so it is not works)**

**//delete data from table\_unique2**

SQL> DELETE FROM table\_unique2;

**PRIMARY KEY constraint :-**

PRIMARY KEY is one of the **candidates** key , which uniquely identifies a record in a table.

-used to define key column of a table.

-it is provided with an automatic index.

-A primary key constraint combines a NOT NULL and UNIQUE behavior in one declaration.

**Characterstics of PRIMARY KEY :-**

 There should be at the most one Primary Key or Composite primary key per table.

 PK column do not accept null values.

 PK column do not accept duplicate values.

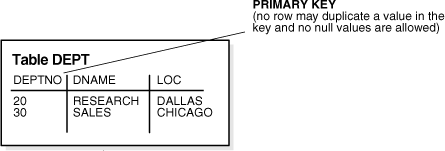
 RAW,LONG RAW,VARRAY,NESTED TABLE,BFILE columns cannot be declared with PK

 If PK is composite then uniqueness is determined by the combination of columns.

 A composite primary key cannot have more than 32 columns

 It is recommended that PK column should be short and numeric.

 Oracle automatically creates Unique Index on PK column

**EX:**

**// PRMARY KEY @ COLUMN LEVEL**

**SYNTAX : column\_name DATA\_TYPE(SIZE) PRIMARY KEY**

SQL> CREATE TABLE student1 (

sid VARCHAR2(4) **PRIMARY KEY**

CHECK (sid LIKE 'V%' AND LENGTH(sid)=4 ) , name VARCHAR2(10));

SQL> DESC student1;

Name Null? Type

SID NOT NULL VARCHAR2(4) NAME VARCHAR2(10)

**CASE 2: ADD PRIMARY KEY @ ALTER LEVEL**

SQL> CREATE TABLE student2( sid VARCHAR2(4),

name VARCHAR2(10));

**SYNTAX: ALTER TABLE <tablename> ADD PRIMARY KEY (col\_name); SQL> ALTER TABLE student2 ADD PRIMARY KEY(sid);**

**Table altered.**

**SQL> DESC student2;**

**Name Null? Type**

**----------------------------------------- -------- ---------------------------- SID NOT NULL VARCHAR2(4) NAME VARCHAR2(10)**

**CASE 3 : ADD PRIMARY KEY @ TABLE LEVEL**

**here, we can create a simple and composite primary keys;**

**SYNTAX: CREATE TABLE < tablename>( col\_name1 datatype[size],**

**col\_name2 datatype[size],**

**:**

**col\_namen datatype[size], PRIMARY KEY (col\_name);**

**//SIMPLE PRIMARY KEY @ TABLE LEVEL**

SQL> CREATE TABLE student3(

**SQL> DESC student3;**

Name Null? Type

sid VARCHAR2(4), name VARCHAR2(10),

marks NUMBER(3), PRIMARY KEY(sid) );

SID NOT NULL VARCHAR2(4) NAME VARCHAR2(10)

MARKS NUMBER(3)

**// COMPOSITE PRIMARY KEY @ TABLE LEVEL SYNTAX:**

**CREATE TABLE < tablename>( col\_name1 datatype[size],**

**col\_name2 datatype[size],**

**:**

**col\_namen datatype[size],**

**PRIMARY KEY (col\_name1,col\_name2….colmn\_name n);**

SQL> CREATE TABLE student4(

sid VARCHAR2(4), name VARCHAR2(10),

marks NUMBER(3),

PRIMARY KEY(sid,name) ); // WITH OUT LABEL

 CONSTRAINT pk PRIMARY KEY(sid,name) // WITH LABEL

**SQL> DESC STUDENT4;**

**Name Null? Type**

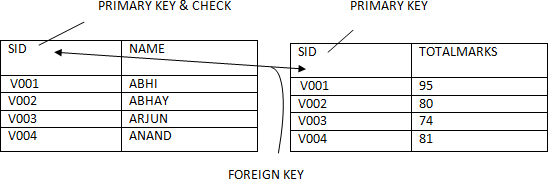
**----------------------------------------- -------- ---------------------------- SID NOT NULL VARCHAR2(4)**

**NAME NOT NULL VARCHAR2(10) MARKS NUMBER(3)**

**FOREIGN KEY Constraint:-**

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

***EX:* TABLES: STYDENT 1 & MARKS 1**

****

***CHILD TABLE ‘ MARKS1’ :***

**ADDING PRIMARY AND CHECK CONSTRAINT @ CREATE LEVEL**

SQL> CREATE TABLE marks1(

sid VARCHAR2(4) PRIMARY KEY CHECK( sid LIKE 'V%' AND LENGTH(sid)=4),

marks NUMBER(3) );

**// ADDING PRIMARY AND FOREIGN KEY @ TABLE/ CREATE LEVEL**

**SYNTAX: CREATE TABLE <tablename>(**

**col\_name1 datatype[size] , col\_name2 datatype[size] ,**

**:**

**col\_name n datatype[size],**

**FOREIGN KEY(column\_name) REFERENCES <parent\_table\_name>(column\_name)); EX:** SQL> CREATE TABLE marks3(

sid VARCHAR2(4),

marks NUMBER(3), PRIMARY KEY(sid),

FOREIGN KEY(sid) REFERENCES student1(sid));

**SQL> DESC MARKS3;**

**Name Null? Type**

**----------------------------------------- -------- ---------------------------- SID NOT NULL VARCHAR2(4) MARKS NUMBER(3)**

**Query : ADD CHECK CONSTRAINT @ ALTER LEVEL ( ON EXISTING TABLE) MARKS IN BETWEEN 0 AND 100.**

**SQL> ALTER TABLE marks3 ADD CHECK ( marks>0 AND marks< =100 );**

**ADD CHECK CONSTRAINT @ TABLE LEVEL (AT THE END OF TABLE DEFINITION)**

**MARKS IN BETWEEN 0 AND 100.**

**SQL> CREATE TABLE marks3 ( sid VARCHAR2(4),**

**sec VARCHAR2(2),**

**marks NUMBER(3),**

**CHECK(marks>0 AND marks<=100) );**

**//ADDING FOREIGN KEY @ ALTER LEVEL**

**SQL> ALTER TABLE marks1 ADD FOREIGN KEY (sid) REFERENCES STUDENT1(sid);**

**SQL> desc marks1;**

**Name Null? Type**

**----------------------------------------- -------- ---------------------------- SID NOT NULL VARCHAR2(4) MARKS NUMBER(3)**

**Note :-**

 PRIMARY KEY cannot be dropped if it referenced by any FOREIGN KEY constraint.

If PRIMARY KEY is dropped with CASCADE option then along with PRIMARY KEY referencing FOREING KEY is also dropped.

PRIMARY KEY column cannot be dropped if it is referenced by some FOREIGN KEY.

PRIMARY KEY table cannot be dropped if it is referenced by some FOREIGN KEY.

PRIMARY KEY table cannot be truncated if it is referenced by some FOREIGN KEY.

**Note::** Once the primary key and foreign key relationship has been created then you can not remove any parent record if the dependent childs exists.

**USING ON DELETE CASCADE**

By using this clause you can remove the parent record even if childs exists. Because when ever you remove parent record oracle automatically removes all its dependent records from child table, if this clause is present while creating foreign key constraint.

**Ex: Consider twe tables dept(parent) and emp(child) tables. TABLE LEVEL**



**SQL> create table emp(empno number(2), ename varchar(10), deptno number(2), primary key(empno), foreign key(deptno) references dept(deptno) on delete cascade); // without label**

**SQL> create table emp(empno number(2), ename varchar(10), deptno number(2), constraint pk primary key(empno), constraint fk foreign key(deptno) references dept(deptno) on delete cascade); // with label**

**ALTER LEVEL**

**SQL> alter table emp add foreign key(deptno) references dept(deptno) on delete cascade;**

**SQL> alter table emp add constraint fk foreign key(deptno) references dept(deptno) on delete cascade;**

**Enabling/Disabling a Constraint:**

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

Disable constraint

Performing the DML operation DML operation

Enable constraint

**Disabling Constraint:- Syntax :-**

**ALTER TABLE <tabname> DISABLE CONSTRAINT <constraint\_name> ; Example :-**

**SQL>ALTER TABLE student1 DISABLE CONSTRAINT ck ; SQL>ALTER TABLE mark1 DISABLE PRIMARY KEY CASCADE;**

**NOTE:-**

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

**Enabling Constraint :- Syntax :-**

**ALTER TABLE <tabname> ENABLE CONSTRAINT <name>**

**Example :-**

**SQL>ALTER TABLE student1 ENABLE CONSTRAINT ck;**

**III. SET OPERATIONS IN SQL**

SQL supports few Set operations to be performed on table data. These are used to get meaningful results from data, under different special conditions. The SET operators combine the results of two or more component queries into one result. Queries containing SET operators are called Compound Queries.

The number of columns and data types of the columns being selected must be identical in all the SELECT statements used in the query. The names of the columns need not be identical.

All SET operators have equal precedence. If a SQL statement contains multiple SET operators, the oracle server evaluates them from left (top) to right (bottom) if no parentheses explicitly specify another order.

**Introduction**

SQL set operators allows combine results from two or more SELECT statements. At first sight this looks similar to SQL joins although there is a big difference. SQL joins tends to combine columns i.e. with each additionally joined table it is possible to select more and more columns. SQL set operators on the other hand combine rows from different queries with strong preconditions .

* Retrieve the same number of columns and
* The data types of corresponding columns in each involved SELECT must be compatible (either the same or with possibility implicitly convert to the data types of the first SELECT statement).

**Set operator types**

According to SQL Standard there are following Set operator types:

* UNION ---returns all rows selected by either query. To return all rows from multiple tables and eliminates any duplicate rows.
* UNION ALL-- returns all rows from multiple tables including duplicates.
* INTERSECT – returns all rows common to multiple queries.
* MINUS—returns rows from the first query that are not present in second query.

Note: Whenever these operators used select statement must have

* + Equal no. of columns.
  + Similar data type columns.

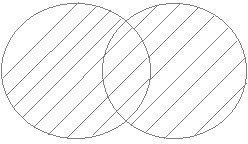
**Syntax :-**

**SELECT statement 1**

**UNION / UNION ALL / INTERSECT / MINUS**

**SELECT statement 2 ; Rules :-**

1. No of columns returned by first query must be equal to no of columns returned by second query
2. Corresponding columns datatype type must be same.
3. **UNION**
   * UNION operator combines data returned by two SELECT statement.
   * Eliminates duplicates.
   * Sorts result.
   * This will combine the records of multiple tables having the same structure.



**Example :-**

1. **SQL>SELECT job FROM emp WHERE deptno=10**

**UNION**

**SELECT job FROM emp WHERE deptno=20 ;**

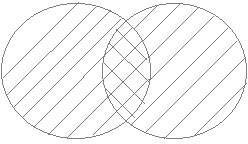
1. **SQL>SELECT job,sal FROM emp WHERE deptno=10**

**UNION**

**SELECT job,sal FROM emp WHERE deptno=20 ORDER BY sal ; NOTE:-** ORDER BY clause must be used with last query.

1. SQL> select \* from student1 **union** select \* from student2;
2. **UNION ALL**

This will combine the records of multiple tables having the same structure but including duplicates. IT is similar to UNION but it includes duplicates.



**Example :-**

**SQL>SELECT job FROM emp WHERE deptno=10 UNION ALL**

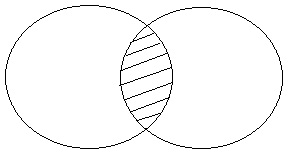
**SELECT job FROM emp WHERE deptno=20 ;**

SQL> select \* from student1 **union all** select \* from student2;

1. **INTERSECT**

This will give the common records of multiple tables having the same structure.

INTERSECT operator returns common values from the result of two SELECT statements.



**Example:-**

Display common jobs belongs to 10th and 20th departments ?

**EX 1: SQL>SELECT job FROM emp WHERE deptno=10**

**INTERSECT**

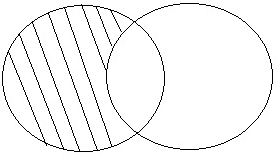
**SELECT job FROM emp WHERE deptno=20;**

**EX2: SQL> select \* from student1 intersect select \* from student2;**

1. **MINUS**

This will give the records of a table whose records are not in other tables having the same structure.

MINUS operator returns values present in the result of first SELECT statement and not present in the result of second SELECT statement.



**Example:-**

Display jobs in 10th dept and not in 20th dept ?

**EX1: SQL>SELECT job FROM emp WHERE deptno=10 MINUS**

**SELECT job FROM emp WHERE deptno=20;**

E**x2: SQL> select \* from student1 minus select \* from student2;**

**UNION vs JOIN :-**

**UNION JOIN**

Union combines data

Union is performed on similar structures on

Join relates data

Join can be performed also be performed dissimilar structures also

1. **SQL JOINS**

A SQL JOIN is an Operation , used to retrieve data from multiple tables. It is performed whenever two or more tables are joined in a SQL statement. so, SQL Join clause is used to combine records from two or more tables in a database. A JOIN is a means for combining fields from two tables by using values common to each. Several operators can be used to join tables,

such as =, <>, <=, >=, !=, BETWEEN, LIKE, and NOT; these all to be used to join tables. However, the most common operator is the equal symbol.

**SQL Join Types:**

There are different types of joins available in SQL:

* + **INNER JOIN**: Returns rows when there is a match in both tables.
  + **OUTER JOIN** : Returns all rows even there is a match or no match in tables.
    - **LEFT JOIN/LEFT OUTER JOIN**: Returns all rows from the left table,

even if there are no matches in the right table.

-**RIGHT JOIN/RIGHT OUTER JOIN** : Returns all rows from the right table, even if

there are

tables.

no matches in the left table.

-**FULL JOIN/FULL OUTER JOIN** : Returns rows when there is a match in one of the

* + **SELF JOIN**: It is used to join a table to itself as if the table were two tables, temporarily renaming at least one table in the SQL statement.
  + **CARTESIAN JOIN or CROSS JOIN** : It returns the Cartesian product of the sets of records from the two or more joined tables.

Based on **Operators**, **The Join** can be classified as

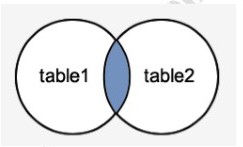
* + - **Inner join or Equi Join**
    - **Non-Equi Join**
* **NATURAL JOIN:** It is performed only when common column name is same. In this,no need to specify join condition explicitly , ORACLE automatically performs join operation on the column with same name.
  1. **​SQL INNER JOIN (simple join)**

**It is the most common type of SQL join. SQL INNER JOINS return all rows from multiple tables where the join condition is met.**

**Syntax**

**SELECT columns FROM table1 INNER JOIN table2 ON table1.column = Table2.column; Visual Illustration**

**In this visual diagram, the SQL INNER JOIN returns the shaded area:**



**The SQL INNER JOIN would return the records where table1 and table2 intersect. Let's look at some data to explain how the INNER JOINS work with example.**

**We have a table called SUPPLIERS with two fields (supplier\_id and supplier\_name). It contains the following data:**

**supplier\_id supplier\_name 10000 ibm**

**10001 hewlett packard**

**10002 microsoft**

**10003 nvidia**

**We have another table called ORDERS with three fields (order\_id, supplier\_id, and order\_date).**

**It contains the following data:**

|  |  |  |
| --- | --- | --- |
| **order\_id** | **supplier\_id** | **order\_date** |
| **500125** | **10000** | **2003/05/12** |
| **500126** | **10001** | **2003/05/13** |
| **500127** | **10004** | **2003/05/14** |

**Example of INNER JOIN:**

**Q: List supplier id, name and order id of supplier.**

**SELECT s.supplier\_id, s.supplier\_name, od.order\_date FROM suppliers s INNER JOIN orders od ON s.supplier\_id = od.supplier\_id;**

**This SQL INNER JOIN example would return all rows from the suppliers and orders tables where there is a matching supplier\_id value in both the suppliers and orders tables. Our result set would look like this:**

|  |  |  |
| --- | --- | --- |
| **supplier\_id** | **name** | **order\_date** |
| **10000** | **ibm** | **2003/05/12** |
| **10001** | **hewlett packard** | **2003/05/13** |

**The rows for Microsoft and NVIDIA from the supplier table would be omitted, since the supplier\_id's 10002 and 10003 do not exist in both tables.**

**The row for 500127 (order\_id) from the orders table would be omitted, since the supplier\_id 10004 does not exist in the suppliers table.**

* 1. **OUTER JOIN:**

Inner / Equi join returns only matching records from both the tables but not unmatched record, An Outer join retrieves all row even when one of the column met join condition.

**Types of outer join:**

1. LEFT JOIN/LEFT OUTER JOIN
2. RIGHT JOIN/RIGHT OUTER JOIN
3. FULL JOIN/FULL OUTER JOIN
   * 1. **LEFT OUTER JOIN**

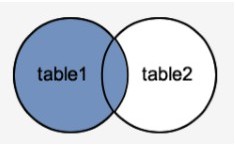
**This type of join returns all rows from the LEFT-hand table specified in the ON condition and only those rows from the other table where the joined fields are equal (join condition is met).**

**Syntax**

**SELECT columns FROM table1 LEFT [OUTER] JOIN table2 ON table1.column = table2.column;**

**Visual Illustration**

**In this visual diagram, the SQL LEFT OUTER JOIN returns the shaded area:**

****

**The SQL LEFT OUTER JOIN would return the all records from table1 and only those records from table2 that intersect with table1.**

**Example**

**SELECT suppliers.supplier\_id, suppliers.supplier\_name, orders.order\_date FROM suppliers LEFT OUTER JOIN orders ON suppliers.supplier\_id = orders.supplier\_id;**

**This LEFT OUTER JOIN example would return all rows from the suppliers table and only those rows from the orders table where the joined fields are equal.**

**supplier\_id supplier\_name order\_date**

**--------------**

**----------------------**

**-----------------------**

**10000 ibm 2003/05/12**

**10001 hewlett packard 2003/05/13**

**10002 microsoft <null>**

**10003 nvidia <null>**

**The rows for Microsoft and NVIDIA would be included because a LEFT OUTER JOIN was used. However, you will notice that the order\_date field for those records contains a**

**<null> value.**

**2.2 SQL RIGHT OUTER JOIN**

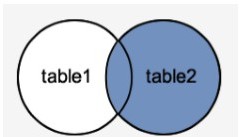
**This type of join returns all rows from the RIGHT-hand table specified in the ON condition and only those rows from the other table where the joined fields are equal (join condition is met).**

**Syntax**

**SELECT columns FROM table1 RIGHT [OUTER] JOIN table2 ON table1.column = table2.column;**

**In some databases, the RIGHT OUTER JOIN keywords are replaced with RIGHT JOIN. Visual Illustration**

**In this visual diagram, the SQL RIGHT OUTER JOIN returns the shaded area:**

****

**The SQL RIGHT OUTER JOIN would return the all records from table2 and only those records from table1 that intersect with table2.**

**Example**

**SELECT orders.order\_id, orders.order\_date, suppliers.supplier\_name FROM suppliers RIGHT OUTER JOIN orders ON suppliers.supplier\_id = orders.supplier\_id;**

**This RIGHT OUTER JOIN example would return all rows from the orders table and only those rows from the suppliers table where the joined fields are equal.**

**If a supplier\_id value in the orders table does not exist in the suppliers table, all fields in the suppliers table will display as <null> in the result set.**

|  |  |  |
| --- | --- | --- |
| **order\_id**  **------------** | **order\_date**  **---------------** | **supplier\_name**  **-----------------** |
| **500125** | **2013/05/12** | **ibm** |
| **500126** | **2013/05/13** | **hewlett packard** |
| **500127** | **2013/05/14** | **<null>** |

**The row for 500127 (order\_id) would be included because a RIGHT OUTER JOIN was used. However, you will notice that the supplier\_name field for that record contains a**

**<null> value.**

* 1. **SQL FULL OUTER JOIN**

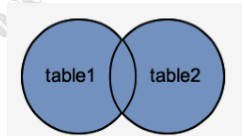
**This type of join returns all rows from the LEFT-hand table and RIGHT-hand table with nulls in place where the join condition is not met.**

**Syntax**

**SELECT columns FROM table1 FULL [OUTER] JOIN table2 ON table1.column = table2.column; In some databases, the FULL OUTER JOIN keywords are replaced with FULL JOIN.**

**Visual Illustration**

**In this visual diagram, the SQL FULL OUTER JOIN returns the shaded area:**

****

**The SQL FULL OUTER JOIN would return the all records from both table1 and table2. Example**

**Here is an example of a SQL FULL OUTER JOIN:**

**Query : Find supplier id, supplier name and order date of suppliers who have ordered.**

**SELECT suppliers.supplier\_id, suppliers.supplier\_name, orders.order\_date FROM suppliers FULL OUTER JOIN orders ON suppliers.supplier\_id = orders.supplier\_id;**

**This FULL OUTER JOIN example would return all rows from the suppliers table and all rows from the orders table and whenever the join condition is not met, <nulls> would be extended to those fields in the result set.**

**If a supplier\_id value in the suppliers table does not exist in the orders table, all fields in the orders table will display as <null> in the result set. If a supplier\_id value in the orders table does not exist in the suppliers table, all fields in the suppliers table will display as**

**<null> in the result set.**

|  |  |  |
| --- | --- | --- |
| **supplier\_id**  **-----------------** | **supplier\_name**  **--------------------** | **order\_date**  **---------------------** |
| **10000** | **ibm** | **2013/05/12** |
| **10001** | **hewlett packard** | **2013/05/13** |
| **10002** | **microsoft** | **<null>** |
| **10003** | **nvidia** | **<null>** |
| **<null>** | **<null>** | **2013/05/14** |

**The rows for Microsoft and NVIDIA would be included because a FULL OUTER JOIN was used. However, you will notice that the order\_date field for those records contains a**

**<null> value.**

**The row for supplier\_id 10004 would be also included because a FULL OUTER JOIN was used. However, you will notice that the supplier\_id and supplier\_name field for those records contain a <null> value.**

**Equi join :**

When the Join Condition is based on EQUALITY (=) operator, the join is said to be an Equi join. It is also called as **Inner Join.**

Syntax

**Select col1,col2,…From <table 1>,<table 2>Where <join condition with ‘=’ > .**

**Ex.Query : Find supplier id, supplier name and order date of suppliers who have ordered . select s.supplierid, s.uppliername ,o.order\_date from suppliers s, orders o where s.supplierid =o.supplierid.**

|  |  |  |
| --- | --- | --- |
| **supplier\_id** | **name** | **order\_date** |
| **10000** | **ibm** | **2003/05/12** |
| **10001** | **hewlett packard** | **2003/05/13** |

**Non Equi Join :-**

When the join condition based on other than equality operator , the join is said to be a Non-Equi join.

**Syntax:-**

**Select col1,col2,……. From <table 1>,<table 2>**

**Where <join condition > [AND <join cond> AND <cond>** **]**

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

**Query : Find supplier id,supplier name and order date in between 50025 and 500127.**

sql> select s.supplier\_id,s.supplier\_name,o.order\_date from suppliers s , orders o where o.order\_id between 500125 and 500127;

SUPPLIER\_ID SUPPLIER\_N ORDER\_DAT

10000 ibm 12-may-03

10000 ibm 13-may-03

10000 ibm 14-may-03

10001 hewlett 12-may-03

10001 hewlett 13-may-03

10001 hewlett 14-may-03

10002 microsoft 12-may-03

10002 microsoft 13-may-03

10002 microsoft 14-may-03

10003 nvidia 12-may-03

10003 nvidia 13-may-03

10003 nvidia 14-may-03

**Query : Find supplier id,supplier name and order date above 500126.**

sql> select s.supplier\_id,s.supplier\_name,o.order\_date from suppliers s , orders o where o.order\_id >500126;

SUPPLIER\_ID SUPPLIER\_NO ORDER\_DAT

|  |  |  |
| --- | --- | --- |
| 10000 | ibm | 14-may-03 |
| 10001 | hewlett | 14-may-03 |
| 10002 | microsoft | 14-may-03 |
| 10003 | nvidia | 14-may-03 |

**Self Join :-**

Joining a table to itself is called Self Join.

* + - Self Join is performed when tables having self refrential integrity.
    - To perform Self Join same table must be listed twice with different alias.
    - Self Join is Equi Join within the table.

It is used to join a table to itself as if the table were two tables, temporarily renaming at least one table in the SQL statement.

**Syntax :**

**(Here T1 and T2 refers same table)**

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

Example:

select s1.supplier\_id ,s1.supplier\_name ,s2.supplier\_id from suppliers s1, suppliers s2 where s1.supplier\_id=s2.supplier\_id ;

**supplier\_id supplier\_name supplier\_id**

**-----------------**

**------------------**

**---------------**

**10000 ibm** **10000**

**10001 hewlett packard 10001**

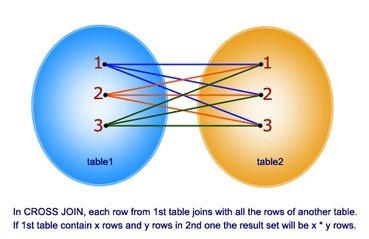
**10002 microsoft** **10002**

**10003 nvidia** **10003**

**CROSS JOIN:**

It returns the Cartesian product of the sets of records from the two or more joined tables. In Cartesian product, each element of one set is combined with every element of another set to form the resultant elements of Cartesian product.

Sytax: SELECT \* FROM <tablename1> CROSS JOIN <tablename2>



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

Example: sql> SELECT \* FROM suppliers CROSS JOIN orders; supplier\_id supplier\_n order\_id supplier\_id order\_dat

|  |  |  |  |
| --- | --- | --- | --- |
|  | 10000 ibm | 500125 | 10000 12-may-03 |
| 10000 ibm | 500126 | 10001 13-may-03 |
| 10000 ibm | 500127 | 10003 14-may-03 |
| 10001 hewlett | 500125 | 10000 12-may-03 |
| 10001 hewlett | 500126 | 10001 13-may-03 |
| 10001 hewlett | 500127 | 10003 14-may-03 |
| 10002 microsoft | 500125 | 10000 12-may-03 |
| 10002 microsoft | 500126 | 10001 13-may-03 |
| 10002 microsoft | 500127 | 10003 14-may-03 |
| 10003 nvidia | 500125 | 10000 12-may-03 |
| **NATURAL JOIN:** | 10003 nvidia | 500126 | 10001 13-may-03 |

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

Syntax: SELECT <column list> FROM table1 NATURAL JOIN table2;

**Example: ( Sailors table)**

**SELECT sid,sname,sid FROM sailors NATURAL JOIN reserves ; //both tables have same column name.**

**SID SNAME SID**

**---------- ---------- ----------**

|  |  |
| --- | --- |
| **22 DUSTIN** | **22** |
| **22 DUSTIN** | **22** |
| **22 DUSTIN** | **22** |
| **22 DUSTIN** | **22** |
| **31 LUBBER** | **31** |
| **31 LUBBER** | **31** |
| **31 LUBBER** | **31** |
| **64 HORTIO** | **64** |
| **64 HORTIO** | **64** |
| **74 HORTIO** | **74** |

1. **VIEWS**

A view in SQL is a logical subset of data from one or more tables. View is used to restrict data access.Data abstraction is usually required after a table is created and populated with data. Data held by some tables might require restricted access to prevent all users from accessing all columns of a table, for data security reasons. Such a security issue can be solved by creating several tables with appropriate columns and assigning specific users to each such table, as required. This answers data security requirements very well but gives rise to a great deal of redundant data being resident in tables, in the database.To reduce redundant data to the minimum possible, Oracle provides Virtual tables which are Views.

**View Definition :-**

A View is a virtual table based on the result returned by a SELECT query.

The most basic purpose of a view is restricting access to specific column/rows from a table thus allowing different users to see only certain rows or columns of a table.

**Composition Of View:-**

A view is composed of rows and columns, very similar to table. The fields in a view are fields from one or more database tables in the database.

SQL functions, WHERE clauses and JOIN statements can be applied to a view in the same manner as they are applied to a table.

**View storage:-**

Oracle does not store the view data. It recreates the data, using the view’s SELECT statement, every time a user queries a view.

A view is stored only as a definition in Oracle’s system catalog.

When a reference is made to a view, its definition is scanned, the base table is opened and the view is created on top of the base table.This, therefore, means that a view never holds data, until a specific call to the view is made. This reduces redundant data on the HDD to a very large extent.

**Advantages Of View:-**

Security:- Each user can be given permission to access only a set of views that contain specific data.

Query simplicity:- A view can drawn from several different tables and present it as a single table turning multiple table queries into single table queries against the view.

Data Integrity:- If data is accessed and entered through a view, the DBMS can automatically check the data to ensure that it meets specified integrity constraints.

**Disadvantage of View:-**

Performance:- Views only create the appearance of the table but the RDBMS must still translate queries against the views into the queries against the underlined source tables. If the view is defined on a complex multiple table query then even a simple query against the view becomes a complicated join and takes a long time to execute.

Types of Views :-

* Simple Views
* Complex Views

Simple Views :-

a View based on single table is called simple view.

Syntax:-

CREATE VIEW <View Name> AS

SELECT<ColumnName1>,<ColumnName2>.. FROM <TableName>

[WHERE <COND>] [WITH CHECK OPTION] [WITH READ ONLY]

Example :-

SQL>CREATE VIEW emp\_v AS

SELECT empno,ename,sal FROM emp ;

Views can also be used for manipulating the data that is available in the base tables[i.e. the user can perform the Insert, Update and Delete operations through view.

Views on which data manipulation can be done are called Updateable Views.

If an Insert, Update or Delete SQL statement is fired on a view, modifications to data in the view are passed to the underlying base table.

For a view to be updatable,it should meet the following criteria:

Views defined from Single table.

If the user wants to INSERT records with the help of a view, then the PRIMARY KEY column(s) and all the NOT NULL columns must be included in the view.

**Inserting record through view :-**

**SQL>INSERT INTO emp\_v VALUES(1,’A’,5000,200) ;**

**Updating record throught view :-**

*Updating a View:*

A view can be updated under certain conditions:

* The SELECT clause may not contain the keyword DISTINCT.
* The SELECT clause may not contain summary functions.
* The SELECT clause may not contain set functions.
* The SELECT clause may not contain set operators.
* The SELECT clause may not contain an ORDER BY clause.
* The FROM clause may not contain multiple tables.
* The WHERE clause may not contain subqueries.
* The query may not contain GROUP BY or HAVING.
* Calculated columns may not be updated.
* All NOT NULL columns from the base table must be included in the view in order for the INSERT query to function.

So if a view satisfies all the above-mentioned rules then you can update a view.

**EX: SQL>UPDATE emp\_v SET sal=2000 WHERE empno=1;**

**Deleting record throught view :-**

**SQL>DELETE FROM emp\_v WHERE empno=1; With Check Option :-**

If VIEW created with WITH CHECK OPTION then any DML operation through that view violates where condition then that DML operation returns error.

**Example :- SQL>CREATE VIEW V2**

**AS**

**SELECT empno,ename,sal,deptno FROM emp WHERE deptno=10**

**WITH CHECK OPTION ;**

Then insert the record into emp table through view V2

**SQL>INSERT INTO V2 VALUES(2323,’RAJU’,4000,20) ;**

The above INSERT returns error because DML operation violating WHERE clause.

**Complex Views :-**

A view is said to complex view

If it based on more than one table

Query contains

AGGREGATE functions DISTINCT clause GROUP BY clause HAVING clause

Sub-queries Constants

Strings or Values Expressions UNION,INTERSECT,MINUS operators.

Example 1 :-

SQL>CREATE VIEW V3 AS

SELECT E.empno,E.ename,E.sal,D.dname,D.loc FROM emp E JOIN dept D

USING(deptno) ;

NON- UPDATABLE VIEWS:

we cannot perform insert or update or delete operations on base table through complex views. Complex views are not updatable views.

Example 2 :-

SQL>CREATE VIEW V2 AS

SELECT deptno,SUM(sal) AS sumsal FROM EMP

GROUP BY deptno;

Destroying a View:-

The DROP VIEW command is used to destroy a view from the database. Syntax:-

DROP VIEW<viewName> Example :-

SQL>DROP VIEW emp\_v;

DIFFERENCES BETWEEN SIMPLE AND COMPLEX VIEWS:

|  |  |
| --- | --- |
| SIMPLE | COMPLEX |
| Created from one table | Created from one or more tables |
| Does not contain functions | Conations functions |
| Does not contain groups of data | Contain groups of data |

**MATERIALIZED VIEW: @ DATAWAREHOUSE SYSTEMS**

A materialized view in Oracle is a database object that contains the results of a query. They are local copies of data located remotely, or are used to create summary tables based on aggregations of a table's data. Materialized views, which store data based on remote tables are also, know as snapshots.

A materialized view can query tables, views, and other materialized views. Collectively these are called master tables (a replication term) or detail tables (a data warehouse term).

For replication purposes, materialized views allow you to maintain copies of remote data on your local node. These copies are read-only. If you want to update the local copies, you have to use the Advanced Replication feature. You can select data from a materialized view as you would from a table or view.

For data warehousing purposes, the materialized views commonly created are aggregate views, single-table aggregate views, and join views.

In replication environments, the materialized views commonly created are primary key, rowid, and subquery materialized views.

SYNTAX:

CREATE MATERIALIZED VIEW view-name BUILD [IMMEDIATE | DEFERRED] REFRESH [FAST | COMPLETE | FORCE ] ON [COMMIT | DEMAND ]

[[ENABLE | DISABLE] QUERY REWRITE] [ON PREBUILT TABLE]

AS

SELECT COLUMN\_LIST FROM TABLE\_NAME;

The BUILD clause options are shown below.

* IMMEDIATE : The materialized view is populated immediately.
* DEFERRED : The materialized view is populated on the first requested refresh.

The following refresh types are available.

* FAST : A fast refresh is attempted. If materialized view logs are not present against the source tables in advance, the creation fails.
* COMPLETE : The table segment supporting the materialized view is truncated and repopulated completely using the associated query.
* FORCE : A fast refresh is attempted. If one is not possible a complete refresh is performed.

A refresh can be triggered in one of two ways.

* ON COMMIT : The refresh is triggered by a committed data change in one of the dependent tables.
* ON DEMAND : The refresh is initiated by a manual request or a scheduled task.

The QUERY REWRITE clause tells the optimizer if the materialized view should be consider for query rewrite operations. An example of the query rewrite functionality is shown below.

The ON PREBUILT TABLE clause tells the database to use an existing table segment, which must have the same name as the materialized view and support the same column structure as the query.

Example:

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.

1. **ORDERING**

USING “ ORDER BY” clause:

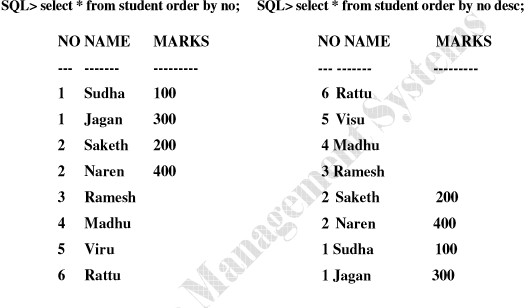
This will be used to ordering the columns data (ascending or descending). Syntax1: (simple form)

**select \* from <table\_name> order by <col> desc;**

Note: By default oracle will use ascending order.

If you want output in descending order you have to use desc keyword after the column. Ex:

SQL> select \* from student order by no; SQL> select \* from student order by no desc;



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

**Syntax2 : ( complex form)**

**SELECT *expr* FROM *table***

**[WHERE *condition(s)*]**

**[ORDER BY {*column*, *expr*} [ASC|DESC]];**

In the syntax,

**ORDER BY :**specifies the order in which the retrieved rows are displayed. orders the rows in ascending order ( default order)

orders the rows in descending order

**Ordering of Data :-**

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

**Examples :-**

Arrange employee records in ascending order of their sal ?

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

Arrange employee records in descending order of their sal ?

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

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

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

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

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

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

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

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

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

**SQL>SELECT \* FROM emp ORDER BY comm DESC NULLS LAST ; VIII.GROUP BY AND HAVING CLAUSE**

**GROUP BY clause**

**Using group by, we can create groups of related information. Columns used in select must be used with group by, otherwise it was not a group by expression.**

**SELECT [DISTINCT] select-list FROM from-list**

**WHERE qualification GROUP BY grouping-list**

**HAVING group-qualification**

* The select list in the SELECT clause contain
  1. A list of column names
  2. A list of terms having the form aggop( aggregate operators) Every column that appear in (1) must also appear in **grouping-list**
* The expression appearing in the **group-qualification** in the HAVING clause must have a single value per group.

**Ex: SQL> select deptno, sum(sal) from emp group by deptno; SQL> select deptno, sum(sal) from emp group by deptno; DEPTNO SUM(SAL)**

**---------- ----------**

|  |  |
| --- | --- |
| **10** | **8750** |
| **20** | **10875** |
| **30** | **9400** |

**SQL> select deptno,job,sum(sal) from emp group by deptno,job; Sql>** Find the age of the youngest sailor for each rating level.

SQL> **Select s.rating, MIN(s.age) from sailors s GROUP BY s.rating;**

Find the age of the youngest sailor who is eligible to vote for each rating level with at least two such sailors ?

SQL> **select s.rating, MIN(s.age) as minage from sailors s where s.age>=18**

**GROUP BY s.rating HAVING COUNT(\*) > 1;**

For each red boat find the number of reservations for this boat?

SQL> **Select b.bid, COUNT(\*) AS reservationcount from boats b, reserves r where r.bid=b.bid and b.color=’red’**

**GROUP BY b.bid;**

Find the average age of sailors for each rating level that has at least two sailors ?

SQL> **Select s.rating, AVG(s.age) AS avgage from sailors s**

**GROUP BY s.rating**

**HAVING COUNT(\*) > 1;**

**IX. AGGREGATION**

It is a group operation, which will be works on all records of a table. To do this, Group functions required to process group of rows and Returns one value from that group.

These functions are also called **AGGREGATE** functions or **GROUP** functions

* **Aggregate functions - max(),min(),sum(),avg(),count(),count(\*).**

Group functions will be applied on all the rows but produces single output.

* 1. SUM

This will give the sum of the values of the specified column. Syntax: sum (column)

Ex: SQL> select sum(sal) from emp;

* 1. AVG

This will give the average of the values of the specified column. Syntax: avg (column)

Ex: SQL> select avg(sal) from emp;

* 1. MAX

This will give the maximum of the values of the specified column. Syntax: max (column)

Ex: SQL> select max(sal) from emp;

* 1. MIN

This will give the minimum of the values of the specified column. Syntax: min (column)

Ex: SQL> select min(sal) from emp;

* 1. COUNT

This will give the count of the values of the specified column. Syntax: count (column)

Ex: SQL> select count(sal),count(\*) from emp;

**X. SUB QUERIES**

What is subquery in SQL?

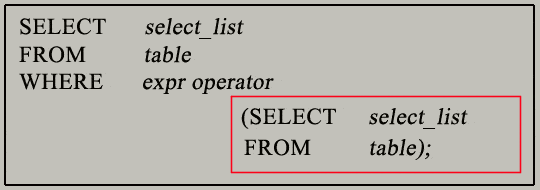
A subquery is a SQL query nested inside a larger query.

* A subquery may occur in :
  + A SELECT clause
  + A FROM clause
  + A WHERE clause
* The subquery can be nested inside a SELECT, INSERT, UPDATE, or DELETE statement or inside another subquery.
* A subquery is usually added within the WHERE Clause of another SQL SELECT statement.
* You can use the comparison operators, such as >, <, or =. The comparison operator can also be a multiple-row operator, such as IN, ANY, or ALL.
* A subquery is also called an inner query or inner select, while the statement containing a subquery is also called an outer query or outer select.
* The inner query executes first before its parent query so that the results of inner query can be passed to the outer query.

You can use a subquery in a SELECT, INSERT, DELETE, or UPDATE statement to perform the following tasks :

* Compare an expression to the result of the query.
* Determine if an expression is included in the results of the query.
* Check whether the query selects any rows.

**Syntax :**

****

* The subquery (inner query) executes once before the main query (outer query) executes.
* The main query (outer query) use the subquery result.

**SQL Subqueries Example :**

In this section, you will learn the requirements of using subqueries. We have the following two tables **'student'** and **'marks'** with common field 'SID'.

**SQL> select \*from student1**; SID NAME

v001 abhi v002 abhay v003 arjun v004 anand

**SQL> select \*from marks;**

SID TOTALMARKS

|  |  |
| --- | --- |
| v001 | 95 |
| v002 | 80 |
| v003 | 74 |
| v004 | 81 |

Now we want to write a query **to identify all students who get better marks than that of the student who's StudentID is 'V002',** but we do not know the marks of **'V002'**.

- To solve this problem, we require **two** queries.

One query returns the marks (stored in Totalmarks field) of 'V002' and a second query identifies the students who get better marks than the result of the first query.

**SQL> select \*from marks where sid='v002'; Query Result:**

**SID TOTALMARKS**

**---------- ---------- v002** **80**

The result of query is **80**.

- Using the result of this query, here we have written another query to identify the students who get better marks than 80. Here is the query :

**Second query :**

**SQL> select s.sid,s.name,m.totalmarks from student1 s, marks m where s.sid=m.sid and m.totalmarks>80;**

**SID NAME TOTALMARKS**

**---- ---------- ----------**

**v001 abhi** **95**

**v004 anand** **81**

Above two queries identified students who get better number than the student who's StudentID is 'V002' (Abhi).

You can combine the above two queries by placing one query inside the other. The subquery (also called the 'inner query') is the query inside the parentheses. See the following code and query result :

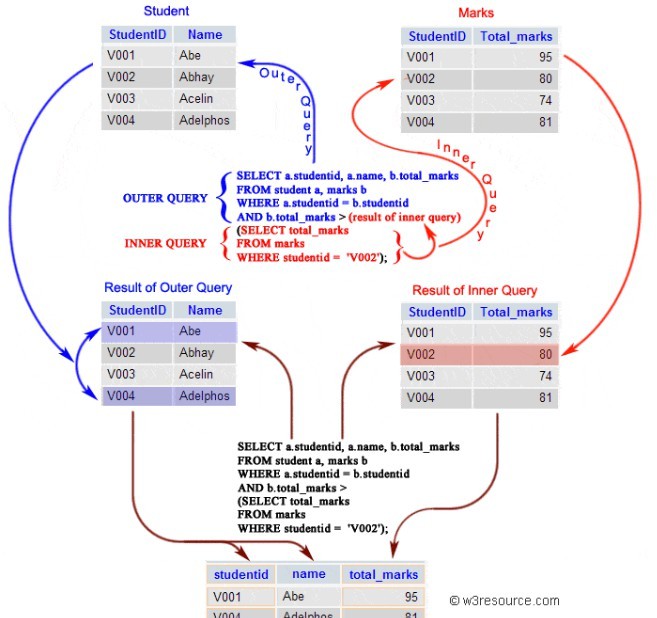
**SQL> select s.sid,s.name,m.totalmarks from student1 s,marks m where s.sid=m.sid and m.totalmarks >(select totalmarks from marks where sid='v002');**

**SID NAME TOTALMARKS**

**---- ---------- ----------**

**v001 abhi** **95**

**v004 anand** **81**

****

**Subqueries: Guidelines**

There are some guidelines to consider when using subqueries :

-A subquery must be enclosed in parentheses.

-A subquery must be placed on the right side of the comparison operator.

-Subqueries cannot manipulate their results internally, therefore ORDER BY clause cannot be added in to a subquery.You can use a ORDER BY clause in the main SELECT statement (outer query) which will be last clause.

-Use single-row operators with single-row subqueries.

-If a subquery (inner query) returns a null value to the outer query, the outer query will not return any rows when using certain comparison operators in a WHERE clause.

**Type of Subqueries**

* **Single row subquery** : Returns zero or one row.
* **Multiple row subquery** : Returns one or more rows.
* **Multiple column subquery** : Returns one or more columns.
* **Correlated subqueries** : Reference one or more columns in the outer SQL statement. The subquery is known as a correlated subquery because the subquery is related to the outer SQL statement.
* **Nested subqueries** : Subqueries are placed within another subqueries.

1. **SINGLE ROW SUBQUERIES:-** Returns zero or one row.

If inner query returns only one row then it is called single row subquery.

**Syntax :-**

**SELECT <collist> FROM <tabname>**

**WHERE colname OPERATOR (SELECT statement) Operator** can be < > <= >= = <>

**Examples:- (on Emp Table)**

Q: Display employee records whose job equals to job of SMITH?

**SQL>SELECT \* FROM emp**

**WHERE job = (SELECT job FROM emp WHERE ename=’SMITH’) ;**

Q: Display employee name earning maximum salary ?

**SQL>SELECT ename FROM emp**

**WHERE sal = (SELECT MAX(sal) FROM emp) ;**

Example2: (on SAILORS \_BOAT\_RESERVATION DATABASE )

**SQL> SELECT \* FROM SAILORS;**

**SID SNAME RATING AGE**

**---------- ---------- ---------- ----------**

|  |  |  |  |
| --- | --- | --- | --- |
| **22** | **DUSTIN** | **7** | **45** |
| **29** | **BRUTUS** | **1** | **33** |
| **31** | **LUBBER** | **8** | **55.5** |
| **32** | **CANDY** | **8** | **25.5** |
| **58** | **RUSTY** | **10** | **35** |
| **64** | **HORATIO** | **7** | **35** |
| **71** | **ZOBRA** | **10** | **16** |
| **74** | **HORATIO** | **9** | **35** |
| **85** | **ART** | **3** | **25.5** |
| **95** | **BOB** | **3** | **63.5** |

**SQL> SELECT \* FROM BOATS; BID BNAME COLOR**

**---------- ---------- ----------**

* 1. **INTERLAKE BLUE**
  2. **INTERLAKE RED**
  3. **CLIPPER GREEN**
  4. **MARINE RED**

**SQL> SELECT \* FROM RESERVES; SID BID DAY**

**---------- ---------- ---------**

|  |  |  |
| --- | --- | --- |
| **22** | **101** | **10-OCT-17** |
| **22** | **102** | **10-OCT-17** |
| **22** | **103** | **10-OCT-17** |
| **22** | **104** | **10-JUL-17** |
| **31** | **102** | **11-OCT-17** |
| **31** | **103** | **11-JUN-17** |
| **31** | **104** | **11-DEC-17** |
| **64** | **101** | **09-MAY-17** |
| **64** | **102** | **09-AUG-17** |
| **74** | **103** | **09-AUG-17** |

**Q: Find the sailor’s ID whose name is equal to ‘DUSTIN’**

**SQL> SELECT SID FROM SAILORS WHERE SID = (SELECT SID FROM SAILORS WHERE SNAME='DUSTIN');**

**SID**

**----------**

**22**

**Q:Find sailors records whose name equals to ‘ DUSTIN’?**

**SQL> SELECT \*FROM SAILORS WHERE SID = (SELECT SID FROM SAILORS WHERE SNAME='DUSTIN');**

**SID SNAME RATING AGE**

**----------**

**----------**

**----------**

**----------**

**22 DUSTIN 7** **45**

**Q:Find the rating of a sailor whose name is ‘DUSTIN’.**

SQL> SELECT RATING FROM SAILORS WHERE SID = (SELECT SID FROM SAILORS WHERE SNAME='DUSTIN');

RATING

7

**Q: Find the sailors records whose sid is geater than ‘dustin’?**

**SQL> SELECT \*FROM SAILORS WHERE SID > (SELECT SID FROM SAILORS WHERE SNAME='DUSTIN');**

**SID SNAME RATING AGE**

**---------- ---------- ---------- ----------**

|  |  |  |
| --- | --- | --- |
| **29 BRUTUS** | **1** | **33** |
| **31 LUBBER** | **8** | **55.5** |
| **32 CANDY** | **8** | **25.5** |
| **58 RUSTY** | **10** | **35** |
| **64 HORATIO** | **7** | **35** |
| **71 ZOBRA** | **10** | **16** |
| **74 HORATIO** | **9** | **35** |
| **85 ART** | **3** | **25.5** |
| **95 BOB** | **3** | **63.5** |

**Q:Find the sailors records ,whose sailors’ having maximum rating .**

**SQL> SELECT \*FROM SAILORS WHERE RATING = (SELECT MAX(RATING) FROM SAILORS);**

**SID SNAME RATING AGE**

**---------- ---------- ---------- ----------**

|  |  |  |
| --- | --- | --- |
| **58 RUSTY** | **10** | **35** |
| **71 ZOBRA** | **10** | **16** |

**Q:Find the records of sailors whose rating is same as ‘DUSTIN’**

**SQL> SELECT \*FROM SAILORS WHERE RATING = (SELECT RATING FROM SAILORS WHERE SNAME='DUSTIN');**

**SID SNAME RATING AGE**

**---------- ---------- ---------- ----------**

**22 DUSTIN 7** **45**

**64 HORATIO 7** **35**

**Q:Find the records of sailors whose rating is higher than ‘DUSTIN’**

**SQL> SELECT \*FROM SAILORS WHERE RATING > (SELECT MAX(RATING) FROM SAILORS WHERE SNAME='DUSTIN');**

**SID SNAME RATING AGE**

**---------- ---------- ---------- ----------**

|  |  |  |
| --- | --- | --- |
| **31 LUBBER** | **8** | **55.5** |
| **32 CANDY** | **8** | **25.5** |
| **58 RUSTY** | **10** | **35** |
| **71 ZOBRA** | **10** | **16** |
| **74 HORATIO**  **MULTI ROW SUBQUERIES :** | **9** | **35** |

if inner query returns more than one row then it is called multi row subquery.

**Syntax :-**

**SQL>SELECT <collist> FROM <tabname>**

**WHERE colname OPERATOR (SELECT statement) ;**

Here, OPERATOR must be IN , NOT IN, ANY, ALL

**IN operator :-**

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

**Syntax:-**

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

**Note :-**

IN ( ... ) is actually translated by Oracle server to a set of ‘OR’ conditions: a =value1 OR a = value2 OR a = value3. So using IN ( ... ) has no performance benefits, and it is used for logical simplicity.

**Example :-**

Q:Display employee records working as CLERK OR MANAGER ?

**SQL>SELECT \* FROM emp WHERE job IN (‘CLERK’,’MANAGER’) ;**

Q:Find the name of sailors who have reserved boat 103

**SQL> SELECT S.SNAME FROM SAILORS S WHERE S.SID IN (SELECT R.SID FROM RESERVES R WHERE R.BID=103);**

**SNAME**

**---------- DUSTIN LUBBER HORATIO**

Q:Find the name of sailors who have reserved a red boat

**SQL> SELECT S.SNAME FROM SAILORS S WHERE S.SID IN (SELECT R.SID FROM RESERVES R WHERE R.BID IN (SELECT B.BID FROM BOATS B WHERE B.COLOR='RED'));**

**SNAME**

**---------- DUSTIN LUBBER HORATIO**

Q:Find the names of sailors who have not reserved a red boat.

**SELECT S.SNAME FROM SAILORS S WHERE S.SID NOT IN (SELECT R.SID FROM RESERVES R WHERE R.BID IN (SELECT B.BID FROM BOATS B WHERE B.COLOR**

**= 'RED'));**

**SNAME**

**---------- BRUTUS CANDY RUSTY ZOBRA HORATIO ART**

**BOB**

**Using EXISTS operator :-**

EXISTS operator returns TRUE or FALSE.

If inner query returns at least one record then EXISTS returns TRUE otherwise returns FALSE.

ORACLE recommends EXISTS and NOT EXISTS operators instead of IN and NOT IN. Q: Find the name of sailors who have reserved boat 103

**SQL> SELECT S.SNAME FROM SAILORS S WHERE EXISTS (SELECT \* FROM RESERVES R WHERE R.BID=103 AND R.SID = S.SID) ;**

**SNAME**

**---------- DUSTIN LUBBER HORATIO**

Q:Find the name of sailors who have not reserved boat 103

**SQL> SELECT S.SNAME FROM SAILORS S WHERE NOT EXISTS (SELECT \* FROM RESERVES R WHERE R.BID=103 AND R.SID = S.SID) ;**

**SNAME**

**---------- BRUTUS CANDY RUSTY HORATIO ZOBRA ART**

**BOB**

**ANY operator:-**

Compares a value to each value in a list or returned by a query. Must be preceded by =, !=, >, <,

<=, >=. Evaluates to FALSE if the query returns no rows.

Select employees whose salary is greater than any salesman’s salary ?

**SQL>SELECT ename FROM emp**

**WHERE SAL > ANY ( SELECT sal FROM emp WHERE job = 'SALESMAN');**

Q:Find sailors whose rating is better than some sailor called Horatio?

**SQL> SELECT S.SID FROM SAILORS S WHERE S.RATING > ANY ( SELECT S2.RATING FROM SAILORS S2 WHERE S2.SNAME=’HORATIO’) ;**

**SID**

**----------**

**58**

**71**

**74**

**31**

**32**

**ALL operator :-**

Compares a value to every value in a list or returned by a query. Must be preceded by =, !=, >,

<, <=, >=. evaluates to TRUE if the query returns no rows.

**Example:-**

Select employees whose salary is greater than every salesman’s salary ?

**SQL>SELECT ename FROM emp**

**WHERE SAL > ALL ( SELECT sal FROM emp WHERE job = 'SALESMAN');**

Q:Find sailors whose rating is better than every sailor called Horation?

**SQL> SELECT S.SID FROM SAILORS S WHERE S.RATING > ALL ( SELECT S2.RATING FROM SAILORS S2 WHERE S2.SNAME=’HORATIO’) ;**

**SID**

**----------**

**58**

**71**

**Multi Column Subqueries:-**

If inner query returns more than one column value then it is called MULTI COLUMN subquery.

**Example :-**

Display employee names earning maximum salaries in their dept ?

**SQL>SELECT ename FROM emp**

**WHERE (deptno,sal) IN (SELECT deptno,MAX(sal)**

**FROM emp**

**GROUP BY deptno) ;**

**SQL> SELECT SNAME FROM SAILORS WHERE (RATING,AGE) IN (SELECT RATING,MAX(AGE) FROM SAILORS GROUP BY RATING);**

**SNAME**

**---------- DUSTIN BRUTUS LUBBER RUSTY HORATIO BOB**

**SQL> SELECT SID,SNAME FROM SAILORS WHERE (RATING,AGE) IN (SELECT RATING,MAX(AGE) FROM SAILORS GROUP BY RATING);**

**SID SNAME**

**---------- ----------**

**22 DUSTIN**

**29 BRUTUS**

**31 LUBBER**

**58 RUSTY**

**74 HORATIO**

**95 BOB**

**Nested Queries:-**

A subquery embedded in another subquery is called NESTED QUERY.

Queries can be nested upto 255 level.

**Example :-**

Display employee name earning second maximum salary ?

**SQL>SELECT ename FROM emp**

**WHERE sal = (SELECT MAX(sal) FROM EMP**

**WHERE sal < (SELECT MAX(sal) FROM emp)) ;**

Q:Find the names of sailors who have not reserved a red boat.

**SELECT S.SNAME FROM SAILORS S WHERE S.SID NOT IN (SELECT R.SID FROM RESERVES R WHERE R.BID IN (SELECT B.BID FROM BOATS B WHERE B.COLOR**

**= 'RED'));**

**SNAME**

**---------- BRUTUS CANDY RUSTY ZOBRA HORATIO ART**

**BOB**

**CORRELATED SUB QUERIES:**

In the Co-Related sub query a parent query will be executed first and based on the output of outer query the inner query execute.

If parent query returns N rows ,inner query executed for N times.

If a subquery references one or more columns of parent query is called CO-RELATED subquery because it is related to outer query. This subquery executes once for each and every row of main query.

**Example1 :-**

Display employee names earning more than avg(sal) of their dept ?

**SQL>SELECT ename FROM emp x**

**WHERE sal > (SELECT AVG(sal) FROM emp**

**WHERE deptno=x.deptno);**

**Example2**: Find sailors whose rating more than avg(rating ) of their id.

SQL> SELECT S.SNAME FROM SAILORS S WHERE RATING > (SELECT AVG(RATING) FROM SAILORS WHERE SID=S.SID);

no rows selected.

**SUB QUERIES WITH SET OPERATORS:**

Q1) Find the names of sailors who have reserved a red or a green boat?

SQL> **Select s.sname from sailors s, reserves r, boats b where s.sid=r.sid and r.bid=b.bid and (b.color = ‘red’ or b.color= ‘green’);**

**Or**

**SQL> Select s.sname from sailors s, reserves r, boats b where s.sid=r.sid and r.bid=b.bid and b.color=’red’**

# UNION

**Select s.sname from sailors s, reserves r, boats b where s.sid=r.sid and r.bid=b.bid and b.color=’green’;**

|  |
| --- |
| SNAME |
| Dustin |
| Lubber |
| Horatio |

Q2) Find the names of sailors who have reserved a red and a green boat?

**SQL> Select s.sname from sailors s, reserves r, boats b where s.sid=r.sid and r.bid=b.bid and b.color=’red’**

# INTERSECT

**Select s.sname from sailors s, reserves r, boats b where s.sid=r.sid and r.bid=b.bid and b.color=’green’;**

|  |
| --- |
| SNAME |
| Dustin |
| Lubber |

Q3) Find the names of sailors who have reserved a red boat but not green boat?

Horatio

**SQL> Select s.sname from sailors s, reserves r, boats b where s.sid=r.sid and r.bid=b.bid and b.color=’red’**

# MINUS

**Select s.sname from sailors s, reserves r, boats b where s.sid=r.sid and r.bid=b.bid and b.color=’green’;**

# NO ROWS SELECTED

Q4) Find all sids of sailors who have a rating of 10 or reserved boat 104?

SQL>**select s.sid from sailors s where s.rating=10 UNION**

**Select r.sid from reserves r where r.bid=104;**

|  |
| --- |
| SID |
| 22 |
| 31 |
| 58 |
| 71 |