Grouping using GROUP BY and HAVING:

GROUP BY Clause:-

The **GROUP BY** clause is another section of the **select** statement. This optional clause tells Oracle to group rows based on distinct values that exist for **specified columns.** The GROUP BY clause creates a data set, containing several sets of records **grouped together** based on a condition.

Syntax:-	**Example**:-
SELECT <ColumnName1>**,** <ColumnName2>**,** <ColumnNameN>**, AGGREGATE_FUNCTION (**<Expression>**)** **FROM TableName WHERE** <condition> **GROUP BY** <ColumnName1>, <ColumnName2>, <ColumnNameN>;	1) Select dno,sum**(sal)** from emp **group by dno**; 2) Select eno,dno,ename,**sum(sal)** from emp where dno=22 **group by dno,eno,ename**; output: 1). dno sal 11 8000 22 5800

HAVING Clause:-

The **HAVING** clause can be used in conjunction with the **GROUP BY** clause, **HAVING** imposes a condition on the **GROUP BY** clause, which further filters the groups created by the **GROUP BY** clause. Each column specification specified in the HAVING clause must occur within a statistical function or must occur in the list of columns named in the **GROUP BY** clause.

Syntax:-	**Example**:-
SELECT <ColumnName1>**,** <ColumnName2>**,** <ColumnNameN>**, AGGREGATE_FUNCTION (**<Expression>**)** **FROM TableName WHERE** <condition> **GROUP BY** <ColumnName1>, <ColumnName2>, <ColumnNameN> **HAVING AGGREGATE_FUNCTION (**<Expression>**) <operator>** <value>;	select cust_no,count(fd_no) from emp **group by cust_no** **having count(fd_no) > 2;**

Rules for Group By and having Clause:-

- Columns listed in the select statement have to be listed in the GROUP BY clause
- Columns listed in the GROUP BY clause need not be listed in the SELECT statement.
- Only group functions can be used in the HAVING clause.
- The group functions listed in the having clause need not be listed in the SELECT statement.

Sub queries

- A **subquery** is a form of an SQL statement that appears inside another SQL statement.
- It is also termed as **nested query**.
- The statement containing a subquery is called a **parent** statement.
- The parent statement users the rows (i.e. the result set) returned by the subquery.

Syntax:-	**Example:-**
Select <ColumnName1>,<ColumnName2>,.. **from** <tablename> where <expr> <operator> **(select** <columname1>,<columnname2> from <tablanme> where <condition> group by <ColumnName> having <condition> Order by <columName>);	1. Select eno,ename,sal from emp where **sal > (select sal from emp where name='Jayesh';**

- A subquery must be enclosed with simple brackets or peranthesis.
- Place the subquery on the right side of the comparison condition.
- One common error with subquery is more than one row returned with single row subquery. At that time use the IN, ANY or ALL operator.

It can be used for the following:

- To insert records in a target table
- To create tables and insert records in the table created
- To update records in a target table
- To create views
- To provide values for conditions in WHERE, HAVING, IN and so on used with SELECT, UPDATE, and DELETE statements.

Two Types of Subquery:-

1) **Single Row Subquery (=, >, <, >=, <=, <>)**
2) **Multiple Row Subquery (IN, ANY, ALL)**

ANY:-

The ANY operator compares value to each value returned by the subquery. ANY is equivalent to IN.

< ANY :- Less than the maximum.
> ANY :- Greater than the minimum.

ALL:-

The ALL operator compares a value to every value returned by a subquery.

>ALL :- Greater than the maximum.
<ALL :- Less than the minimum.

One of the value return by the inner query is NULL value and hence the entire query returns no rows. The reason is that all condition compare NULL value result in a NULL. So whenever NULL values are likely to be part of the result set of a subquery **do not use NOT IN operator.**

So use the **<> operator.**

Joins

Joining Multiple Tables(Equi Joins):-

Sometimes it is necessary to work with multiple tables as through they were a single entity. Then a single SQL sentence can manipulate data from all the tables. **Joins** are used to achieve this. Tables are joined on columns that have the same **data type** and **data width** in the tables.

Tables in a database can be related to each other with keys. A primary key is a column with a unique value for each row. The purpose is to bind data together, across tables, without repeating all of data in every table.

The JOIN operator specifies how to relate tables in the query.

Types of JOIN:

- INNER
- OUTER(LEFT,RIGHT,FULL)
- CROSS

➢ **INNER JOIN:**

Inner joins are also known as Equi Joins. There are the most common joins used in SQL*Plus. They are know as Equi joins because the where statement generally compares two columns from two tables with the equivalence operator=. This type of join is by far the most commonly used. In fact, many systems use this type as the default join. This type of join can be used in situations where selecting only those rows that have values in common in the columns specified in the ON clause, is required. In short, the INNER JOIN returns all rows both table where a match is.

Syntax:-	**Example:**
ANSI-style **SELECT**<ColumnNm1>, <ColumnNm2>, <ColumnNmN> **FROM** <TableName1> **INNER JOIN** <TableName2> O**N** **<**TableName1**>. <**ColumnNm1**>=<**TableName2**>.<**ColumnName2**>** **WHERE <**Condition**>** **ORDER BY <**ColumnName1**>, <**ColumnName2**>;**	**ANSI-style** **SELECT eno,e.dno,ename,dname** **from emp e inner join dept d** **on e.dno=d.dno;**
Theta style:- SELECT <ColumnName1>, <ColumnName2>, <ColumnNameN> FROM <TableName1>, <TableName2> **WHERE** **<TableName1>.<ColumnName1>=<TableName2>.<ColumnName2>** **AND** <condition> **ORDER BY** <ColumnName1>, <ColumnName2>;	**Theta style:-** **SELECT eno,e.dno,ename,dname** **from emp e,dept d** **where e.dno=d.dno;** **and eno=111;**

In the above syntax :

- ColumnName1 in TableName1 is usually that table's Primary Key.
- ColumnName2 in TableName2 is a Foreign Key in that table.
- ColumnName1 in ColumnName2 must have the same Data Type and for certain data types, the same size.

- **OUTER JOIN:**

Outer joins are similar to inner joins, but give a bit more flexibility when selecting data from related tables. This type of join can be used in situations where it is desired, to select all rows from the tables on the left (or right, or both) regardless of whether the other table has values in common and (usually) enter NULL where data is missing.

Three types of Outer Join :

1) **Left Outer join**
2) **Right Outer Join**
3) **Full Outer Join**

- **Left Outer Join:-**

This type of join can be used in situations where it is desired, to select all rows from the tables on the left where there is match.

Syntax:- (Left Outer Join)	**Example:-**
ANSI-style **SELECT**<ColumnNm1>, <ColumnNm2>, <ColumnNmN> **FROM** <TableName1> **LEFT JOIN** <TableName2> O**N** **<**TableName1**>. <**ColumnNm1**>=<**TableName2**>.<**ColumnName2**>** **WHERE <**Condition**>** **ORDER BY <**ColumnName1**>, <**ColumnName2**>;** **Theta style:-** SELECT <ColumnName1>, <ColumnName2>, <ColumnNameN> FROM <TableName1>, <TableName2> **WHERE <TableName1>.<ColumnName1>=<TableName2>.<ColumnName2>(+)** **AND** <condition> **ORDER BY** <ColumnName1>, <ColumnName2>;	**ANSI-style** **SELECT eno,e.dno,ename,dname** **from emp e left join dept d** **on e.dno=d.dno;** **Theta style:-** **SELECT eno,e.dno,ename,dname** **from emp e,dept d** **where e.dno=d.dno(+);** **(i.e) Display all the data of emp table , if dno of both table is match.**

- **Right Outer Join:-**

This type of join can be used in situations where it is desired, to select all rows from the tables on the Right where there is match.

Syntax:- (Left Outer Join)	**Example:-**
ANSI-style **SELECT**<ColumnNm1>, <ColumnNm2>, <ColumnNmN> **FROM** <TableName1> **RIGHT JOIN** <TableName2> O**N** **<**TableName1**>. <**ColumnNm1**>=<**TableName2**>.<**ColumnName2**>** **WHERE <**Condition**>** **ORDER BY <**ColumnName1**>, <**ColumnName2**>;** **Theta style:-** SELECT <ColumnName1>, <ColumnName2>, <ColumnNameN> FROM <TableName1>, <TableName2> **WHERE <TableName1>.<ColumnName1>(+)=<TableName2>.<ColumnName2>** **AND** <condition> **ORDER BY** <ColumnName1>, <ColumnName2>;	**ANSI-style** **SELECT eno,e.dno,ename,dname from emp e right join dept d on e.dno=d.dno;** **Theta style:-** **SELECT eno,e.dno,ename,dname from emp e,dept d where e.dno(+)=d.dno;** **(i.e) Display all the data of dept table , if dno of both table is match.**

- **Full Outer Join:-**

Syntax:- (Full Outer Join)	**Example:-**
ANSI-style **SELECT**<ColumnNm1>, <ColumnNm2>, <ColumnNmN> **FROM** <TableName1> **FULL JOIN** <TableName2> O**N** **<**TableName1**>. <**ColumnNm1**>=<**TableName2**>.<**ColumnName2**>** **WHERE <**Condition**>** **ORDER BY <**ColumnName1**>, <**ColumnName2**>;**	**ANSI-style** **SELECT eno,e.dno,ename,dname from emp e Full join dept d on e.dno=d.dno;** **(i.e) Display all the data of emp table and emp , if dno of both table is match.**

- **CROSS JOIN:**

A cross joins returns what's known as a Cartesian product. This means that the join combines every row from the left table with every row in the table. As can be imagined, sometimes this join produces a mess, but under the right circumstances, it can be very useful. This type of join can be used in situations where it is desired, to select all possible combinations of rows and columns from both tables. This kind of join is usually not preferred as it may run for a very long time and produce a huge result set that may not be useful.

Syntax:-	**Example:-**
ANSI-style **SELECT**<ColumnNm1>, <ColumnNm2>, <ColumnNmN> **FROM** <TableName1> **CROSS JOIN** <TableName2> O**N** **<**TableName1**>. <**ColumnNm1**>=<**TableName2**>.<**ColumnName2**>** **WHERE <**Condition**>** **ORDER BY <**ColumnName1**>, <**ColumnName2**>;**	**ANSI-style** **SELECT eno,e.dno,ename,dname from emp e cross join dept d on e.dno=d.dno;**

Indexes:-

- When a select statement is fired to search for a particular record, the Oracle engine must first locate the table on the hard disk. The oracle engine reads system information and finds the start location of a table's record on the current storage media. The oracle engine then performs a sequential search to locate records that matches user-defined criteria as specified in the **select.**
- Indexing involves forming a two dimensional matrix completely independent of the table on which the index is being created. This two dimensional matrix will have a **single column**, which will hold sorted data, extracted from the table column(s) on which the index is created.
- Another column is called **the address field** identifies the location of the record in the oracle database.
- For every data value held in the index the oracle engine inserts a unique ROWID value. This is done for every data value inserted into the index, without exception.

- Users and application developers can also use ROWIDs for the following functions:

1. Rowids are the fastest means of accessing particular rows.
2. Rowids can be used to see how a table is organized.
3. Rowids are unique identifiers for rows in a given table.

- The ROWID format used by oracle is as follows:

  BBBBBBB.RRRR.FFFF

- Where, **FFFF** is a unique number given by the oracle engine to **each Data file**. Data files are the files used by the oracle engine to store user data.

- Each data file is further divided into Data Blocks and each block is given a unique number. Thus block number can be used to identify the data block in which a record is stored. **BBBBBBB** is the **Block number** in which the record is stored.
- Each data block can store one or more records. Thus each record in the data block is given a unique **Record number**. **RRRR** is a unique record number.

Duplicate / Unique Index

Oracle allows the creation of two types of indexes.

1. Index that allows duplicate values for the indexed columns i.e. **Duplicate Index**
2. Index that deny duplicate values for the indexed columns i.e. **Unique Index**

Creation of an Index:-

An index can be created on one more columns. Based on the number columns included in the index, an index can be:

- **Simple index**
- **Composite index**

Creating a simple index

- An index created on a single column of a table is called a simple index.

Syntax:	**Example:-**
Create index <IndexName> ON <TableName> (<ColumnName>);	Create indx_emp on emp (eno);

- **Composite index**

- An index created on more than one column is called a composite index.

Syntax:	**Example:-**
Create index <IndexName> ON <TableName> (<ColumnName1>,<ColumnName2>);	Create indx_emp on emp (eno,dno);

Creating of unique index

A unique index can also be created on one or more columns. If an index is created on a single column it is called simple index.

Syntax:	**Example:-**
Create UNIQUE index <IndexName> ON <TableName> (<ColumnName>);	Create unique indx_emp on emp (eno);

Creating of composite unique index

If an index is created on more than one column it is called composite index.

Syntax:	**Example:-**
Create UNIQUE index <IndexName> ON <TableName> (<ColumnName1,<ColumnName2>);	Create unique indx_emp on emp (eno,dno);

Reverse key indexes

Creating a reverse key index when compare to a simple index, reverse each byte of the column being indexed while keeping the column order.

Syntax:	**Example:-**
Create index <IndexName> ON <TableName> (<ColumnName>) Reverse;	Create indx_emp on emp (eno) reverse;

A reverse key index can be rebuilt into a normal index using the keywords Rebuild Noreverse.

Syntax: **Alter index <IndexName> REBUILD NOREVERSE;**	**Example:-** Create index indx_emp on emp (eno) rebuild noreverse;

Alter index <IndexName> REBUILD NOREVERSE

Views:-

After a table is created and populated with data, it may become necessary to prevent all users from accessing all columns of a table, for data security reasons. This would mean creating several tables having the appropriate number of columns and assigning specific users to each table.

To reduce redundant data to the minimum possible, oracle allows the creation of an object called a **view.** A view is mapped to a select sentence. The table on which the view is based is described in the FROM clause of the select statement. The select clause consists of a sub-set of the columns of the table.

Thus a view which mapped to a table will in infect have a sub-set of the actual columns of the table from which it is built.

Creating View:-

Syntax: create view <view_name> as select <columname1>,<columnanme2> from <table_name> where <columnanme>=<expression list>	**Example:** create view view_emp as select eno,ename,sal from emp **where eno=33;**
The **ORDER BY** clause **cannot** be used while creation a view.	

- **Read Only View:-**

If a view is used to only look at table that view is called a **Read Only** view. ISer can not perform Insert, Update and Delete Operations. Read Only view is created using **WITH READ ONLY** option.

Syntax: create view <view_name> as select <columname1>,<columnanme2> from <table_name> **WITH READ ONLY**;	**Example:** create view view_emp as select eno,ename,sal from emp **WITH READ ONLY**;

❖ **Updatable View:-**

Vies can also be used for data manipulation i.e user can perform Insert, Update , Delete operations. Views on which data manipulation can be done are called **Updatable Views**.

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

From a single Table	**From Multiple Tables:** (If Table is created using referencing clause i.e using a foreign key)
▪ If the user wants to **INSERT** records with the help of as view, then the **PRIMARY KEY** column(s) and **NOT NULL** column(s) must be **included** in the view. ▪ The user can **UPDATE, DELETE** records with the help of a view even if the **PRIAMRY KEY** column and **NOT NULL** column(s) are **excluded** from the view definition.	▪ The **INSERT** operation is not allowed ▪ The **DELETE** or **MODIFY** operations **do not** affect the **Master** table. ▪ The view can be used to **MODIFY** the columns of the **child table** included in the view. ▪ If a **DELETE** operation is executed on the view, the corresponding records from the **child table** will be deleted.

Example :- Create View using single Table	**Example: Create View using Multiple Tables.**
Create view view_emp as select eno,dno,ename,sal from emp;	Create view view_emp1 as select eno,e.dno,ename,sal from emp e,dept d where e.dno-d.dno;

Restrictions on UPDATABLE VIEW:-

For the view to be updatable the view definition must not include

- Aggregate Functions
- Disinct, Group By and Having Clause
- Sub-Queries
- Union, Intersect and Union clause

Constant and value Expressions like sal*0.10

Sequences:-

- The quickest way to retrieve data from a table is to have a column in the table whose data uniquely identifies a row.
- A constraint is attached to a specific column in a table that ensure that the column is never left empty(Not Null) and data values in the columns are unique (Primary Key). Since user do data entry, and entered the duplicate value, which violates the constraints and values is rejected, so achieve this sequence is used.
- If the value entered into this column is computer generated it will always fulfill the unique constraint and the row will always be accepted for storage.
- Oracle provides an object called Sequence that can generate a numeric value
- The value generated have a maximum of 38 digits.

The Sequence can be defined to:

- Generate numbers in ascending or descending order
- Provided intervals between numbers.
- Caching of sequence numbers in memory to speed up their availability.

Creating a sequences:-

- The starting number
- The maximum number that can be generated by a sequence.
- The increment value for generating the next number.

Syntax:	**Example:-**
CREATE SEQUENCE <sequenceName> [INCREMENT BY <IntegerValue> START WITH <IntegerValue> MAXVALUE <IntegerValue>/ MAXVALUE MINVALUE <IntegerValue> / MINVALUE CYCLE / NOCYCLE CACHE <IntegerValue> / NOCHE ORDER / NOORDER]	[Ascending Order] 1. create sequence seq1 increment by 1 maxvalue 20 minvalue 1 cache 5 cycle; [Descending Oprder] 2. create sequence seq2 increment by -1 start with 10 maxvalue 10 minvalue 1 cache 20 cycle;

Keywords and Parameters:-

INCREMENT BY:- Specifies the interval between sequence numbers. It can be any positive, negative value but not a zero. Default value is 1.

MINVALUE:- Specifies the sequence minimum value.

NOMINVALUE:- Specifies a minimum value of 1 for an ascending sequence. And – 10^26 for descending sequence.

MAXVALUE:- Specifies the maximum value that a sequence can generate.

NOMAXVALUE:- Specifies a maximum 10^27 for an ascending sequence or -1 for a descending sequence. This is the default clause.

START WITH:- Specifies the first sequence number to be generated.

CYCLE:- Specifies that the sequence continues to generate repeat values after reaching either its maximum value.

NOCYCLE:- Specifies that a sequence cannot more values after reaching the maximum value.

CACHE:- Specifies hoe\w many values of a sequence Oracle pre-allocates and keeps in memory for faster access.

NOCACHE:- Specifies that the values of a sequence are mot pre-allocated.

ORDER:- This guarantees that sequence numbers are generated in the order of request.. This is only necessary if using a parallel server in parallel mode option.

NOORDER:- This does not guarantee sequence numbers are generated in order of request. This is only necessary if using a parallel server in parallel mode option. The default is NOORDER clause.

- **Referencing a Sequence: (Pseudocolumns of sequence)**

Once a sequence is created then the view the values held in cache. This using the two columns NEXTVAL and CURRVAL columns.

NEXTVAL:

Syntax:-	Example:-
select <SequenceName>.NextVal form Dual;	1) Select seq1.nextval form dual; 2) Insert into emp (eno,enm) values(seq1.nextval,'vishal');

This will Display the next value held in the cache on the screen. Every time nextval references a sequence its output is automatically incrememted from old value to new value.

CURRVAL:-

Syntax:-	Example:-
select <SequenceName>.CurrVal form Dual;	Select seq1.currval form dual;

This gives the current value of a sequence.

Altering a Sequence:-

A sequence once created can be altered. This is achieved by a ALTER SEQUENCE statement.

Syntax:	**Example:-**
ALTER SEQUENCE <sequenceName> [INCREMENT BY <IntegerValue> START WITH <IntegerValue> MAXVALUE <IntegerValue>/ MAXVALUE MINVALUE <IntegerValue> / MINVALUE CYCLE / NOCYCLE CACHE <IntegerValue> / NOCHE ORDER / NOORDER]	Alter sequence seq1 increment by 1 maxvalue 50 minvalue 1 cache 5 cycle;

Dropping a Sequence:-

To DROP SEQUENCE command is used to remove the sequence from the database.

Syntax:	**Example:-**
DROP SEQUENCE <sequenceName>;	Drop sequence seq2;

Data control language Statements – Grant And Revoke

- Objects that are created by a user are owned and controlled by the user. If a user wishes to access any of the objects belonging to another user, the owner of the object will have to give permissions for such access. This is called Granting of Privileges.
- Privileges once given can be taken back by the owner of the object. This is called Revoking of Privileges.

❖ Grant Statement :

➢ **Granting Privileges Using the GRANT Statement**

The Grant statement provides various types of access to database objects such as tables, views and sequences and so on.

Syntax:

GRANT <Object Privileges>

ON <ObjectName>

TO <UserName>

[WITH GRANT OPTION];

❖ **OBJECT PRIVILEGES**

Each object privilege that is granted authorizes the grantee to perform some operation on the object. A user can grant all the privileges or grant only specific object privileges.

The list of object privileges is as follows:

ALTER : Allows the grantee to change the table definition with the ALTER TABLE command.

DELETE : Allows the grantee to remove the records from the table with the DELETE command.

INDEX : Allows the grantee to create an index on the table with the CREATE INDEX command.

INSERT : Allows the grantee to add records to the table with the INSERT command.

SELECT : Allows the grantee to query the table with the SELECT command.

UPDATE : Allows the grantee to modify the records in the tables with the UPDATE command.

WITH GRANT OPTION

The WITH GRANT OPTION allows the grantee to in turn grant object privileges to other users.

Examples:

1> Give the user John permission to only view and modify records in the table CUST_MSTR.

SQL> GRANT SELECT, UPDATE ON CUST_MSTR TO BCA1;

➢ Revoke Statement:

REVOKING PRIVILEGES GIVEN

- Privileges once given can be denied to a user using the REVOKE command. The object owner can revoke privileges granted to another user. A user of an object who is not the owner, but has been granted the GRANT privilege, has the power to REVOKE the privileges from a grantee.

- **Revoking Permissions Using the REVOKE Statement**

The REVOKE statement is used to deny the grant given on an object.

Syntax:

Revoke <Object Privileges> ON <ObjectName> FROM <UserName>;

Note:

The revoke command is used to revoke object privileges that the user previously granted directly to the grantee. The REVOKE command cannot be used to revoke the privileges granted through the operating system.

Example

All privileges on the table NOMINEE_MSTR have been granted to harsh. Take back the Delete privilege on the table.

SQL> REVOKE DELETE ON CUST_MSTR FROM BCA1;

SQL> REVOKE SELECT ON EMP_MSTR FROM BCA2;