UNIT - 5

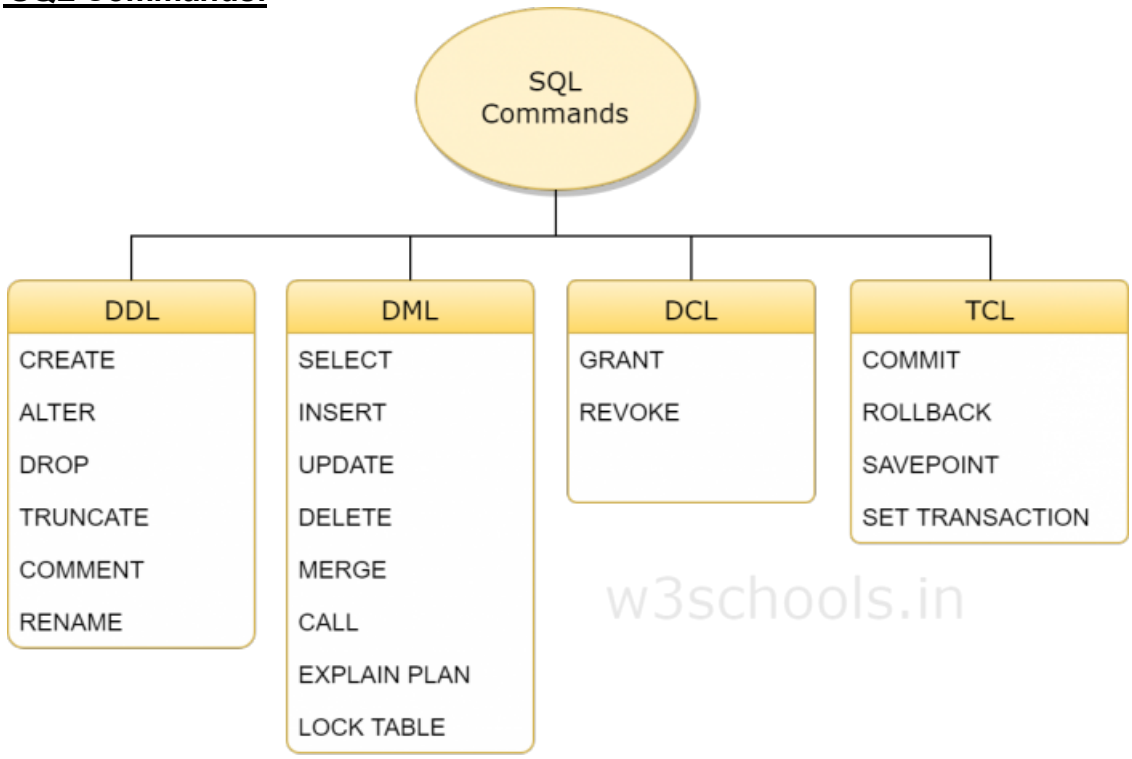
What is SQL?

- SQL stands for Structured Query Language
- SQL lets you access and manipulate databases
- SQL became a standard of the American National Standards Institute (ANSI) in 1986, and of the International Organization for Standardization (ISO) in 1987

What Can SQL do?

- SQL can execute queries against a database
- SQL can retrieve data from a database
- SQL can insert records in a database
- SQL can update records in a database
- SQL can delete records from a database
- SQL can create new databases
- SQL can create new tables in a database
- SQL can create stored procedures in a database
- SQL can create views in a database
- SQL can set permissions on tables, procedures, and views

SQL Commands:



DDL

DDL is short name of **Data Definition Language**, which deals with database schemas and descriptions, of how the data should reside in the database.

- CREATE - to create a database and its objects like (table, index, views, store procedure, function, and triggers)

The SQL CREATE DATABASE Statement

The CREATE DATABASE statement is used to create a new SQL database.

Syntax

```
CREATE DATABASE databasename;
```

CREATE DATABASE Example

The following SQL statement creates a database called "testDB":

Example

```
CREATE DATABASE testDB;
```

The SQL CREATE TABLE Statement

The CREATE TABLE statement is used to create a new table in a database.

Syntax

```
CREATE TABLE table_name (
    column1 datatype,
    column2 datatype,
    column3 datatype,
   ....
);
```

The column parameters specify the names of the columns of the table.

The datatype parameter specifies the type of data the column can hold (e.g. varchar, integer, date, etc.).

Example

```
CREATE TABLE Persons (
    PersonID int,
    LastName varchar(255),
    FirstName varchar(255),
    Address varchar(255),
    City varchar(255)
);
```

- ALTER - alters the structure of the existing database

ALTER TABLE - ADD Column

To add a column in a table, use the following syntax:

```
ALTER TABLE table_name
ADD column_name datatype;
```

The following SQL adds an "Email" column to the "Customers" table:

Example

```
ALTER TABLE Customers
ADD Email varchar(255);
```

ALTER TABLE - DROP COLUMN

To delete a column in a table, use the following syntax (notice that some database systems don't allow deleting a column):

```
ALTER TABLE table_name
DROP COLUMN column_name;
```

The following SQL deletes the "Email" column from the "Customers" table:

Example

```
ALTER TABLE Customers
DROP COLUMN Email;
```

ALTER TABLE - ALTER/MODIFY COLUMN

To change the data type of a column in a table, use the following syntax:

```
ALTER TABLE table_name
MODIFY column_name datatype;
```

Example

```
ALTER TABLE Persons
ALTER COLUMN DateOfBirth year;
```

- DROP - delete objects from the database

Syntax

```
DROP TABLE table_name;
```

Example

```
DROP TABLE Shippers;
```

- TRUNCATE - remove all records from a table, including all spaces allocated for the records are removed

Syntax

```
TRUNCATE TABLE table_name;
```

- COMMENT - add comments to the data dictionary
- RENAME - rename an object

DML

DML is short name of **Data Manipulation Language** which deals with data manipulation and includes most common SQL statements such SELECT, INSERT, UPDATE, DELETE, etc., and it is used to store, modify, retrieve, delete and update data in a database.

- SELECT - retrieve data from a database

SELECT Syntax

```
SELECT column1, column2, ...
FROM table_name;
```

Example

```
SELECT CustomerName, City FROM Customers;

SELECT * FROM Customers;
```

- INSERT - insert data into a table

INSERT INTO Syntax

It is possible to write the INSERT INTO statement in two ways.

The first way specifies both the column names and the values to be inserted:

```
INSERT INTO table_name (column1, column2, column3, ...)
VALUES (value1, value2, value3, ...);
```

OR

```
INSERT INTO table_name
VALUES (value1, value2, value3, ...);
```

Example

```
INSERT INTO Customers (CustomerName, ContactName, Address, City, PostalCode, Country)
VALUES ('Cardinal', 'Tom B. Erichsen', 'Skagen 21', 'Stavanger', '4006', 'Norway');

INSERT INTO Customers (CustomerName, City, Country)
VALUES ('Cardinal', 'Stavanger', 'Norway');
```

- UPDATE - updates existing data within a table

UPDATE Syntax

```
UPDATE table_name
SET column1 = value1, column2 = value2, ...
WHERE condition;
```

Example

```
UPDATE Customers
SET ContactName = 'Alfred Schmidt', City= 'Frankfurt'
WHERE CustomerID = 1;
```

- DELETE - Delete all records from a database table

DELETE Syntax

```
DELETE FROM table_name WHERE condition;
```

Example

```
DELETE FROM Customers WHERE CustomerName='Alfreds Futterkiste';
```

Delete All Records

It is possible to delete all rows in a table without deleting the table. This means that the table structure, attributes, and indexes will be intact:

```
DELETE FROM table_name;
```

The following SQL statement deletes all rows in the "Customers" table, without deleting the table:

Example

```
DELETE FROM Customers;
```

- MERGE - UPSERT operation (insert or update)
- CALL - call a PL/SQL or Java subprogram
- EXPLAIN PLAN - interpretation of the data access path
- LOCK TABLE - concurrency Control

DCL

DCL is short name of **Data Control Language** which includes commands such as GRANT and mostly concerned with rights, permissions and other controls of the database system.

- GRANT - allow users access privileges to the database
- REVOKE - withdraw users access privileges given by using the GRANT command

TCL

TCL is short name of Transaction Control Language which deals with a transaction within a database.

- COMMIT - commits a Transaction
- ROLLBACK - rollback a transaction in case of any error occurs
- SAVEPOINT - to rollback the transaction making points within groups
- SET TRANSACTION - specify characteristics of the transaction

The SQL WHERE Clause

The WHERE clause is used to filter records.

The WHERE clause is used to extract only those records that fulfill a specified condition.

WHERE Syntax

```
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

Example

```
SELECT * FROM Customers
WHERE Country='Mexico';
```

Operators in The WHERE Clause

The following operators can be used in the WHERE clause:

Operator	**Description**
=	Equal
>	Greater than
<	Less than
>=	Greater than or equal
<=	Less than or equal
<>	Not equal. **Note:** In some versions of SQL this operator may be written as !=
BETWEEN	Between a certain range
LIKE	Search for a pattern
IN	To specify multiple possible values for a column

SQL Constraints

SQL constraints are used to specify rules for data in a table.

SQL Create Constraints

Constraints can be specified when the table is created with the `CREATE TABLE` statement, or after the table is created with the `ALTER TABLE` statement.

Syntax

```
CREATE TABLE table_name (
    column1 datatype constraint,
    column2 datatype constraint,
    column3 datatype constraint,
    ....
);
```

SQL Constraints

SQL constraints are used to specify rules for the data in a table.

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

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

The following constraints are commonly used in SQL:

- `NOT NULL` - Ensures that a column cannot have a NULL value

SQL NOT NULL Constraint

By default, a column can hold NULL values.

The `NOT NULL` constraint enforces a column to NOT accept NULL values.

This enforces a field to always contain a value, which means that you cannot insert a new record, or update a record without adding a value to this field.

SQL NOT NULL on CREATE TABLE

The following SQL ensures that the "ID", "LastName", and "FirstName" columns will NOT accept NULL values when the "Persons" table is created:

Example

```
CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255) NOT NULL,
    Age int
);
```

SQL NOT NULL on ALTER TABLE

To create a NOT NULL constraint on the "Age" column when the "Persons" table is already created, use the following SQL:

```
ALTER TABLE Persons
MODIFY Age int NOT NULL;
```

- UNIQUE - Ensures that all values in a column are different

SQL UNIQUE Constraint

The UNIQUE constraint ensures that all values in a column are different.

Both the UNIQUE and PRIMARY KEY constraints provide a guarantee for uniqueness for a column or set of columns.

A PRIMARY KEY constraint automatically has a UNIQUE constraint.

However, you can have many UNIQUE constraints per table, but only one PRIMARY KEY constraint per table.

SQL UNIQUE Constraint on CREATE TABLE

The following SQL creates a UNIQUE constraint on the "ID" column when the "Persons" table is created:

SQL Server / Oracle / MS Access:

```
CREATE TABLE Persons (
    ID int NOT NULL UNIQUE,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int
);
```

SQL UNIQUE Constraint on ALTER TABLE

To create a UNIQUE constraint on the "ID" column when the table is already created, use the following SQL:

MySQL / SQL Server / Oracle / MS Access:

```
ALTER TABLE Persons
ADD UNIQUE (ID);
```

- PRIMARY KEY - A combination of a NOT NULL and UNIQUE. Uniquely identifies each row in a table

SQL PRIMARY KEY Constraint

The PRIMARY KEY constraint uniquely identifies each record in a table.

Primary keys must contain UNIQUE values, and cannot contain NULL values.

A table can have only ONE primary key; and in the table, this primary key can consist of single or multiple columns (fields).

SQL PRIMARY KEY on CREATE TABLE

The following SQL creates a PRIMARY KEY on the "ID" column when the "Persons" table is created:

```
CREATE TABLE Persons (
    ID int NOT NULL PRIMARY KEY,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int
);
```

To allow naming of a PRIMARY KEY constraint, and for defining a PRIMARY KEY constraint on multiple columns, use the following SQL syntax:

MySQL / SQL Server / Oracle / MS Access:

```
CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int,
    CONSTRAINT PK_Person PRIMARY KEY (ID,LastName)
);
```

SQL PRIMARY KEY on ALTER TABLE

To create a PRIMARY KEY constraint on the "ID" column when the table is already created, use the following SQL:

MySQL / SQL Server / Oracle / MS Access:

```
ALTER TABLE Persons
ADD PRIMARY KEY (ID);
```

DROP a PRIMARY KEY Constraint

To drop a PRIMARY KEY constraint, use the following SQL:

MySQL:

```
ALTER TABLE Persons
DROP PRIMARY KEY;
```

SQL Server / Oracle / MS Access:

```
ALTER TABLE Persons
DROP CONSTRAINT PK_Person;
```

- FOREIGN KEY - Prevents actions that would destroy links between tables

SQL FOREIGN KEY Constraint

The FOREIGN KEY constraint is used to prevent actions that would destroy links between tables.

A FOREIGN KEY is a field (or collection of fields) in one table, that refers to the PRIMARY KEY in another table.

The table with the foreign key is called the child table, and the table with the primary key is called the referenced or parent table.

Look at the following two tables:

Persons Table

PersonID	LastName	FirstName	Age
1	Hansen	Ola	30
2	Svendson	Tove	23
3	Pettersen	Kari	20

Orders Table

OrderID	OrderNumber	PersonID
1	77895	3
2	44678	3
3	22456	2
4	24562	1

Notice that the "PersonID" column in the "Orders" table points to the "PersonID" column in the "Persons" table.

The "PersonID" column in the "Persons" table is the `PRIMARY KEY` in the "Persons" table.

The "PersonID" column in the "Orders" table is a `FOREIGN KEY` in the "Orders" table.

The `FOREIGN KEY` constraint prevents invalid data from being inserted into the foreign key column, because it has to be one of the values contained in the parent table.

SQL FOREIGN KEY on CREATE TABLE

The following SQL creates a FOREIGN KEY on the "PersonID" column when the "Orders" table is created:

SQL Server / Oracle / MS Access:

```
CREATE TABLE Orders (
    OrderID int NOT NULL PRIMARY KEY,
    OrderNumber int NOT NULL,
    PersonID int FOREIGN KEY REFERENCES Persons(PersonID)
);
```

SQL FOREIGN KEY on ALTER TABLE

To create a FOREIGN KEY constraint on the "PersonID" column when the "Orders" table is already created, use the following SQL:

MySQL / SQL Server / Oracle / MS Access:

```
ALTER TABLE Orders
ADD FOREIGN KEY (PersonID) REFERENCES Persons(PersonID);
```

DROP a FOREIGN KEY Constraint

To drop a FOREIGN KEY constraint, use the following SQL:

SQL Server / Oracle / MS Access:

```
ALTER TABLE Orders
DROP CONSTRAINT FK_PersonOrder;
```

- CHECK - Ensures that the values in a column satisfies a specific condition

SQL CHECK Constraint

The CHECK constraint is used to limit the value range that can be placed in a column.

If you define a CHECK constraint on a column it will allow only certain values for this column.

If you define a CHECK constraint on a table it can limit the values in certain columns based on values in other columns in the row.

SQL CHECK on CREATE TABLE

The following SQL creates a CHECK constraint on the "Age" column when the "Persons" table is created. The CHECK constraint ensures that the age of a person must be 18, or older:

SQL Server / Oracle / MS Access:

```
CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int CHECK (Age>=18)
);
```

SQL CHECK on ALTER TABLE

To create a CHECK constraint on the "Age" column when the table is already created, use the following SQL:

MySQL / SQL Server / Oracle / MS Access:

```
ALTER TABLE Persons
ADD CHECK (Age>=18);
```

DROP a CHECK Constraint

To drop a CHECK constraint, use the following SQL:

SQL Server / Oracle / MS Access:

```
ALTER TABLE Persons
DROP CONSTRAINT CHK_PersonAge;
```

- DEFAULT - Sets a default value for a column if no value is specified

SQL DEFAULT Constraint

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

The default value will be added to all new records, if no other value is specified.

SQL DEFAULT on CREATE TABLE

The following SQL sets a DEFAULT value for the "City" column when the "Persons" table is created:

My SQL / SQL Server / Oracle / MS Access:

```
CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int,
    City varchar(255) DEFAULT 'Sandnes'
);
```

SQL DEFAULT on ALTER TABLE

To create a DEFAULT constraint on the "City" column when the table is already created, use the following SQL:

```
ALTER TABLE Persons
MODIFY City DEFAULT 'Sandnes';
```

- CREATE INDEX - Used to create and retrieve data from the database very quickly

SQL CREATE INDEX Statement

The CREATE INDEX statement is used to create indexes in tables.

Indexes are used to retrieve data from the database more quickly than otherwise. The users cannot see the indexes, they are just used to speed up searches/queries.

CREATE INDEX Syntax

Creates an index on a table. Duplicate values are allowed:

```
CREATE INDEX index_name
ON table_name (column1, column2, ...);
```

CREATE INDEX Example

The SQL statement below creates an index named "idx_lastname" on the "LastName" column in the "Persons" table:

```
CREATE INDEX idx_lastname
ON Persons (LastName);
```

SQL Set Operation

The SQL Set operation is used to combine the two or more SQL SELECT statements.

Types of Set Operation

1. Union
2. UnionAll
3. Intersect
4. Minus

1. Union

- The SQL Union operation is used to combine the result of two or more SQL SELECT queries.
- In the union operation, all the number of datatype and columns must be same in both the tables on which UNION operation is being applied.
- The union operation eliminates the duplicate rows from its resultset.

Syntax

```
SELECT column_name FROM table1
UNION
SELECT column_name FROM table2;
```

Example:

The First table

ID	NAME
1	Jack
2	Harry
3	Jackson

ID	NAME

3	Jackson
4	Stephan
5	David

The Second table

Union SQL query will be:

```
SELECT * FROM First
UNION
SELECT * FROM Second;
```

The resultset table will look like:

ID	NAME
1	Jack
2	Harry
3	Jackson
4	Stephan
5	David

2. Union All

Union All operation is equal to the Union operation. It returns the set without removing duplication and sorting the data.

Syntax:

```
SELECT column_name FROM table1
UNION ALL
SELECT column_name FROM table2;
```

Example: Using the above First and Second table.

Union All query will be like:

```
SELECT * FROM First
UNION ALL
SELECT * FROM Second;
```

The resultset table will look like:

ID	NAME
1	Jack
2	Harry
3	Jackson
3	Jackson
4	Stephan
5	David

3. Intersect

- It is used to combine two SELECT statements. The Intersect operation returns the common rows from both the SELECT statements.
- In the Intersect operation, the number of datatype and columns must be the same.
- It has no duplicates and it arranges the data in ascending order by default.

Syntax

```
SELECT column_name FROM table1
INTERSECT
SELECT column_name FROM table2;
```

Example:

Using the above First and Second table.

Intersect query will be:

```
SELECT * FROM First
INTERSECT
SELECT * FROM Second;
```

The resultset table will look like:

ID	NAME
3	Jackson

4. Minus

- It combines the result of two SELECT statements. Minus operator is used to display the rows which are present in the first query but absent in the second query.
- It has no duplicates and data arranged in ascending order by default.

Syntax:

```
SELECT column_name FROM table1
MINUS
SELECT column_name FROM table2;
```

Example

Using the above First and Second table.

Minus query will be:

```
SELECT * FROM First
MINUS
SELECT * FROM Second;
```

The resultset table will look like:

ID	NAME
1	Jack
2	Harry

SQL - Sub Queries

A Subquery or Inner query or a Nested query is a query within another SQL query and embedded within the WHERE clause.

A subquery is used to return data that will be used in the main query as a condition to further restrict the data to be retrieved.

Subqueries can be used with the SELECT, INSERT, UPDATE, and DELETE statements along with the operators like =, <, >, >=, <=, IN, BETWEEN, etc.

There are a few rules that subqueries must follow −

- Subqueries must be enclosed within parentheses.
- A subquery can have only one column in the SELECT clause, unless multiple columns are in the main query for the subquery to compare its selected columns.
- An ORDER BY command cannot be used in a subquery, although the main query can use an ORDER BY. The GROUP BY command can be used to perform the same function as the ORDER BY in a subquery.
- Subqueries that return more than one row can only be used with multiple value operators such as the IN operator.
- A subquery cannot be immediately enclosed in a set function.
- The BETWEEN operator cannot be used with a subquery. However, the BETWEEN operator can be used within the subquery.

Subqueries with the SELECT Statement

Subqueries are most frequently used with the SELECT statement. The basic syntax is as follows −

```
SELECT column_name [, column_name ]
FROM   table1 [, table2 ]
WHERE  column_name OPERATOR
   (SELECT column_name [, column_name ]
   FROM table1 [, table2 ]
   [WHERE])
```

Example

Consider the CUSTOMERS table having the following records −

```
+------+--------------+-------+--------------+--------------+
| ID   | NAME         | AGE   | ADDRESS      | SALARY       |
+------+--------------+-------+--------------+--------------+
|    1 | Ramesh       |    35 | Ahmedabad    |    2000.00   |
```

```
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+-----+----------+-----+-----------+----------+
```

Now, let us check the following subquery with a SELECT statement.

```
SQL> SELECT *
   FROM CUSTOMERS
   WHERE ID IN (SELECT ID
         FROM CUSTOMERS
         WHERE SALARY > 4500) ;
```

This would produce the following result.

```
+-----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS | SALARY   |
+-----+----------+-----+-----------+----------+
|  4 | Chaitali |  25 | Mumbai  |  6500.00 |
|  5 | Hardik   |  27 | Bhopal  |  8500.00 |
|  7 | Muffy    |  24 | Indore  | 10000.00 |
+-----+----------+-----+-----------+----------+
```

SQL - Group By

The SQL **GROUP BY** clause is used in collaboration with the SELECT statement to arrange identical data into groups. This GROUP BY clause follows the WHERE clause in a SELECT statement and precedes the ORDER BY clause.

Syntax

The basic syntax of a GROUP BY clause is shown in the following code block. The GROUP BY clause must follow the conditions in the WHERE clause and must precede the ORDER BY clause if one is used.

```
SELECT column1, column2
FROM table_name
WHERE [ conditions ]
GROUP BY column1, column2
ORDER BY column1, column2
```

Example

Consider the CUSTOMERS table is having the following records −

```
+-----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+-----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
```

```
|  3 | kaushik  |  23 | Kota       |  2000.00 |
|  4 | Chaitali |  25 | Mumbai     |  6500.00 |
|  5 | Hardik   |  27 | Bhopal     |  8500.00 |
|  6 | Komal    |  22 | MP         |  4500.00 |
|  7 | Muffy    |  24 | Indore     | 10000.00 |
+----+----------+-----+------------+----------+
```

If you want to know the total amount of the salary on each customer, then the GROUP BY query would be as follows.

```
SQL> SELECT NAME, SUM(SALARY) FROM CUSTOMERS
   GROUP BY NAME;
```

This would produce the following result −

```
+----------+-----------------+
| NAME     | SUM(SALARY)     |
+----------+-----------------+
| Chaitali |      6500.00    |
| Hardik   |      8500.00    |
| kaushik  |      2000.00    |
| Khilan   |      1500.00    |
| Komal    |      4500.00    |
| Muffy    |     10000.00    |
| Ramesh   |      2000.00    |
+----------+-----------------+
```

SQL - ORDER BY Clause

The SQL **ORDER BY** clause is used to sort the data in ascending or descending order, based on one or more columns. Some databases sort the query results in an ascending order by default.

Syntax

The basic syntax of the ORDER BY clause is as follows −

```
SELECT column-list
FROM table_name
[WHERE condition]
[ORDER BY column1, column2, .. columnN] [ASC | DESC];
```

You can use more than one column in the ORDER BY clause. Make sure whatever column you are using to sort that column should be in the column-list.

Example

Consider the CUSTOMERS table having the following records −

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
```

```
|  5 | Hardik    |  27 | Bhopal     |  8500.00 |
|  6 | Komal     |  22 | MP         |  4500.00 |
|  7 | Muffy     |  24 | Indore     | 10000.00 |
+-----+-------------+-------+--------------+-------------+
```

The following code block has an example, which would sort the result in an ascending order by the NAME and the SALARY −

```
SQL> SELECT * FROM CUSTOMERS
   ORDER BY NAME, SALARY;
```

This would produce the following result −

```
+-----+-------------+-------+--------------+-------------+
| ID  | NAME        | AGE   | ADDRESS      | SALARY      |
+-----+-------------+-------+--------------+-------------+
|  4 | Chaitali |  25 | Mumbai     |  6500.00 |
|  5 | Hardik    |  27 | Bhopal     |  8500.00 |
|  3 | kaushik   |  23 | Kota       |  2000.00 |
|  2 | Khilan    |  25 | Delhi      |  1500.00 |
|  6 | Komal     |  22 | MP         |  4500.00 |
|  7 | Muffy     |  24 | Indore     | 10000.00 |
|  1 | Ramesh    |  32 | Ahmedabad |  2000.00 |
+-----+-------------+-------+--------------+-------------+
```

The following code block has an example, which would sort the result in the descending order by NAME.

```
SQL> SELECT * FROM CUSTOMERS
   ORDER BY NAME DESC;
```

This would produce the following result −

```
+-----+-------------+-------+--------------+-------------+
| ID  | NAME        | AGE   | ADDRESS      | SALARY      |
+-----+-------------+-------+--------------+-------------+
|  1 | Ramesh    |  32 | Ahmedabad |  2000.00 |
|  7 | Muffy     |  24 | Indore     | 10000.00 |
|  6 | Komal     |  22 | MP         |  4500.00 |
|  2 | Khilan    |  25 | Delhi      |  1500.00 |
|  3 | kaushik   |  23 | Kota       |  2000.00 |
|  5 | Hardik    |  27 | Bhopal     |  8500.00 |
|  4 | Chaitali |  25 | Mumbai     |  6500.00 |
+-----+-------------+-------+--------------+-------------+
```

SQL - Transactions Control Command

Transaction Control

The following commands are used to control transactions.

- **COMMIT** − to save the changes.
- **ROLLBACK** − to roll back the changes.
- **SAVEPOINT** − creates points within the groups of transactions in which to ROLLBACK.

The COMMIT Command

The COMMIT command is the transactional command used to save changes invoked by a transaction to the database.

The COMMIT command is the transactional command used to save changes invoked by a transaction to the database. The COMMIT command saves all the transactions to the database since the last COMMIT or ROLLBACK command.

The syntax for the COMMIT command is as follows.

```
COMMIT;
```

Example

Consider the CUSTOMERS table having the following records −

```
+-----+----------+-------+-----------+----------+
| ID  | NAME     | AGE   | ADDRESS   | SALARY   |
+-----+----------+-------+-----------+----------+
|   1 | Ramesh   |   32  | Ahmedabad |  2000.00 |
|   2 | Khilan   |   25  | Delhi     |  1500.00 |
|   3 | kaushik  |   23  | Kota      |  2000.00 |
|   4 | Chaitali |   25  | Mumbai    |  6500.00 |
|   5 | Hardik   |   27  | Bhopal    |  8500.00 |
|   6 | Komal    |   22  | MP        |  4500.00 |
|   7 | Muffy    |   24  | Indore    | 10000.00 |
+-----+----------+-------+-----------+----------+
```

Following is an example which would delete those records from the table which have age = 25 and then COMMIT the changes in the database.

```
SQL> DELETE FROM CUSTOMERS
   WHERE AGE = 25;
SQL> COMMIT;
```

Thus, two rows from the table would be deleted and the SELECT statement would produce the following result.

```
+-----+----------+-------+-----------+----------+
| ID  | NAME     | AGE   | ADDRESS   | SALARY   |
+-----+----------+-------+-----------+----------+
|   1 | Ramesh   |   32  | Ahmedabad |  2000.00 |
|   3 | kaushik  |   23  | Kota      |  2000.00 |
|   5 | Hardik   |   27  | Bhopal    |  8500.00 |
|   6 | Komal    |   22  | MP        |  4500.00 |
|   7 | Muffy    |   24  | Indore    | 10000.00 |
+-----+----------+-------+-----------+----------+
```

The ROLLBACK Command

The ROLLBACK command is the transactional command used to undo transactions that have not already been saved to the database. This command can only be used to undo transactions since the last COMMIT or ROLLBACK command was issued.

The syntax for a ROLLBACK command is as follows −

```
ROLLBACK;
```

Example

Consider the CUSTOMERS table having the following records −

```
+-----+----------+-------+-----------+----------+
| ID  | NAME     | AGE   | ADDRESS   | SALARY   |
+-----+----------+-------+-----------+----------+
|  1  | Ramesh   |  32   | Ahmedabad |  2000.00 |
|  2  | Khilan   |  25   | Delhi     |  1500.00 |
|  3  | kaushik  |  23   | Kota      |  2000.00 |
|  4  | Chaitali |  25   | Mumbai    |  6500.00 |
|  5  | Hardik   |  27   | Bhopal    |  8500.00 |
|  6  | Komal    |  22   | MP        |  4500.00 |
|  7  | Muffy    |  24   | Indore    | 10000.00 |
+-----+----------+-------+-----------+----------+
```

Following is an example, which would delete those records from the table which have the age = 25 and then ROLLBACK the changes in the database.

```
SQL> DELETE FROM CUSTOMERS
   WHERE AGE = 25;
SQL> ROLLBACK;
```

Thus, the delete operation would not impact the table and the SELECT statement would produce the fol lowing result.

```
+-----+----------+-------+-----------+----------+
| ID  | NAME     | AGE   | ADDRESS   | SALARY   |
+-----+----------+-------+-----------+----------+
|  1  | Ramesh   |  32   | Ahmedabad |  2000.00 |
|  2  | Khilan   |  25   | Delhi     |  1500.00 |
|  3  | kaushik  |  23   | Kota      |  2000.00 |
|  4  | Chaitali |  25   | Mumbai    |  6500.00 |
|  5  | Hardik   |  27   | Bhopal    |  8500.00 |
|  6  | Komal    |  22   | MP        |  4500.00 |
|  7  | Muffy    |  24   | Indore    | 10000.00 |
+-----+----------+-------+-----------+----------+
```

The SAVEPOINT Command

A SAVEPOINT is a point in a transaction when you can roll the transaction back to a certain point without rolling back the entire transaction.

The syntax for a SAVEPOINT command is as shown below.

```
SAVEPOINT SAVEPOINT_NAME;
```

This command serves only in the creation of a SAVEPOINT among all the transactional statements. The ROLLBACK command is used to undo a group of transactions.

The syntax for rolling back to a SAVEPOINT is as shown below.

```
ROLLBACK TO SAVEPOINT_NAME;
```

Following is an example where you plan to delete the three different records from the CUSTOMERS table. You want to create a SAVEPOINT before each delete, so that you can ROLLBACK to any SAVEPOINT at any time to return the appropriate data to its original state.

Example

Consider the CUSTOMERS table having the following records.

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

The following code block contains the series of operations.

```
SQL> SAVEPOINT SP1;
Savepoint created.
SQL> DELETE FROM CUSTOMERS WHERE ID=1;
1 row deleted.
SQL> SAVEPOINT SP2;
Savepoint created.
SQL> DELETE FROM CUSTOMERS WHERE ID=2;
1 row deleted.
SQL> SAVEPOINT SP3;
Savepoint created.
SQL> DELETE FROM CUSTOMERS WHERE ID=3;
1 row deleted.
```

Now that the three deletions have taken place, let us assume that you have changed your mind and decided to ROLLBACK to the SAVEPOINT that you identified as SP2. Because SP2 was created after the first deletion, the last two deletions are undone −

```
SQL> ROLLBACK TO SP2;
Rollback complete.
```

SQL - Using Joins

The SQL **Joins** 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.

Consider the following two tables –

Table 1 – CUSTOMERS Table

```
+-----+----------+-------+-----------+----------+
| ID  | NAME     | AGE   | ADDRESS   | SALARY   |
+-----+----------+-------+-----------+----------+
|  1  | Ramesh   |  32   | Ahmedabad |  2000.00 |
|  2  | Khilan   |  25   | Delhi     |  1500.00 |
|  3  | kaushik  |  23   | Kota      |  2000.00 |
|  4  | Chaitali |  25   | Mumbai    |  6500.00 |
|  5  | Hardik   |  27   | Bhopal    |  8500.00 |
|  6  | Komal    |  22   | MP        |  4500.00 |
|  7  | Muffy    |  24   | Indore    | 10000.00 |
+-----+----------+-------+-----------+----------+
```

Table 2 – ORDERS Table

```
+-------+---------------------+-------------+--------+
|OID    | DATE                | CUSTOMER_ID | AMOUNT |
+-------+---------------------+-------------+--------+
| 102   | 2009-10-08 00:00:00 |           3 |   3000 |
| 100   | 2009-10-08 00:00:00 |           3 |   1500 |
| 101   | 2009-11-20 00:00:00 |           2 |   1560 |
| 103   | 2008-05-20 00:00:00 |           4 |   2060 |
+-------+---------------------+-------------+--------+
```

Now, let us join these two tables in our SELECT statement as shown below.

```
SQL> SELECT ID, NAME, AGE, AMOUNT
   FROM CUSTOMERS, ORDERS
   WHERE  CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

This would produce the following result.

```
+-----+----------+-------+-----------+
| ID  | NAME     | AGE   | AMOUNT    |
+-----+----------+-------+-----------+
|  3  | kaushik  |  23   |    3000   |
|  3  | kaushik  |  23   |    1500   |
|  2  | Khilan   |  25   |    1560   |
|  4  | Chaitali |  25   |    2060   |
+-----+----------+-------+-----------+
```

Here, it is noticeable that the join is performed in the WHERE clause. Several operators can be used to join tables, such as =, <, >, <>, <=, >=, !=, BETWEEN, LIKE, and NOT; they can all be used to join tables. However, the most common operator is the equal to symbol.

There are different types of joins available in SQL –

- INNER JOIN – returns rows when there is a match in both tables.
- LEFT JOIN – returns all rows from the left table, even if there are no matches in the right table.

- RIGHT JOIN – returns all rows from the right table, even if there are no matches in the left table.
- FULL JOIN – returns rows when there is a match in one of the tables.

SQL - INNER JOINS

The most important and frequently used of the joins is the **INNER JOIN**. They are also referred to as an **EQUIJOIN**.

The INNER JOIN creates a new result table by combining column values of two tables (table1 and table2) based upon the join-predicate. The query compares each row of table1 with each row of table2 to find all pairs of rows which satisfy the join-predicate. When the join-predicate is satisfied, column values for each matched pair of rows of A and B are combined into a result row.

Syntax

The basic syntax of the **INNER JOIN** is as follows.

```
SELECT table1.column1, table2.column2...
FROM table1
INNER JOIN table2
ON table1.common_field = table2.common_field;
```

Example

Consider the following two tables.

Table 1 − CUSTOMERS Table is as follows.

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000.00
2	Khilan	25	Delhi	1500.00
3	kaushik	23	Kota	2000.00
4	Chaitali	25	Mumbai	6500.00
5	Hardik	27	Bhopal	8500.00
6	Komal	22	MP	4500.00
7	Muffy	24	Indore	10000.00

Table 2 − ORDERS Table is as follows.

OID	DATE	CUSTOMER_ID	AMOUNT
102	2009-10-08 00:00:00	3	3000
100	2009-10-08 00:00:00	3	1500
101	2009-11-20 00:00:00	2	1560
103	2008-05-20 00:00:00	4	2060

Now, let us join these two tables using the INNER JOIN as follows −

```
SQL> SELECT  ID, NAME, AMOUNT, DATE
   FROM CUSTOMERS
   INNER JOIN ORDERS
   ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

This would produce the following result.

```
+-----+-------------+-----------+---------------------------+
| ID  | NAME        | AMOUNT    | DATE                      |
+-----+-------------+-----------+---------------------------+
|   3 | kaushik     |    3000   | 2009-10-08 00:00:00       |
|   3 | kaushik     |    1500   | 2009-10-08 00:00:00       |
|   2 | Khilan      |    1560   | 2009-11-20 00:00:00       |
|   4 | Chaitali    |    2060   | 2008-05-20 00:00:00       |
+-----+-------------+-----------+---------------------------+
```

SQL - LEFT JOINS

The SQL **LEFT JOIN** returns all rows from the left table, even if there are no matches in the right table. This means that if the ON clause matches 0 (zero) records in the right table; the join will still return a row in the result, but with NULL in each column from the right table.

This means that a left join returns all the values from the left table, plus matched values from the right table or NULL in case of no matching join predicate.

Syntax

The basic syntax of a **LEFT JOIN** is as follows.

```
SELECT table1.column1, table2.column2...
FROM table1
LEFT JOIN table2
ON table1.common_field = table2.common_field;
```

Here, the given condition could be any given expression based on your requirement.

Example

Consider the following two tables,

Table 1 − CUSTOMERS Table is as follows.

```
+-----+-------------+-------+-------------+-------------+
| ID  | NAME        | AGE   | ADDRESS     | SALARY      |
+-----+-------------+-------+-------------+-------------+
|   1 | Ramesh      |   32  | Ahmedabad   |   2000.00   |
|   2 | Khilan      |   25  | Delhi       |   1500.00   |
|   3 | kaushik     |   23  | Kota        |   2000.00   |
|   4 | Chaitali    |   25  | Mumbai      |   6500.00   |
|   5 | Hardik      |   27  | Bhopal      |   8500.00   |
|   6 | Komal       |   22  | MP          |   4500.00   |
|   7 | Muffy       |   24  | Indore      |  10000.00   |
+-----+-------------+-------+-------------+-------------+
```

Table 2 − Orders Table is as follows.

```
+-------+---------------------------+-----------------+-----------+
| OID   | DATE                      | CUSTOMER_ID     | AMOUNT    |
```

```
+-------+---------------------------+-----------------+-----------+
| 102 | 2009-10-08 00:00:00 |            3 |   3000 |
| 100 | 2009-10-08 00:00:00 |            3 |   1500 |
| 101 | 2009-11-20 00:00:00 |            2 |   1560 |
| 103 | 2008-05-20 00:00:00 |            4 |   2060 |
+-------+---------------------------+-----------------+-----------+
```

Now, let us join these two tables using the LEFT JOIN as follows.

```
SQL> SELECT  ID, NAME, AMOUNT, DATE
   FROM CUSTOMERS
   LEFT JOIN ORDERS
   ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

This would produce the following result −

```
+-----+-------------+-----------+---------------------------+
| ID | NAME      | AMOUNT | DATE                      |
+-----+-------------+-----------+---------------------------+
|  1 | Ramesh    |   NULL | NULL                      |
|  2 | Khilan    |   1560 | 2009-11-20 00:00:00 |
|  3 | kaushik   |   3000 | 2009-10-08 00:00:00 |
|  3 | kaushik   |   1500 | 2009-10-08 00:00:00 |
|  4 | Chaitali |   2060 | 2008-05-20 00:00:00 |
|  5 | Hardik    |   NULL | NULL                      |
|  6 | Komal     |   NULL | NULL                      |
|  7 | Muffy     |   NULL | NULL                      |
+-----+-------------+-----------+---------------------------+
```

SQL - RIGHT JOINS

The SQL **RIGHT JOIN** returns all rows from the right table, even if there are no matches in the left table. This means that if the ON clause matches 0 (zero) records in the left table; the join will still return a row in the result, but with NULL in each column from the left table.

This means that a right join returns all the values from the right table, plus matched values from the left table or NULL in case of no matching join predicate.

Syntax

The basic syntax of a **RIGHT JOIN** is as follow.

```
SELECT table1.column1, table2.column2...
FROM table1
RIGHT JOIN table2
ON table1.common_field = table2.common_field;
```

Example

Consider the following two tables,

Table 1 − CUSTOMERS Table is as follows.

```
+-----+-------------+-------+--------------+-------------+
| ID | NAME      | AGE | ADDRESS    | SALARY    |
```

```
+------+-------------+-------+--------------+-------------+
|  1   | Ramesh      |  32   | Ahmedabad    |  2000.00    |
|  2   | Khilan      |  25   | Delhi        |  1500.00    |
|  3   | kaushik     |  23   | Kota         |  2000.00    |
|  4   | Chaitali    |  25   | Mumbai       |  6500.00    |
|  5   | Hardik      |  27   | Bhopal       |  8500.00    |
|  6   | Komal       |  22   | MP           |  4500.00    |
|  7   | Muffy       |  24   | Indore       | 10000.00    |
+------+-------------+-------+--------------+-------------+
```

Table 2 − ORDERS Table is as follows.

```
+-------+---------------------------+-----------------+-----------+
|OID    | DATE                      | CUSTOMER_ID | AMOUNT |
+-------+---------------------------+-----------------+-----------+
| 102 | 2009-10-08 00:00:00 |            3 |    3000 |
| 100 | 2009-10-08 00:00:00 |            3 |    1500 |
| 101 | 2009-11-20 00:00:00 |            2 |    1560 |
| 103 | 2008-05-20 00:00:00 |            4 |    2060 |
+-------+---------------------------+-----------------+-----------+
```

Now, let us join these two tables using the RIGHT JOIN as follows.

```
SQL> SELECT  ID, NAME, AMOUNT, DATE
   FROM CUSTOMERS
   RIGHT JOIN ORDERS
   ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

This would produce the following result −

```
+--------+-------------+-----------+----------------------------+
| ID     | NAME        | AMOUNT | DATE                       |
+--------+-------------+-----------+----------------------------+
|      3 | kaushik   |    3000 | 2009-10-08 00:00:00 |
|      3 | kaushik   |    1500 | 2009-10-08 00:00:00 |
|      2 | Khilan    |    1560 | 2009-11-20 00:00:00 |
|      4 | Chaitali |    2060 | 2008-05-20 00:00:00 |
+--------+-------------+-----------+----------------------------+
```

SQL - FULL JOINS

The SQL **FULL JOIN** combines the results of both left and right outer joins.

The joined table will contain all records from both the tables and fill in NULLs for missing matches on either side.

Syntax

The basic syntax of a **FULL JOIN** is as follows −

```
SELECT table1.column1, table2.column2...
FROM table1
FULL JOIN table2
ON table1.common_field = table2.common_field;
```

Here, the given condition could be any given expression based on your requirement.

Example

Consider the following two tables.

Table 1 − CUSTOMERS Table is as follows.

```
+-----+----------+-------+-----------+----------+
| ID  | NAME     | AGE   | ADDRESS   | SALARY   |
+-----+----------+-------+-----------+----------+
|   1 | Ramesh   |   32  | Ahmedabad |  2000.00 |
|   2 | Khilan   |   25  | Delhi     |  1500.00 |
|   3 | kaushik  |   23  | Kota      |  2000.00 |
|   4 | Chaitali |   25  | Mumbai    |  6500.00 |
|   5 | Hardik   |   27  | Bhopal    |  8500.00 |
|   6 | Komal    |   22  | MP        |  4500.00 |
|   7 | Muffy    |   24  | Indore    | 10000.00 |
+-----+----------+-------+-----------+----------+
```

Table 2 − ORDERS Table is as follows.

```
+-------+---------------------------+-------------+--------+
|OID    | DATE                      | CUSTOMER_ID | AMOUNT |
+-------+---------------------------+-------------+--------+
| 102   | 2009-10-08 00:00:00       |           3 |   3000 |
| 100   | 2009-10-08 00:00:00       |           3 |   1500 |
| 101   | 2009-11-20 00:00:00       |           2 |   1560 |
| 103   | 2008-05-20 00:00:00       |           4 |   2060 |
+-------+---------------------------+-------------+--------+
```

Now, let us join these two tables using FULL JOIN as follows.

```
SQL> SELECT  ID, NAME, AMOUNT, DATE
   FROM CUSTOMERS
   FULL JOIN ORDERS
   ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

This would produce the following result −

```
+--------+-------------+-----------+----------------------------+
| ID     | NAME        | AMOUNT    | DATE                       |
+--------+-------------+-----------+----------------------------+
|      1 | Ramesh      |    NULL   | NULL                       |
|      2 | Khilan      |    1560   | 2009-11-20 00:00:00        |
|      3 | kaushik     |    3000   | 2009-10-08 00:00:00        |
|      3 | kaushik     |    1500   | 2009-10-08 00:00:00        |
|      4 | Chaitali    |    2060   | 2008-05-20 00:00:00        |
|      5 | Hardik      |    NULL   | NULL                       |
|      6 | Komal       |    NULL   | NULL                       |
|      7 | Muffy       |    NULL   | NULL                       |
|      3 | kaushik     |    3000   | 2009-10-08 00:00:00        |
|      3 | kaushik     |    1500   | 2009-10-08 00:00:00        |
|      2 | Khilan      |    1560   | 2009-11-20 00:00:00        |
|      4 | Chaitali    |    2060   | 2008-05-20 00:00:00        |
+--------+-------------+-----------+----------------------------+
```

SQL - Using Views

A view is nothing more than a SQL statement that is stored in the database with an associated name. A view is actually a composition of a table in the form of a predefined SQL query.

A view can contain all rows of a table or select rows from a table. A view can be created from one or many tables which depends on the written SQL query to create a view.

Views, which are a type of virtual tables allow users to do the following −

- Structure data in a way that users or classes of users find natural or intuitive.
- Restrict access to the data in such a way that a user can see and (sometimes) modify exactly what they need and no more.
- Summarize data from various tables which can be used to generate reports.

Creating Views

Database views are created using the **CREATE VIEW** statement. Views can be created from a single table, multiple tables or another view.

To create a view, a user must have the appropriate system privilege according to the specific implementation.

The basic **CREATE VIEW** syntax is as follows −

```
CREATE VIEW view_name AS
SELECT column1, column2.....
FROM table_name
WHERE [condition];
```

You can include multiple tables in your SELECT statement in a similar way as you use them in a normal SQL SELECT query.

Example

Consider the CUSTOMERS table having the following records −

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

Following is an example to create a view from the CUSTOMERS table. This view would be used to have customer name and age from the CUSTOMERS table.

```
SQL > CREATE VIEW CUSTOMERS_VIEW AS
SELECT name, age
FROM  CUSTOMERS;
```

Now, you can query CUSTOMERS_VIEW in a similar way as you query an actual table. Following is an example for the same.

```
SQL > SELECT * FROM CUSTOMERS_VIEW;
```

This would produce the following result.

```
+--------------+-------+
| name      | age |
+--------------+-------+
| Ramesh    |  32 |
| Khilan    |  25 |
| kaushik   |  23 |
| Chaitali |  25 |
| Hardik    |  27 |
| Komal     |  22 |
| Muffy     |  24 |
+--------------+-------+
```

Dropping Views

Obviously, where you have a view, you need a way to drop the view if it is no longer needed. The syntax is very simple and is given below −

```
DROP VIEW view_name;
```

Following is an example to drop the CUSTOMERS_VIEW from the CUSTOMERS table.

```
DROP VIEW CUSTOMERS_VIEW;
```

SQL ANY and ALL Operators

The ANY and ALL operators allow you to perform a comparison between a single column value and a range of other values.

The SQL ANY Operator

The ANY operator:

- returns a boolean value as a result
- returns TRUE if ANY of the subquery values meet the condition

ANY means that the condition will be true if the operation is true for any of the values in the range.

ANY Syntax

```
SELECT column_name(s)
FROM table_name
WHERE column_name operator ANY
```

```
  (SELECT column_name
  FROM table_name
  WHERE condition);
```

Example

```
SELECT ProductName
FROM Products
WHERE ProductID = ANY
  (SELECT ProductID
  FROM OrderDetails
  WHERE Quantity = 10);
```

The SQL ALL Operator

The ALL operator:

- returns a boolean value as a result
- returns TRUE if ALL of the subquery values meet the condition
- is used with SELECT, WHERE and HAVING statements

ALL means that the condition will be true only if the operation is true for all values in the range.

ALL Syntax With SELECT

```
SELECT ALL column_name(s)
FROM table_name
WHERE condition;
```

Example

```
SELECT ALL ProductName
FROM Products
WHERE TRUE;
```