SELECT	SELECT query is used to retrieve a data from SQL tables.
SYNTAX	**To get specific columns from the table:** Select column1_name, column2_name From table_name; **To get all columns from the table:** Select * from table_name;
EXAMPLE	**To get specific columns from the table:** Select roll_no,name From STUDENT; **To get all columns from the table:** Select * from STUDENT;
SELECT WITH WHERE CLAUSE	It is use to get specific columns from the table
SYNTAX	**To get specific columns from the table:** Select column1_name, column2_name From table_name Where column_name=value;
EXAMPLE	**To get specific columns from the table:** Select ROLL_NO,NAME From STUDENT Where STREAM='BBA';

5.1 USING WHERE CLAUSE AND OPERATORS WITH WHERE CLAUSE:

5.1.1 IN, BETWEEN , LIKE, NOT IN, =, !=, >, =, <=, WILDCARD OPERATORS

Statement	Description
IN	The IN operator allows you to specify multiple values in a WHERE clause. The IN operator is a shorthand for multiple OR conditions.

SYNTAX	**IN:** SELECT *column_name(s)* FROM *table_name* WHERE *column_name* IN (*value1*, *value2*, ...);
	NOT IN: SELECT *column_name(s)* FROM *table_name* WHERE *column_name NOT* IN (*value1*, *value2*, ...);

EXAMPLE	**Display record of student who belong to 'BCA','BBA' or 'BCOM' stream.** select * From STUDENT WHERE STREAM IN('BCA','BBA','MCOM');

<table>
<tr><td></td><td><b>Display record of student who does not belong to ‘BCA’,’BBA’ or ‘BCOM’ stream.</b>
select * From STUDENT WHERE STREAM NOT IN('BCA','BBA','MCOM');</td></tr>
<tr><td><b>BETWEEN</b></td><td>The BETWEEN operator is used to select values within a range. The BETWEEN operator selects values within a range. The values can be numbers, text, or dates.</td></tr>
<tr><td>SYNTAX</td><td>SELECT column_name(s) FROM table_name WHERE column_name BETWEEN value1 AND value2;</td></tr>
<tr><td>EXAMPLE</td><td><b>Select student list whos DOB is between ‘1997-08-06’ and ‘1998-09-26’.</b>
select * From STUDENT Where DOB BETWEEN '1997-08-06' AND '1998-09-26';</td></tr>
<tr><td><b>LIKE</b></td><td>• The SQL <b>LIKE</b> clause is used to compare a value to similar values using wildcard operators. The LIKE operator is used in a WHERE clause to search for a specified pattern in a column.
• The LIKE operator is used to search for a specified pattern in a column.
• You can also combine any number of conditions using AND or OR operators. In SQL, wildcard characters are used with the SQL LIKE operator.
• <b>SQL wildcards</b> are used to search for data within a table.
• There are <b>two wildcards</b> used in conjunction with the LIKE operator:

<b>% - The percent sign represents zero, one, or multiple characters</b>
<b>_ - The underscore represents a single character.</b>

<table>
<tr><th>LIKE OPRATOR</th><th>DESCRIPTION</th></tr>
<tr><td>WHERE NAME LIKE ‘a%’</td><td>Finds any values that starts with “a”</td></tr>
<tr><td>WHERE NAME LIKE ‘%a’</td><td>Finds any values that ends with “a”</td></tr>
<tr><td>WHERE NAME LIKE ‘%or%’</td><td>Finds any values that have “or” in any position</td></tr>
</table>
</td></tr>
</table>

	WHERE NAME LIKE ‘_r%’ Finds any values that have “r” in the second position WHERE NAME LIKE ‘a%o’ Finds any values that starts with “a” and ends with “o”

SYNTAX	**Like Operator:** SELECT column1_name, column2_name FROM table_name WHERE column_name LIKE pattern;
	Not Like Operator:
	Using the NOT keyword allows you to select records that does NOT match the pattern.
	SELECT column1_name, column2_name FROM table_name WHERE column_name NOT LIKE pattern;

EXAMPLE	**List the records of students having ‘a’ as last character.** select * from STUDENT Where NAME like '%a'; **List records of students having ‘h’ at any position of name.** select * from STUDENT Where NAME like '%h%'; **List records of students having born year ‘1998’.** select * from STUDENT Where DOB like '1998%'; **Select all the students with a born year NOT containing ‘1998’.** select * from STUDENT Where DOB NOT like '1998%'
WHERE CLUSE WITH DIFFERENT OPRATORS	**=,!=,>,<,>=,<=,BETWEEN,RANGE,LIKE,PATTERN SEARCH,IN,AND,OR**

5.1.2 ORDER BY, GROUP BY, DISTINCT

Statement	Description
ORDER BY	The SQL **ORDER BY** clause is used to sort the data in ascending or descending order.
SYNTAX	SELECT Column1_name,Column2_name From Table_name [Where Condition] ORDER BY Column_name [ASC/DESC];
EXAMPLE	**To Arrange Name of STUDENT in Ascending order.** select * from STUDENT ORDER BY NAME ASC;

	To Arrange Name of STUDENT in Descending order. select * from STUDENT ORDER BY NAME DESC;
GROUP BY	The GROUP BY statement is used in conjunction with the aggregate functions to group the result-set by one or more columns. Group by clause is used to group the results of a SELECT query based on one or more columns.
SYNTAX	SELECTcolumn_name,aggregate_function(column_name) FROM table_name WHERE column_name operator value GROUP BY column_name;
EXAMPLE	Report the no. of students in each stream. select STREAM,COUNT(STREAM) As Student FROM STUDENT GROUP BY STREAM;
DISTINCT	it is used to fetch out different values from a table.
SYNTAX	**To get different values from the table:** Select distinct column1_name From table_name;
EXAMPLE	**To get different values from the table:** Select DISTINCT STREAM From STUDENT;

5.1.3 AND, OR OPERATORS, EXISTS AND NOT EXISTS

Statement	**Description**
AND/OR	These operators are used to filter records. **AND Operator:** This operator displays record(s) if both conditions are **true**. **OR Operator:** This operator displays record(s) if any one of the condition is **true.**

EXAMPLE	**AND EX:** Find out those students records having greater than 70 marks **and** Stream is BBA.
	Query: SELECT * From STUDENT WHERE MARKS>75 AND STREAM='BBA';
	OR EX: Find out those students records having greater than equal to 50 marks **or** stream is BCA.
	Query: select NAME from STUDENT where MARKS>=50 OR STREAM ='BCA';

EXISTS	The EXISTS condition in SQL is used to check whether the result of a correlated nested query is empty (contains no

	tuples) or not. The result of EXISTS is a boolean value True or False. It can be used in a SELECT, UPDATE, INSERT or DELETE statement.

SYNTAX	SELECT column_name(s)
	FROM table_name
	WHERE EXISTS
	(SELECT column_name FROM table_name WHERE condition);

EXAMPLE	To fetch the first and last name of the customers who placed atleast one order.
	
	SELECT fname, lname
	FROM Customers
	WHERE EXISTS (SELECT * FROM Orders
	WHERE Customers.customer_id = Orders.c_id);

NOT EXIST	The NOT EXISTS in SQL Server will check the Subquery for rows existence, and if there are no rows then it will return TRUE, otherwise FALSE

SYNTAX	SELECT column_name(s)
	FROM table_name
	WHERE NOT EXISTS
	(SELECT column_name FROM table_name WHERE condition);

EXAMPLE	Fetch last and first name of the customers who has not placed any order.
	SELECT lname, fname
	FROM Customer
	WHERE NOT EXISTS (SELECT * FROM Orders
	WHERE Customers.customer_id = Orders.c_id);

5.1.4 USE OF ALIAS

Statement	Description
ALIASES	Aliases are the temporary names given to table or column for the purpose of a particular SQL query. It is used when name of column or table is used other than their original names, but the modified name is only temporary.

	• Aliases are created to make table or column names more readable. • The renaming is just a temporary change and table name does not change in the original database. • Aliases are useful when table or column names are big or not very readable. • These are preferred when there are more than one table involved in a query.
SYNTAX	**For column alias:** SELECT *column_name* AS *alias_name* FROM *table_name;* **alias_name:** temporary alias name to be used in replacement of original column name **table_name:** name of table **For table alias:** SELECT *column_name(s)* FROM *table_name* AS *alias_name;* **table_name:** name of table **alias_name:** temporary alias name to be used in replacement of original table name

EXAMPLE	Fetch ROLL_NO from Student table using CODE as alias name.
	SELECT ROLL_NO AS CODE FROM Student;
	Fetch all the orders from the customer with CustomerID=4. We use the "Customers" and "Orders" tables, and give them the table aliases of "c" and "o" respectively.
	SELECT o.OrderID, o.OrderDate, c.CustomerName
	FROM Customers AS c, Orders AS o
	WHERE c.CustomerName='Around the Horn' AND c.CustomerID=o.CustomerID;

5.2 CONSTRAINTS (TABLE LEVEL AND ATTRIBUTE LEVEL)

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
- **UNIQUE** - Ensures that all values in a column are different
- **PRIMARY KEY** - A combination of a NOT NULL and UNIQUE. Uniquely identifies each row in a table
- **FOREIGN KEY** - Uniquely identifies a row/record in another table
- **CHECK** - Ensures that all values in a column satisfies a specific condition
- **DEFAULT** - Sets a default value for a column when no value is specified
- **INDEX** - Used to create and retrieve data from the database very quickly

5.2.1 NOT NULL, CHECK, DEFAULT

Statement	**Description**
NOT NULL	By default, a column can hold NULL values. If you do not want a column to have a NULL value, then you need to define such a constraint on this column specifying that NULL is now not allowed for that column.

SYNTAX	*CREATE TABLE* table_name *(* column1 datatype(size) NOT NULL*,*
	column2 datatype(size) NOT NULL*,*
	column3 datatype(size) *NOT NULL);*

EXAMPLE	**create a new table called CUSTOMERS and adds five columns, three of which, are ID NAME and AGE, In this we specify not to accept NULLs.** Create table Customer (ID Numeric NOT NULL, NAME varchar(10) NOT NULL, AGE Numeric NOT NULL, ADDRESS Varchar(30), SALARY Numeric);
CHECK	• 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 single column it allows 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. • The CHECK Constraint enables a condition to check the value being entered into a record. If the condition evaluates to false, the record violates the constraint and isn't entered the table.

SYNTAX	CREATE TABLE Child_table_name (column1 datatype(size),
	column2 datatype(size),
	column3 datatype(size),
	CHECK (Condition));

EXAMPLE	EX:1:- create table STUD1 (STUD_ID numeric Primary key, STUD_NAME Varchar(25), STUD_STREAM Varchar(10), STUD_AGE numeric, CHECK (STUD_AGE>=18)); EX:2:- create table STU (STUD_ID numeric Primary key, STUD_NAME varchar(20) NOT NULL , CHECK(STUD_NAME like 'H%')); EX:3:- create table EMP (id numeric primary key, gender varchar(10), check (gender in ('Male','Female')));
	CHECK CONSTRAINT VIOLATION ERROR: *Check constraint violation SYS_CT_69 table: STU.*
DEFAULT	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	*CREATE TABLE* table_name *(* column1 datatype(size),
	column2 datatype(size),
	column3 datatype(size) *DEFAULT VALUE);*

EXAMPLE	create table STUD3 (ID numeric primary key, Name varchar(20) NOT NULL, ClgName varchar(35) DEFAULT 'VTCBB');

5.2.2 UNIQUE, PRIMARY KEY, FOREIGN KEY

Statement	Description
UNIQUE KEY	• The UNIQUE Constraint prevents two records from having identical values in a column. 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.

SYNTAX	*CREATE TABLE* table_name *(* column1 datatype(size) UNIQUE,
	column2 datatype(size),
	column3 datatype(size) *);*

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

PRIMARY KEY	• The PRIMARY KEY constraint uniquely identifies each record in a database table. Primary keys must contain UNIQUE values, and cannot contain NULL values. • A table can have only one primary key, which may consist of single or multiple fields. • When multiple fields are used as a primary key, they are called a composite key. If a table has a primary key defined on any field, the
SYNTAX	**Primary Key at Column level** CREATE TABLE table_name (column1 datatype(size) PrimaryKey, column2 datatype(size), column3 datatype(size)); **Primary Key at Table level** CREATE TABLE table_name(column1 datatype(size), column2 datatype(size), column3 datatype(size), Primary key (Column_name));
EXAMPLE	***Primary Key at Column level*** Create table Customer (ID Numeric Primary Key,

```
                                   NAME varchar(10) NOT NULL,
                                   AGE Numeric NOT NULL,
                                   ADDRESS Varchar(30),
                                   SALARY Numeric );
```

Primary Key at Table level

```
Create table Customer ( ID Numeric,
                                   NAME varchar(10) NOT NULL,
                                   AGE Numeric NOT NULL,
                                   ADDRESS Varchar(30),
                                   SALARY Numeric,
                                   Primary key(ID) );
```

FOREIGN KEY

A FOREIGN KEY is a key used to link two tables together. A FOREIGN KEY is a field (or collection of fields) in one table that refers to the PRIMARY KEY in another table. The table containing the foreign key is called the child table, and the table containing the candidate key is called the referenced or parent table.

relationship between 2 tables matches the Primary Key in one of the tables with a Foreign Key in the second table.

Person Table:

PERSONID	FIRSTNAME	LASTNAME	AGE
1	MOHINI	PATEL	26
2	KAJAL	BHANUSHALI	25
3	KRISHNA	PATEL	26

Order Table

ORDERID	ORDERNUMBER	PERSONID
1	7752	3
2	8292	3
3	7884	1
4	4477	2

NOTE:

- 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 is used to prevent actions that would destroy links between tables. • The FOREIGN KEY constraint also prevents invalid data from being inserted into the foreign key column, because it has to be one of the values contained in the table it points to.

SYNTAX	*CREATE TABLE Child*_table_name *(* column1 datatype(size)*,*
	column2 datatype(size)*,*
	column3 datatype(size),
	Foreign key (Column_name) references Parent_table(Column_name));

EXAMPLE	**Parent table: (Person table)** create table person (PersonId numeric primary key, Firstname varchar(20), Lastname varchar(20), Age Numeric, Address Varchar(25)); **Child table: (Order1 table)** create table Order1 (OrderId numeric primary key, OrderNumber numeric, PersonId numeric, foreign Key(PersonId) references Person(PersonId));

5.2.3 ON DELETE CASCADE

There are two ways to maintain the integrity of data in Child table, when a particular record is deleted in main table.

When two tables are connected with Foreign key, and certain data in the main table is deleted, for which record exit in child table too, then we must have some mechanism to save the integrity of data in child table.

Now, let's see how foreign keys in SQL preserve data integrity.

To preserve this integrity we need to set ON DELETEBCASCADE AND ON UPDATE CASDATE to foreign key. So, that when you change or delete data in Parent table then the related records in the child table should also be change.

So here, we are going to delete the referential data that removes records from both tables. We have defined the foreign key in the **order table** as:

FOREIGN KEY (PersonId) REFERENCES Person(PersonId)

ON DELETE CASCADE

ON UPDATE CASCADE.

It means if we delete any person record from the person table, then the related records in the order table should also be deleted. And the ON UPDATE CASCADE will updates automatically on the parent table to referenced fields in the child table (Here, it is PersonId).

- Execute this statement that deletes a record from the table whose name is **MOHINI**.

  DELETE FROM Person **WHERE Name**='MOHINI';

- This action will delete name MOHINI from both the tables.

- Now, test the **ON UPDATE CASCADE**. Here, we are going to update the PersonId of **MOHINI** in the contact table as

  UPDATE Person **SET** id=3 **WHERE Name**='MOHINI';

- This action will update id in both the tables with PersonId of MOHINI=3

5.3 SQL FUNCTIONS :

5.3.1 AGGREGATE FUNCTIONS: AVG(), MAX(), MIN(), SUM(), COUNT(), FIRST(), LAST().

- SQL aggregation function is used to perform the calculations on multiple rows of a single column of a table. It returns a single value.
- It is also used to summarize the data.

Statement	Description
AVG()	The AVG aggregate function will return an average of the values. This also is generally done on numbers.

SYNTAX	Select avg(Field) AS Column_name From table_name;
EXAMPLE	select avg(MARKS) AS Total_Marks from STUDENT;
MAX() & MIN()	The MIN and MAX aggregate functions report the minimum and maximum values. In addition to being used with numeric datatypes, they can be also used with dates to report the earliest and latest dates and with text to report the lowest and highest alphabetically.

SYNTAX	**Min():** Select Min(Field) AS Column_name From Table_name;
	Max(): Select Max(Field) AS Column_name From Table_name;

EXAMPLE	**Min():** select Min(MARKS) As MINIMUM from STUDENT;
	Max(): select MAx(MARKS) As MAXIMUM from STUDENT;

SUM()	To calculate totals.
SYNTAX	Select sum(Field) AS Column_name From table_name;
EXAMPLE	select sum(MARKS) AS Total_Marks from STUDENT;
COUNT()	The COUNT aggregate function simply counts the resulting values or rows. Without a WHERE clause, COUNT counts all the rows in the table. If you add a WHERE clause it will count the rows that are returned. You can use COUNT on any datatype. The COUNT aggregate function can take as an argument either a field name or an asterisk (*). Using an asterisk will simply count the rows. Counting a field will count the number of notnull values in that field.
SYNTAX	**To Count particular Field:**Select count(Field) AS Column_name From Table_name; **To count Total no of Records in a table:** Select count(*) AS Column_name From Table_name;
EXAMPLE	**To count total no. of streams in student table.** select count(STREAM) As NO_STREAM from STUDENT; **To count total no of records in a table student.** select count(*) As Total_recods from STUDENT;
FIRST()	This function returns the first value of the column which you choose.
SYNTAX	SELECT FIRST(ColumnName)

	FROM TableName;

EXAMPLE	SELECT FIRST(Marks)
	FROM Students;

LAST()	Used to return the last value of the column which you choose.

SYNTAX	SELECT LAST(ColumnName)
	FROM TableName;

EXAMPLE	SELECT LAST(Marks)
	FROM Students;

5.3.2 SCALAR FUNCTIONS: UCASE(), LCASE(), ROUND(), MID().

These functions are based on user input, these too returns single value.

Statement	**Description**
UCASE()	This function is used to convert a string column values to Uppercase.

SYNTAX	**SELECT UCASE(ColumnName)
	FROM TableName;**

EXAMPLE	**SELECT UCASE(StudentName)
	FROM Students;**

LCASE()	Used to convert string column values to lowercase

SYNTAX	**SELECT LCASE(ColumnName)
	FROM TableName;**

EXAMPLE	**SELECT LCASE(StudentName)
	FROM Students;**

ROUND()	Rounds off a numeric value to the nearest integer.

SYNTAX	**SELECT ROUND(ColumnName, Decimals)
	FROM TableName;**

EXAMPLE	**SELECT ROUND(Marks) FROM Students;**
MID()	Extracts substrings in SQL from column values having String data type.

SYNTAX	**SELECT MID(ColumnName, Start, Length)
	FROM TableName;**

EXAMPLE	**SELECT MID(StudentName, 2, 3) FROM Students;**

5.4 CREATING SEQUENCE

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

- A sequence is a user defined schema bound object that generates a sequence of numeric values.
- Sequences are frequently used in many databases because many applications require each row in a table to contain a unique value and sequences provides an easy way to generate them.
- The sequence of numeric values is generated in an **ascending or descending order** at defined intervals and can be configured to restart when exceeds max_value.

- **SYNTAX:**

```
CREATE SEQUENCE sequence_name
START WITH initial_value
INCREMENT BY increment_value
MINVALUE minimum value
MAXVALUE maximum value
CYCLE|NOCYCLE ;
```

Where,
sequence_name: Name of the sequence.

initial_value: starting value from where the sequence starts.
Initial_value should be greater than or equal to minimum value and less than equal to maximum value.

increment_value: Value by which sequence will increment itself.
Increment_value can be positive or negative.

minimum_value: Minimum value of the sequence.
maximum_value: Maximum value of the sequence.

cycle: When sequence reaches its set_limit
it starts from beginning.

nocycle: An exception will be thrown
if sequence exceeds its max_value.

- **EXAMPLE-1:**

Following is the sequence query creating sequence in ascending order.

```
CREATE SEQUENCE sequence_1
start with 1
increment by 1
minvalue 0
maxvalue 100
cycle;
```

- Above query will create a sequence named *sequence_1*.Sequence will start from 1 and will be incremented by 1 having maximum value 100. Sequence will repeat itself from start value after exceeding 100.

- **EXAMPLE-2:**

Following is the sequence query creating sequence in descending order.

```
CREATE SEQUENCE sequence_2
start with 100
increment by -1
minvalue 1
maxvalue 100
cycle;
```

- Above query will create a sequence named *sequence_2*.Sequence will start from 100 and should be less than or equal to maximum value and will be incremented by -1 having minimum value 1.

- **Example to use sequence:**
- create a table named students with columns as id and name.

```
CREATE TABLE students
(
ID number(10),
NAME char(20)
);
```

- Now insert values into table.

```
INSERT into students VALUES(sequence_1.nextval,'John');
INSERT into students VALUES(sequence_1.nextval,'Mary');
```

- **Output:**

```
________________________
| ID  |      NAME       |
------------------------
|  1  |     John      |
|  2  |     Mary      |
 ----------------------
```

5.5 VIEWS :

Views in SQL are kind of virtual tables. A view also has rows and columns as they are in a real table in the database. We can create a view by selecting fields from one or more tables present in the database. A View can either have all the rows of a table or specific rows based on certain condition.

Uses of a View :

A good database should contain views due to the given reasons:

1. **Restricting data access –**
   Views provide an additional level of table security by restricting access to a predetermined set of rows and columns of a table.
2. **Hiding data complexity –**
   A view can hide the complexity that exists in a multiple table join.
3. **Simplify commands for the user –**
   Views allows the user to select information from multiple tables without requiring the users to actually know how to perform a join.
4. **Store complex queries –**
   Views can be used to store complex queries.
5. **Rename Columns –**
   Views can also be used to rename the columns without affecting the base tables provided the number of columns in view must match the number of

columns specified in select statement. Thus, renaming helps to to hide the names of the columns of the base tables.

6. **Multiple view facility –**
Different views can be created on the same table for different users.

Views offer the following advantages:

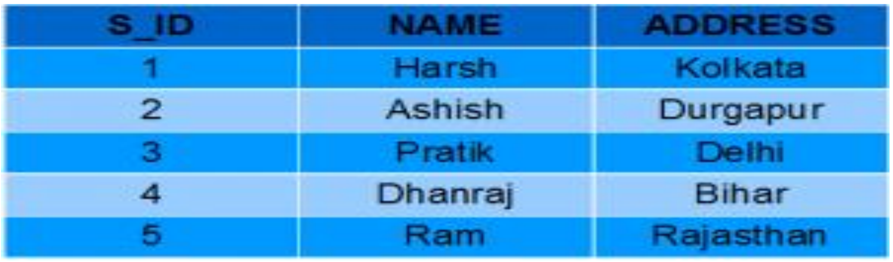
1. Ease of use: A view hides the complexity of the database tables from end users. Essentially we can think of views as a layer of abstraction on top of the database tables.

2. Space savings: Views takes very little space to store, since they do not store actual data.

3. Additional data security: Views can include only certain columns in the table so that only the non-sensitive columns are included and exposed to the end user. In addition, some databases allow views to have different security settings, thus hiding sensitive data from prying eyes.

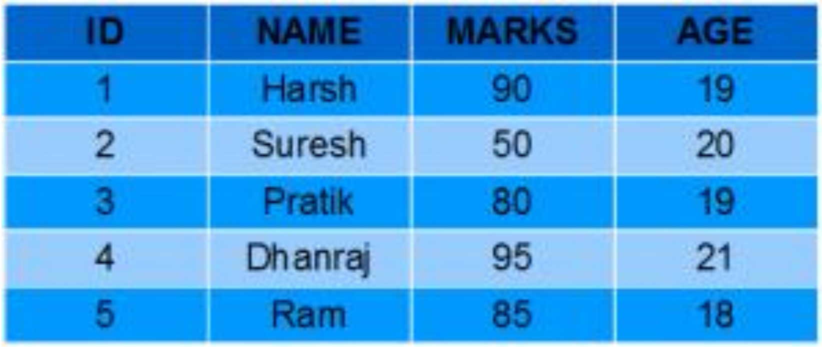
- **Sample Tables**:

StudentDetails



S_ID	NAME	ADDRESS
1	Harsh	Kolkata
2	Ashish	Durgapur
3	Pratik	Delhi
4	Dhanraj	Bihar
5	Ram	Rajasthan

StudentMarks



ID	NAME	MARKS	AGE
1	Harsh	90	19
2	Suresh	50	20
3	Pratik	80	19
4	Dhanraj	95	21
5	Ram	85	18

5.5.1 CREATING SIMPLE VIEW, UPDATING VIEW, DROPPING VIEW.

❖ ***CREATING SIMPLE VIEW:***

You can create View using **CREATE VIEW** statement. A View can be created from a single table or multiple tables.

➢ **Syntax:**

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

Where,view_name: Name for the View

table_name: Name of the table

condition: Condition to select rows

➢ **Example:**

Creating View from a single table:

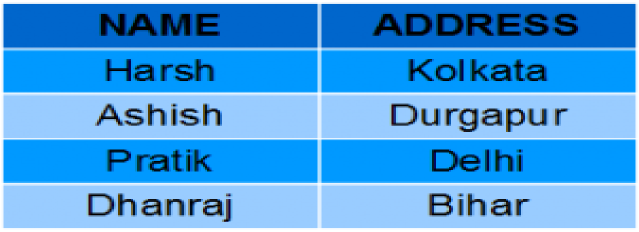
- In this example we will create a View named DetailsView from the table StudentDetails.
- **Query:**

```
CREATE VIEW DetailsView AS
SELECT NAME, ADDRESS
FROM StudentDetails
WHERE S_ID < 5;
```

- To see the data in the View, we can query the view in the same manner as we query a table.

```
SELECT * FROM DetailsView;
```

Output:



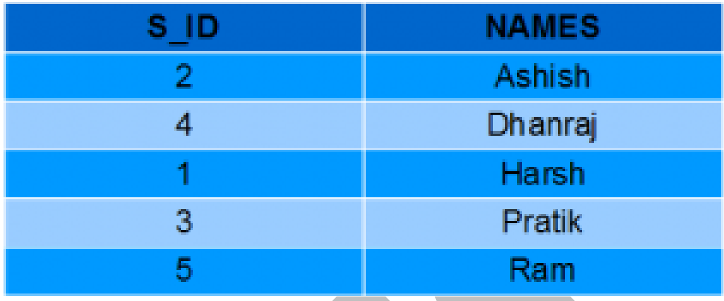
NAME	ADDRESS
Harsh	Kolkata
Ashish	Durgapur
Pratik	Delhi
Dhanraj	Bihar

- In this example, we will create a view named StudentNames from the table StudentDetails.
- **Query:**
  ```
  CREATE VIEW StudentNames AS
  SELECT S_ID, NAME
  FROM StudentDetails
  ORDER BY NAME;
  ```

- If we now query the view as,

  ```
  SELECT * FROM StudentNames;
  ```

Output:



S_ID	NAMES
2	Ashish
4	Dhanraj
1	Harsh
3	Pratik
5	Ram

Creating View from multiple tables:

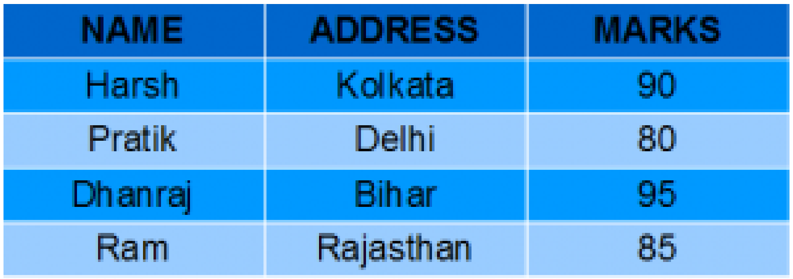
- In this example we will create a View named MarksView from two tables StudentDetails and StudentMarks. To create a View from multiple tables we can simply include multiple tables in the SELECT statement.

- **Query:**
  ```
  CREATE VIEW MarksView AS
  SELECT StudentDetails.NAME, StudentDetails.ADDRESS,
  StudentMarks.MARKS
  FROM StudentDetails, StudentMarks
  WHERE StudentDetails.NAME = StudentMarks.NAME;
  ```

- To display data of View MarksView:

  ```
  SELECT * FROM MarksView;
  ```

Output:



NAME	ADDRESS	MARKS
Harsh	Kolkata	90
Pratik	Delhi	80
Dhanraj	Bihar	95
Ram	Rajasthan	85

❖ ***UPDATING VIEW:***

There are certain conditions needed to be satisfied to update a view. If any one of these conditions is **not** met, then we will not be allowed to update the view.

1. The SELECT statement which is used to create the view should not include GROUP BY clause or ORDER BY clause.
2. The SELECT statement should not have the DISTINCT keyword.
3. The View should have all NOT NULL values.
4. The view should not be created using nested queries or complex queries.
5. The view should be created from a single table. If the view is created using multiple tables then we will not be allowed to update the view.

- We can use the **CREATE OR REPLACE VIEW** statement to add or remove fields from a view.

➢ **Syntax:**

```
CREATE OR REPLACE VIEW view_name AS
SELECT column1,coulmn2,..
FROM table_name
WHERE condition;
```

➢ **Example:**

- if we want to update the view **MarksView** and add the field AGE to this View from **StudentMarks** Table,

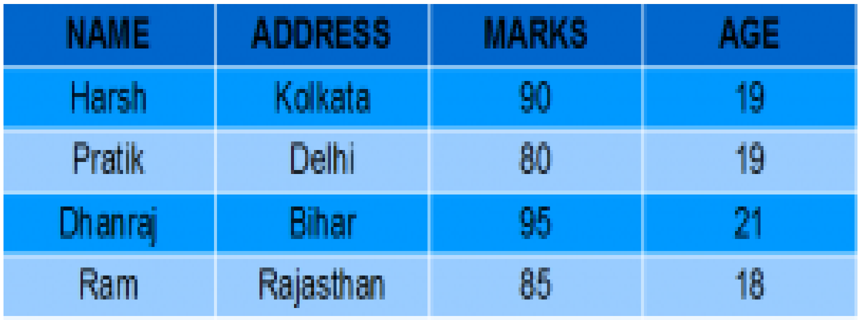
```
CREATE OR REPLACE VIEW MarksView AS
SELECT StudentDetails.NAME, StudentDetails.ADDRESS,
StudentMarks.MARKS, StudentMarks.AGE
FROM StudentDetails, StudentMarks
```

```
WHERE StudentDetails.NAME = StudentMarks.NAME;
```

- If we fetch all the data from MarksView now as:

```
SELECT * FROM MarksView;
```

Output:



NAME	ADDRESS	MARKS	AGE
Harsh	Kolkata	90	19
Pratik	Delhi	80	19
Dhanraj	Bihar	95	21
Ram	Rajasthan	85	18

Inserting a row in a view:

We can insert a row in a View in a same way as we do in a table. We can use the INSERT INTO statement of SQL to insert a row in a View.

- **Syntax:**

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

- **Example**:

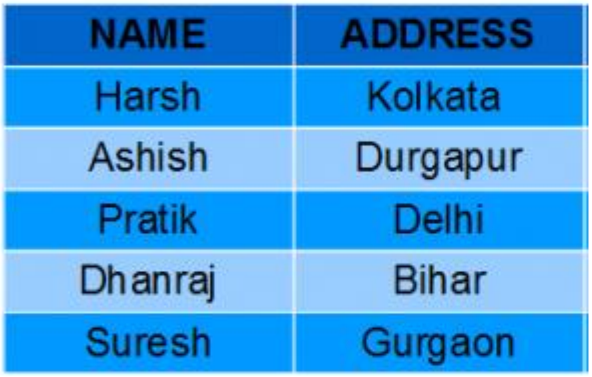
- In the below example we will insert a new row in the View DetailsView which we have created above in the example of “creating views from a single table”.

```
INSERT INTO DetailsView(NAME, ADDRESS)
VALUES("Suresh","Gurgaon");
```

- If we fetch all the data from DetailsView now as,

```
SELECT * FROM DetailsView;
```

Output:



NAME	ADDRESS
Harsh	Kolkata
Ashish	Durgapur
Pratik	Delhi
Dhanraj	Bihar
Suresh	Gurgaon

- ***DROPPING VIEW:***

We have learned about creating a View, but what if a created View is not needed any more? SQL allows us to delete an existing View. We can delete or drop a View using the DROP statement.

- **Syntax:**

```
DROP VIEW view_name;
```

- **Example**:
  if we want to delete the View **MarksView**,

```
DROP VIEW MarksView;
```

Deleting a row from a View:
Deleting rows from a view is also as simple as deleting rows from a table. We can use the DELETE statement of SQL to delete rows from a view. Also deleting a row from a view first delete the row from the actual table and the change is then reflected in the view.

- **Syntax:**

```
DELETE FROM view_name
WHERE condition;
```

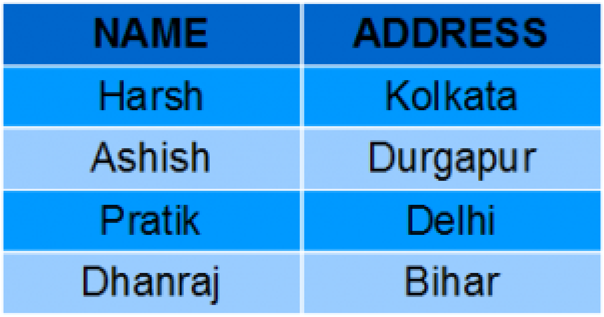
- **Example:**
- In this example we will delete the last row from the view DetailsView which we just added in the above example of inserting rows.

```
DELETE FROM DetailsView
WHERE NAME="Suresh";
```

- If we fetch all the data from DetailsView now as,

```
SELECT * FROM DetailsView;
```

Output:



NAME	ADDRESS
Harsh	Kolkata
Ashish	Durgapur
Pratik	Delhi
Dhanraj	Bihar

5.5.2 DIFFERENCE BETWEEN VIEW AND TABLE.

View	**Table**
The view is treated as a virtual table that is extracted from a database.	The table is structured with a set number of columns and a boundless number of columns
The table is database's which are utilized to hold the information that is utilized in applications and reports.	A view is additionally a database object which is utilized as a table and inquiry that can be connected to different tables.
The view is utilized to query certain information which is contained in a few distinct tables	The table holds fundamental client information and holds cases of a characterized object.
In the view, you will get frequently queried information.	In the table, changing the information in the database likewise changes the information appeared in the view which isn't the