**What is a View in SQL?**

A **View** is a **virtual table** based on the result of a **SELECT query**. It does **not store data itself**, but dynamically fetches data from the underlying tables when queried.

**✅ Purpose of Views**

* To **simplify complex queries**.
* To **restrict data access** (security).
* To **present data in a specific format**.
* To **encapsulate business logic**.

### ****Example 1: Simple View****

Create a view to show employees with salary > 50000

CREATE VIEW HighSalaryEmployees AS

SELECT name, salary

FROM employees

WHERE salary > 50000;

Query the view:

SELECT \* FROM HighSalaryEmployees;

**Output:**

| **name** | **salary** |
| --- | --- |
| Alice | 60000 |

To **perform INSERT, UPDATE, DELETE** via a view, the view must be:

* Based on a **single table**.
* Should **NOT** have:
  + JOINs
  + GROUP BY
  + Aggregations
  + DISTINCT
  + UNION / UNION ALL

### ****Step 1: Create a Simple View****

sql

CopyEdit

CREATE VIEW SimpleEmployeeView AS

SELECT emp\_id, name, salary, dept\_id

FROM employees;

### ✅ ****1. INSERT via View****

sql

CopyEdit

INSERT INTO SimpleEmployeeView (emp\_id, name, salary, dept\_id)

VALUES (4, 'Carol', 55000, 20);

**Effect:** Adds a new row to the **employees** table via the view.

### ✅ ****2. UPDATE via View****

sql

CopyEdit

UPDATE SimpleEmployeeView

SET salary = 65000

WHERE emp\_id = 1;

**Effect:** Updates salary of **John** to **65000** in the base table.

### ✅ ****3. DELETE via View****

sql

CopyEdit

DELETE FROM SimpleEmployeeView

WHERE emp\_id = 3;

**Effect:** Deletes the employee **Bob** (emp\_id = 3) from the base **employees** table.

**What is an Index in SQL?**

An **Index** in SQL is a **performance optimization** feature that speeds up the **retrieval of data** from a table.

* It works like the **index of a book**, helping the database engine **locate data faster**.
* Without an index, the database performs a **full table scan** to find data — slow for large tables.

**✅ Why Use Indexes?**

* To **speed up SELECT queries**.
* To improve **JOIN performance**.
* To optimize **WHERE, ORDER BY, GROUP BY** operations.
* To enforce **uniqueness** (via **unique indexes**

| **Index Type** | **Description** | **Example Usage** |
| --- | --- | --- |
| **Single-Column Index** | Index on one column | salary, name |
| **Composite Index** | Index on multiple columns | first\_name, last\_name |
| **Unique Index** | Ensures all values in a column are unique | For email, user\_id |
| **Clustered Index** | Sorts the data physically in table based on index | Default in **SQL Server** (Primary Key) |
| **Non-Clustered Index** | Creates a separate structure to reference data | Can have multiple per table |
| **Full-Text Index** | Used for searching within text data | Articles, product descriptions |
| **Bitmap Index** | Efficient for columns with **few distinct values** | Gender, Boolean fields (Oracle, Data Warehousing) |

## **Advantages of Indexes**

* **Faster data retrieval**.
* Improve **performance of JOINs**.
* Help in **WHERE, ORDER BY, GROUP BY** queries.
* Enforce **uniqueness** using Unique Indexes.

## ✅ **Disadvantages of Indexes**

| **Issue** | **Impact** |
| --- | --- |
| **Slower write operations** | INSERT, UPDATE, DELETE become slower because indexes must be updated. |
| **More storage** | Indexes consume additional disk space. |
| **Maintenance overhead** | Indexes need to be rebuilt/maintained periodically, especially after heavy updates. |

## ✅ **When to Use Indexes**

* On columns used frequently in:
  + **WHERE conditions**
  + **JOIN conditions**
  + **ORDER BY or GROUP BY**
* On columns that are **searched frequently**.

## ✅ **When NOT to Use Indexes**

* On **small tables** — scanning is already fast.
* On columns that are **frequently updated**.
* On columns with **high number of duplicate values**, unless using a **bitmap index**.

## ✅ **How Index Works (Conceptually)**

Most relational databases implement indexes using **B-Trees** or **Hash indexes**:

* **B-Tree**: Balanced tree that keeps data sorted for fast retrieval.
* **Hash index**: Fast lookup for equality checks (== operator).

## ✅ **Syntax of CASE Statement**

### ✅ ****1. Simple CASE****

sql

CopyEdit

CASE column\_name

WHEN value1 THEN result1

WHEN value2 THEN result2

...

ELSE default\_result

END

### ✅ ****2. Searched CASE****

sql

CopyEdit

CASE

WHEN condition1 THEN result1

WHEN condition2 THEN result2

...

ELSE default\_result

END

## ✅ **Example Data**

| **emp\_id** | **name** | **salary** |
| --- | --- | --- |
| 1 | John | 70000 |
| 2 | Alice | 50000 |
| 3 | Bob | 30000 |

## ✅ **Example 1: Simple CASE**

Categorizing employees by salary:

sql

CopyEdit

SELECT name, salary,

CASE salary

WHEN 70000 THEN 'High'

WHEN 50000 THEN 'Medium'

ELSE 'Low'

END AS salary\_category

FROM employees;

### ✅ Output:

| **name** | **salary** | **salary\_category** |
| --- | --- | --- |
| John | 70000 | High |
| Alice | 50000 | Medium |
| Bob | 30000 | Low |

## ✅ **Example 2: Searched CASE**

sql

CopyEdit

SELECT name, salary,

CASE

WHEN salary >= 60000 THEN 'High'

WHEN salary >= 40000 THEN 'Medium'

ELSE 'Low'

END AS salary\_category

FROM employees;

## ✅ **Example 3: CASE in ORDER BY**

Order employees based on salary category:

sql

CopyEdit

SELECT name, salary

FROM employees

ORDER BY

CASE

WHEN salary >= 60000 THEN 1

WHEN salary >= 40000 THEN 2

ELSE 3

END;

## ✅ **Example 4: CASE in UPDATE**

Update employees' bonus based on salary:

sql

CopyEdit

UPDATE employees

SET bonus =

CASE

WHEN salary >= 60000 THEN 5000

WHEN salary >= 40000 THEN 3000

ELSE 1000

END;

## ✅ **Key Points About CASE**

* Works in SELECT, UPDATE, ORDER BY, and even in WHERE.
* Always ends with an **END** keyword.
* ELSE is **optional** — if omitted, unmatched cases return **NULL**.