SQL, View, Index, Merge Statment, Pivot tables (Day 8)

ndexes In SQL Server

In SQL Server, data is stored in units called **data pages**, which are physical storage units within files (e.g., `.mdf`, `.ndf`). The structure and indexing of a table affect how data is stored and retrieved, influencing the efficiency of searches and inserts.

1. Tables Without a Primary Key (Heap)

- **Heap Storage**:
  - When a table does not have a **Primary Key (PK)**, it is stored as a **heap** (unordered on the disk).
  - This unordered storage results in a **table scan** when querying data since there is no defined order to guide the search.
- **Table Scans**:
  - A table scan means the database engine must check each row in the table to find matching data, which can be slow on large tables.

- **Insert Operations**:
  - Inserts into a heap are fast because each new row is simply added to the end of the table.

2. Tables With a Primary Key (Clustered Index)

- **Clustered Index**:
  - When a table has a **Primary Key**, a **clustered index** is created on that key. The data is physically sorted on the disk based on the PK, making searches faster.
  - The clustered index organizes the data pages as a **binary search tree** where each node holds a range of values, allowing efficient data retrieval.
- **Search Efficiency**:
  - Searches using the clustered index are more efficient as the data is sorted, allowing the database to use a **binary search** method to quickly locate the requested rows.
  - This structure avoids the need for a full table scan.
- **Insert Operations**:
  - Inserts can be slower in a table with a clustered index because SQL Server must maintain the sorted order, sometimes causing data to be reorganized.

How the Clustered Index Works

- **Sorted Data**:
  - Data is physically sorted based on the primary key, which means rows are stored in order on the disk.
- **Binary Search Tree Structure**:
  - The clustered index is structured as a binary search tree, which allows efficient lookups by guiding the search through nodes until the matching data page is located.

Summary

- **Heap (No PK)**:
  - Unordered data, uses table scans, fast inserts.

- **Clustered Index (With PK)**:
    - Ordered by primary key, enables fast searches using a binary search, can slow down inserts.

Non-Clustered Indexes

When you need to search a column that is not the **Primary Key** (PK) or clustered index, SQL Server would otherwise perform a **table scan**, which checks each row, resulting in slower performance. To optimize searches on non-PK columns, **non-clustered indexes** are used.

Characteristics of Non-Clustered Indexes

- **Single Column Sorting**:
    - Non-clustered indexes are typically sorted by a single column that you specify, unlike clustered indexes, which are based on the PK.
- **Separate Data Pages**:
    - A non-clustered index creates a separate set of **data pages** focused on the indexed column, organizing them in a sorted structure.
- **Logical Pointers**:
    - Each entry in the non-clustered index contains pointers (row locators) that reference the original data pages for the full row data.
- **Storage**:
    - Non-clustered indexes exist separately from the main data pages and are only loaded into memory when needed.

Search Process Using a Non-Clustered Index

- When a search condition references a column with a non-clustered index (e.g., **EmployeeName**), the following process is used:
  1. **Index Check**:
     - SQL Server checks if the condition applies to a column with a non-clustered index.
  2. **Binary Search Tree**:
     - The index pages are structured in a **binary search tree** where each node contains sorted values, helping SQL Server quickly locate the target.
  3. **Pointer Retrieval**:
     - Once the target value is found in the non-clustered index, a pointer retrieves the row data from the original data pages.
  4. **Return All Matches**:
     - If multiple rows match (for instance, multiple employees named **Omar**), the index returns pointers to all matching rows.

Non-Clustered Index Performance

- **Slower than Clustered Index**:
  - While non-clustered indexes improve search times, they are still slower than clustered indexes because they require one additional step: retrieving the full data from the main table using pointers after locating the value in the index.
- **Multiple Non-Clustered Indexes Allowed**:
  - A table can have multiple non-clustered indexes, optimizing searches on various columns. However, excessive indexing can slow down inserts and updates, as these indexes need updating with each modification.

Example of Non-Clustered Index Creation

```
-- Creating a non-clustered index on the EmployeeName column CREATE NONCLUSTE
RED INDEX idx_EmployeeName ON Employees(EmployeeName);
```

Using a non-clustered index, searches on **EmployeeName** will be faster, as SQL Server can locate values within the index rather than performing a full table scan. This index stores **EmployeeName** in a sorted structure, while keeping pointers to the complete row data in the original data pages, enhancing retrieval efficiency.

Steps of a Non-Clustered Index Search

1. **Check Condition in `SELECT` Statement**:
   - When you run a query like `SELECT * FROM Employees WHERE Name = 'Omar'`, SQL Server recognizes that `Name` isn't the primary key, so it would typically require a table scan if no index exists.
2. **Navigate to Root of the Index Tree**:
   - SQL Server starts at the root of the non-clustered index (organized as a binary search tree) and begins searching for the target value (`'Omar'`). This tree structure allows for quick comparisons and efficient searching.
3. **Compare by ASCII Code**:
   - To locate `'Omar'`, SQL Server compares ASCII values at each node in the tree, following the sorted order of `Name`. This enables a fast, direct search path to the desired index entry.
4. **Locate Target in Data Pages**:
   - Once SQL Server reaches the leaf node (the bottom level of the index tree) that matches `'Omar'`, it retrieves the index entry, which contains a **pointer** to the original data page row.
5. **Access Full Data Using Pointer**:
   - Using the pointer, SQL Server goes to the original data page stored on the hard disk, where it retrieves all the columns for the matching row(s) with `Name = 'Omar'`.
6. **Handle Multiple Matching Rows**:
   - If there are multiple rows with the name `'Omar'`, SQL Server collects all pointers associated with each `'Omar'` entry in the non-clustered index, retrieves each row, and returns them all in the result set.

Clustered and Non-Clustered Index Limitations

- **Clustered Index**: Each table can only have one clustered index. This is because a clustered index determines the physical order of data in the table. When a primary key (PK) is defined, SQL Server automatically creates a clustered index on that PK by default.
    - For example, if the `Employee` table has a primary key column (`SSN`), it already has a clustered index on `SSN`.
- **Non-Clustered Index**: Tables can have multiple non-clustered indexes, each created on different columns to speed up searches on those columns.
    - If you try to create a second clustered index (e.g., `CREATE CLUSTERED INDEX myindex ON Employee(Fname)`), SQL Server will throw an error because `Employee` already has a clustered index on `SSN`.
    - Instead, you can create a non-clustered index on columns like `Fname` or `Lname`, which helps optimize searches for those columns without altering the primary data order in the table.

Creating Non-Clustered Indexes

- **Example**:
    - `CREATE NONCLUSTERED INDEX myindex ON Employee(Fname)` – This creates a non-clustered index on `Fname`, which can optimize searches based on the `Fname` column.
    - Similarly, you could create another non-clustered index, e.g., `CREATE NONCLUSTERED INDEX myindex1 ON Employee(Lname)`.
- After creating these indexes, you can view them by going to the table in your SQL Server Management Studio:
    - Navigate to the table ➔ expand `+` ➔ go to `Indexes`, where you will see both `myindex` and `myindex1` listed as non-clustered indexes created on `Fname` and `Lname`, respectively.

Automatic Index Creation

- **Primary Key Constraint**: When a primary key is defined on a column, SQL Server automatically creates a clustered index on that column (if there isn't already a clustered index on the table).
    - **Reason**: A primary key requires unique and sorted values, and the clustered index ensures efficient sorting and fast access.
- **Unique Constraint**: Defining a unique constraint on a column automatically creates a non-clustered index on that column.
    - **Reason**: The unique constraint ensures that all values in the column are unique, and a non-clustered index enables this check efficiently without affecting the data's physical order on disk.

Example Code

```
CREATE TABLE Student ( ID INT PRIMARY KEY, Name VARCHAR(50), Age INT UNIQUE
); CREATE UNIQUE INDEX indx ON Student(Age);
```

- In this example:
    - `id` is a primary key, so SQL Server will create a **clustered index** on `id`.
    - `age` is a unique column, so SQL Server will create a **non-clustered index** on `age`.

After creating this table, you can verify the indexes in SQL Server Management Studio:

- Navigate to `Student` ➔ expand `+` ➔ go to `Indexes`.
- Here, you'll see the clustered index on `id` and the non-clustered index on `age`.

Feature	Clustered Index	Non-Clustered Index
Definition	A clustered index sorts and stores the rows of the table on disk based on the indexed column(s). The actual data rows are arranged in the same order as the index.	A non-clustered index creates a separate structure from the actual data, storing pointers to the actual rows in the data pages.
Number Allowed per Table	Only **one** clustered index per table (as it defines the physical order of data).	Multiple non-clustered indexes are allowed (up to 999 in SQL Server).
Primary Purpose	Primarily improves performance for search queries based on primary key or frequently used columns, ensuring fast data retrieval.	Optimizes queries that involve searches, sorting, or filtering based on non-primary key columns.
Physical Storage	Stores data in a **B-tree** structure where leaf nodes contain actual data rows, ordered by the indexed column(s).	Also stored in a **B-tree**, but leaf nodes contain pointers (row identifiers or primary keys) to data rows, rather than the actual data.
Default Index for Primary Key	Automatically created if a **Primary Key** constraint is defined.	Automatically created when a **Unique Constraint** is defined.
Data Retrieval	Faster retrieval for sorted and range queries, as data is already organized. Ideal for `ORDER BY` queries and primary key lookups.	Slightly slower than clustered indexes due to an extra step to follow pointers to actual data.
Data Update Implications	Slower for `INSERT`, `UPDATE`, and `DELETE` on indexed columns, as it may require reordering of data.	Minimal impact on data insertion, update, and deletion; only the index itself needs to be updated, not the physical order of data.
Disk Space Usage	Requires no extra space as it organizes data rows directly in the table.	Requires additional disk space to store the index and pointers, especially if the table has multiple non-clustered indexes.
Storage Location	Stored on disk with the table data in the same physical order, meaning the table is stored by this index.	Stored separately from the table data with pointers back to the original table rows.
Best Use Cases	Suitable for primary key, frequently searched columns, and range	Ideal for columns frequently used in search, filter, or sort operations, but

	queries. Useful for tables with high read-to-write ratios.	not as the primary means of accessing rows.
Range Query Performance	Performs well on range queries (e.g., `BETWEEN`, `<`, `>`, `ORDER BY`) due to sorted data.	Slower for range queries as it involves accessing the table data through pointers.
Impact on Table Scans	Reduces table scan necessity, as data is sorted; table scan becomes index scan.	May still require a table scan if not covering the column being searched.
Access Path	Direct access to data rows due to B-tree structure with actual data at leaf level.	Access requires traversal of the B-tree to reach pointers, which then reference actual rows in the table.
Maintenance Complexity	Higher maintenance due to the need to maintain data ordering on updates.	Lower maintenance but may require frequent rebuilds for optimization on heavily updated columns.

Key Differences Summary:

- **Single Clustered Index**: A table can have only one clustered index since it determines the physical data order.
- **Multiple Non-Clustered Indexes**: You can have multiple non-clustered indexes on a table to optimize various queries.
- **Direct Data Access**: Clustered indexes access data directly, while non-clustered indexes use pointers.

Example Use Cases:

- **Clustered Index**: Best for unique, primary key columns or frequently searched columns that often appear in range-based queries.
- **Non-Clustered Index**: Ideal for columns frequently filtered or sorted but not the primary access path, such as secondary keys and columns used in joins or `WHERE` conditions.

To determine which columns are used most frequently and should be indexed, SQL Server provides two valuable tools: **SQL Server Profiler** and **SQL Server Database Engine Tuning Advisor**.

1. SQL Server Profiler

SQL Server Profiler is a tool that captures and logs SQL Server events, helping you see which queries are frequently executed.

Steps to Use SQL Server Profiler:

1. **Open SQL Server Profiler**: Search for "SQL Server Profiler" on your laptop and open it.
2. **New Trace**: Click on "File" > "New Trace" to start capturing data.
3. **Save Trace**: Before starting the trace, save it as a `.trc` file. This will allow you to use the trace file with the Database Engine Tuning Advisor.
4. **Run Queries**: Run the queries you want to analyze. For example, run `SELECT * FROM Employee WHERE Sex='M'` multiple times in SQL Server Management Studio.
5. **View Profiler Output**: SQL Server Profiler will capture these queries, but it may become cluttered with too many entries. To optimize indexing recommendations without sorting through all these queries, use the **SQL Server Tuning Advisor**.

---

2. SQL Server Database Engine Tuning Advisor

The Database Engine Tuning Advisor helps analyze SQL Server workloads and provides index recommendations based on the data in the trace file.

Steps to Use Database Engine Tuning Advisor:

1. **Open Tuning Advisor**: Search for "Database Engine Tuning Advisor" on your laptop and open it.
2. **Load Trace File**: Choose the `.trc` file you saved from SQL Server Profiler.
3. **Select Database and Tables**: Choose the relevant database(s) and tables you want to analyze for tuning.
4. **Start Analysis**: Click "Start Analysis" from the top bar. The tool will analyze the workload captured in the trace file.
5. **Review Recommendations**: After analysis, it will suggest columns to index. You can implement these suggestions to optimize query performance.

In SQL Server, under the **System Databases** section, you'll find four essential databases that serve different purposes for managing server configurations, templates, tracking, and temporary storage:

1. Master Database

- **Purpose**: Holds the server's metadata and critical configurations.
- **Contents**: Includes usernames, passwords, details of all databases, file locations, and server-level settings.
- **Significance**: Essential for server startup; if corrupted, the server may not start correctly.

2. Model Database

- **Purpose**: Acts as a template for creating new databases.
- **Contents**: Any configurations or objects (like tables or views) you want to replicate in new databases can be created here.
- **Usage**: When a new database is created, it inherits settings and objects from the model database. For example, creating a table in the model database ensures that all future databases will contain that table by default.

3. MSDB Database

- **Purpose**: Used by SQL Server Agent for scheduling and automation.
- **Functionality**: Tracks jobs and tasks with scheduled times, such as automated backups, alerts, and user activity thresholds.
- **Example**: Schedule a daily backup at a specified time or trigger an action when a certain user count is reached.

4. TempDB Database

- **Purpose**: Stores temporary data created during query processing.
- **Usage**:
    - Temporarily stores intermediate results in complex queries (like subqueries).
    - Holds temporary tables or objects used during the session.
    - Once the session or query completes, TempDB automatically clears out these temporary objects.

In SQL Server, there are two types of temporary tables that serve different purposes based on their scope and lifetime: **local temporary tables** and **global temporary tables**.

A. Local Temporary Tables

- **Syntax**: Created using a single `#` before the table name.

```
USE CompanyDB; CREATE TABLE #exam ( id INT );
```

- **Scope**:
  - Session-based tables; the table is only available within the session (or query) that created it.
  - You will **not** find the `#exam` table in the `Company_SD` database; instead, it is stored in the `tempdb` database.
- **Access**:
  - If you open a new query window, you **cannot** access the `#exam` table created in a different session.
  - Creating a new `#exam` table in a new session is allowed; it will create a separate instance of the table.
- **Lifetime**:
  - The table is automatically dropped when the session that created it is closed or when the query is canceled.

B. Global Temporary Tables

- **Syntax**: Created using double `##` before the table name.

```
USE CompanyDB; CREATE TABLE ##exam ( id INT );
```

- **Scope**:
  - Shared across all sessions; the table is accessible to any user or session that connects to the SQL Server instance.
  - Similar to local temporary tables, you will find the `##exam` table in `tempdb`.

- **Access**:
  - When you open a new query window, you **can** access the `##exam` table created in another session.
  - If you attempt to create another `##exam` table in a new session, it will result in an error because global temporary tables must have unique names across all sessions.
- **Lifetime**:
  - The table is dropped automatically when all sessions referencing it are closed or when all queries connected to the server are canceled.

Summary of Differences

Feature	Local Temporary Table (`#`)	Global Temporary Table (`##`)
Scope	Session-based	Shared across all sessions
Access	Not accessible in new sessions	Accessible in any session
Lifetime	Dropped when the session ends	Dropped when no sessions reference it
Name Conflicts	Multiple instances allowed	Must have unique names across sessions
Storage Location	`tempdb`	`tempdb`

Practical Use Cases

- **Local Temporary Tables**: Useful for storing intermediate results or calculations that are only needed for a single session.
- **Global Temporary Tables**: Ideal for scenarios where multiple sessions need to share data, like temporary staging tables for data processing tasks.

Understanding the differences between these types of temporary tables can help you design more efficient and effective database operations based on your application's needs.

ROLLUP, **CUBE**, **GROUPING SETS**, **PIVOT**, and **UNPIVOT** in SQL

1. ROLLUP

The `ROLLUP` operator is used to create subtotals and grand totals for a specified group of columns. It provides a hierarchical grouping.

Example:

```
SELECT Sex, SUM(Salary) AS TotalSalary FROM Employees GROUP BY ROLLUP(Sex);
```

- This query will return:
  - Total salary for each sex (M and F)
  - A grand total of all salaries.

You can also use `ROLLUP` with multiple columns:

```
SELECT Fname, Sex, SUM(Salary) FROM Employees GROUP BY ROLLUP(Fname, Sex);
```

- This will return:
  - The sum of salaries for each `Fname` and `Sex`.
  - The total sum for each `Fname` (regardless of `Sex`).
  - A grand total sum of all salaries.

2. CUBE

The `CUBE` operator generates a result set that represents all possible combinations of the specified columns. It is more comprehensive than `ROLLUP`.

Example:

```
SELECT Fname, Sex, SUM(Salary) FROM Employee GROUP BY CUBE(Fname, Sex);
```

- This query will return:
  - The sum of salaries for each `Fname` and `Sex`.
  - The sum for each `Sex` regardless of `Fname`.
  - The sum for each `Fname` regardless of `Sex`.
  - A grand total of all salaries.

3. GROUPING SETS

`GROUPING SETS` allows you to define multiple groupings in a single query, giving you the flexibility to specify exactly what groupings you want.

Example:

```
SELECT Fname, Sex, SUM(Salary) AS TotalSalary FROM Employees GROUP BY GROUPIN
G SETS((Fname), (Sex));
```

- This query will return:
  - Total salaries grouped by `Fname`.
  - Total salaries grouped by `Sex`.
  - It will not provide a grand total as `ROLLUP` or `CUBE` would.

4. PIVOT

`PIVOT` allows you to convert rows into columns, which is useful for transforming data to make it more readable or analyzable.

Example:

```
SELECT * FROM Sales PIVOT ( SUM(Quantity) FOR Salesman IN ([Ahmed], [Ali]) )
AS PivotTable;
```

- This query transforms the `Sales` data by summing up the `Quantity` sold by each `Salesman`, creating columns for `Ahmed` and `Ali`.

5. UNPIVOT

`UNPIVOT` does the reverse of `PIVOT`, converting columns back into rows.

Example:

```
SELECT Salesman, Quantity FROM Sales UNPIVOT ( Quantity FOR Sales IN ([Ahme
d], [Ali]) ) AS UnpivotTable;
```

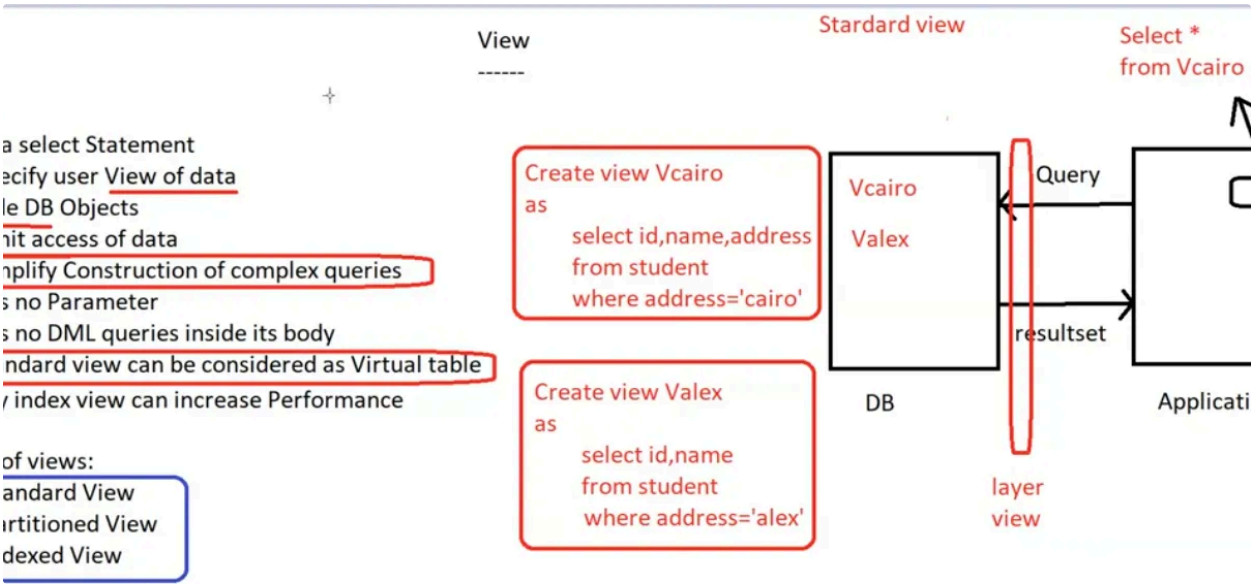
- This query will take columns `Ahmed` and `Ali` from the `Sales` table and turn them into rows under a single `Salesman` column, with their corresponding quantities.

Summary of Key Differences

Feature	ROLLUP	CUBE	GROUPING SETS	PIV
Purpose	Hierarchical subtotals	All combinations of groupings	Custom-defined groupings	Co
Output	Subtotals and grand totals	All possible combinations	Specific groupings	Su
Complexity	Less complex	More complex	Moderate complexity	Tra

View

A **View** in SQL is essentially a virtual table that allows users to access specific parts of data securely and efficiently (View is a select statment).



Explanation of Views

1. **Security**:
   - By creating a view, you can restrict users' access to only specific data within a table.
   - In your example, the `Vcairo` view only shows students from Cairo, meaning users won't see the entire `student` table. This provides an additional layer of security by not exposing all columns or rows directly.
2. **Example**:

```
CREATE VIEW Vcairo AS SELECT * FROM student WHERE address = 'Cairo';
```

   - Now, users can access data using the view:

```
SELECT * FROM Vcairo;
```

   - This command will only return records where `address = 'Cairo'`, preventing access to students from other cities.
3. **Benefits**:
   - **Data Abstraction**: Views abstract the underlying table structure, making data easier to manage and secure.
   - **Simplified Queries**: A view can simplify complex queries, allowing users to fetch data with straightforward SELECT statements.
   - **Read-only Access**: In some databases, views can be set to read-only, preventing accidental modifications to sensitive data.
4. **Use Cases**:
   - **Limit Exposure**: Expose only necessary columns or rows to users.
   - **Data Masking**: Mask certain data columns while allowing access to others.
   - **Simplify Complex Joins**: Combine data from multiple tables in a view to simplify queries for end users.

Characteristics of Views

1. **Standard View**:
   - Acts as a **virtual table** that encapsulates a `SELECT` statement.
   - Provides **data abstraction**, hiding specific database objects and limiting user access to sensitive data.
   - Simplifies **complex query construction**, as users can access predefined data sets.
   - **Cannot contain parameters** or `DML` (Data Manipulation Language) queries within its definition, but `DML` operations (like `INSERT`, `UPDATE`, `DELETE`) can be executed on views as long as they affect the underlying table.
   - Any changes made to the underlying table's data automatically affect the view.
2. **Partitioned View**:
   - Enables access to data **across multiple servers**. This type of view combines tables from different databases or servers using `UNION ALL`.
   - Example:

     ```
     CREATE VIEW Vcairo AS SELECT * FROM sohag_server.v1.dbo.student UNION
     ALL SELECT * FROM assiut_server.v2.dbo.student;
     ```

   - Useful for **scaling** and accessing distributed data from various locations.

3. **Indexed View**:
    - An **indexed view** stores data physically, similar to a table with indexes, which can **improve performance** for frequently accessed views.
    - Only indexed views can improve performance, as standard views remain virtual without physical data storage.

    An **Indexed View** is a view with a unique clustered index, which materializes the view data physically in storage, improving query performance. Indexed views are commonly used when the view involves aggregations, joins, or computations, and you need faster data retrieval.

Example: Indexed View with Aggregation

Consider a scenario where we want a view that shows the total salary by department.

```
USE CompanyDB; GO -- Create the view with aggregation CREATE VIEW VDeptSa
lary WITH SCHEMABINDING -- Required for indexed views AS SELECT Dno, COUN
T_BIG(*) AS EmployeeCount, -- COUNT_BIG required for indexed views with C
OUNT SUM(Salary) AS TotalSalary FROM dbo.Employee GROUP BY Dno; GO -- Cre
ate a clustered index on the view CREATE UNIQUE CLUSTERED INDEX IX_VDeptS
alary ON VDeptSalary (Dno); GO -- Query the indexed view SELECT * FROM VD
eptSalary;
```

Explanation of Key Points:

1. **SCHEMABINDING**: Indexed views require `SCHEMABINDING`, which prevents changes to the schema of the underlying tables (e.g., adding or removing columns) unless the view is dropped or altered.
2. **COUNT_BIG**: When using `COUNT` in an indexed view, SQL Server requires `COUNT_BIG()` instead, which returns a `BIGINT` and is compatible with indexed views.
3. **Unique Clustered Index**: Creating a unique clustered index materializes the view, allowing SQL Server to physically store the aggregated data for quick retrieval.

Benefits of Using an Indexed View:

- **Performance**: Queries that match the indexed view's structure will benefit from the precomputed and stored data, providing faster results.
- **Efficiency**: Indexed views are useful for reporting queries with aggregations or joins that would otherwise take significant processing time.

Key Points

- **Automatic Updates**: Changes to the base table's data are immediately reflected in any view that accesses that table.
- **Performance**: Indexed views can optimize performance, especially for complex queries that would otherwise need multiple joins or calculations.

By using different view types, database administrators and developers can improve data security, simplify user interactions, and potentially enhance performance based on the application's needs.

Basic View Creation

Creates a view that shows all data from the `Employee` table.

```
CREATE VIEW VEmployee AS SELECT * FROM Employees; -- Usage of the view SELECT
* FROM VEmployee; SELECT EmployeeName FROM VEmployee;
```

View with Filtering

Creates a view to filter employees by gender, displaying only male employees.

```
USE CompanyDB GO CREATE VIEW VMale AS SELECT * FROM Employees WHERE Sex =
'M'; GO -- Usage of the view SELECT Fname FROM VMale;
```

View with Aliased Columns

Creates a view with custom column aliases for enhanced readability and specificity.

```
ALTER VIEW VMale (ename1, ename2, id, dat, ad, type, sal, supid, dnum) AS SEL
ECT * FROM Employees WHERE Sex = 'M'; -- Usage with alias SELECT ename1 FROM
VMale;
```

Combining Views with `UNION ALL`

Uses multiple views within a new view, combining them using `UNION ALL`.

```
USE CompanyDB; GO CREATE VIEW Vall AS SELECT * FROM VEmployee UNION ALL SELEC
T * FROM VMale; GO -- Usage of the combined view SELECT * FROM Vall;
```

Joining Tables within a View

Creates a view that joins the `Employee` and `Departments` tables, showing each employee's department name.

```
USE CompanyDB; GO CREATE VIEW V_emp_dep AS SELECT Fname, Dname FROM Employee
e INNER JOIN Departments d ON e.Dno = d.Dnum; GO -- Usage of the joined view
SELECT * FROM V_emp_dep; SELECT v.Fname, Dname, Salary FROM V_emp_dep v INNER
JOIN Employee e ON e.Fname = v.Fname;
```

Using the `WITH ENCRYPTION` option on a view in SQL Server helps to secure its definition by preventing access to the underlying code, making it more secure. When a view is created with encryption, SQL Server hides the view's code, so anyone trying to retrieve the code using `sp_helptext` will not be able to see it.

Here's how to use `WITH ENCRYPTION`:

Example: Creating an Encrypted View

```
USE CompanyDB; GO -- Creating a view with encryption CREATE VIEW Vomar WITH E
NCRYPTION AS SELECT * FROM Employees WHERE Fname = 'omar'; GO -- Attempting t
o retrieve the view's code EXEC sp_helptext 'Vomar'; -- This will return noth
ing because the view is encrypted
```

Explanation:

1. **WITH ENCRYPTION**: Adding `WITH ENCRYPTION` prevents the view's definition from being displayed in system views or retrieved with `sp_helptext`.
2. **sp_helptext**: Normally, this command would show the T-SQL code for the view. However, if the view is encrypted, `sp_helptext`s returns no results, thus hiding the view's structure.

Important Notes:

- **Irreversible**: Once a view is encrypted, the definition cannot be retrieved through regular queries. Be sure to save the code externally before encrypting.
- **Security**: Encrypted views add an additional layer of security, making it harder for unauthorized users to view sensitive logic embedded in the view.

In SQL Server, you can perform DML (Data Manipulation Language) operations—like `INSERT`, `UPDATE`, and `DELETE`—on a view after it has been created, but there are specific rules depending on the type of view and the tables it references.

1. DML on a View with Data from a Single Table

- **INSERT**: If the view is based on a single table, you can insert new rows, and these rows will be added to both the view and the underlying table.

  Example:

  ```
  INSERT INTO VMale VALUES ('mostafa', 'nasser', 112299, '2002-07-09', 'Ass
  uit', 'M', 1211, 112233, 10);
  ```

  - This new row will appear in both `VMale` (the view) and the `Employee` table.

- **Conditions for Inserting, Updating, and Deleting Rows**: If the view is a partial selection of columns from the table, then the columns that were not selected in the view must meet one of these criteria:
  - Be an **identity** column.
  - **Allow null** values.
  - Be a **calculated column**.
  - Have a **default value** defined.

2. DML on a View with Data from Multiple Tables

- **DELETE**: You **cannot delete** data directly from a view that joins multiple tables.
- **INSERT/UPDATE**: You can insert or update data, but only if the action affects columns from a single table in the view.

  Example of Error:

  ```
  INSERT INTO V_emp_dep (Fname, Dname) VALUES ('mostafa', 10); -- Error
  ```

  - This will cause an error because `Fname` is from the `Employee` table and `Dname` is from the `Departments` table.
- **Conditions for Inserting or Updating**: As with single-table views, the remaining columns in the referenced table(s) must meet one of the following conditions:
  - **Identity** column.
  - **Allow null** values.
  - **Calculated** column.
  - **Default value**.

3. Ensuring DML Integrity with `WITH CHECK OPTION`

If you want to prevent inserting or updating values that don't match the view's conditions, you can use the `WITH CHECK OPTION` to enforce this. This ensures that all rows in the view adhere to its filter criteria.

Example:

```
CREATE VIEW VMale1 AS SELECT * FROM Employee WHERE Sex = 'M' WITH CHECK OPTIO
N;
```

- Now, if you try to insert a record with `Sex = 'F'` in `VMale1`, SQL Server will prevent it, as it violates the view's condition (`Sex = 'M'`).

Indexed View

An **indexed view** in SQL Server is a view that has a physical copy of the view data stored on disk with a unique clustered index, improving query performance by materializing the data. Indexed views are particularly useful when a query involves complex aggregations, calculations, or joins that are frequently used, as they allow for faster retrieval of data by precomputing the view's result set.

Key Requirements for an Indexed View:

1. **Schema Binding**: The view must be created with the `WITH SCHEMABINDING` option, which locks the underlying tables' schemas to prevent structural changes.
2. **Unique Clustered Index**: To make a view an indexed view, a unique clustered index must be created on it. This index physically stores the data in the view.
3. **Restrictions**: Indexed views cannot contain columns with `MAX` data types, `TEXT` or `IMAGE` data types, or use `ROW_NUMBER`, `RAND`, and certain other non-deterministic functions.

Example of Creating an Indexed View

Suppose we have a table of `Sales` records, and we frequently query the total sales amount per region. Instead of running this aggregation each time, we can create an indexed view to store the results.

1. **Creating the Sales Table**:

```
CREATE TABLE Sales ( SalesID INT PRIMARY KEY, Region VARCHAR(50), SalesAm
ount DECIMAL(18, 2), SaleDate DATE );
```

2. **Creating the Indexed View**:

```
CREATE VIEW VRegionSales WITH SCHEMABINDING AS SELECT Region, SUM(SalesAm
ount) AS TotalSales FROM dbo.Sales GROUP BY Region;
```

3. **Adding a Unique Clustered Index**:

```
CREATE UNIQUE CLUSTERED INDEX IX_VRegionSales_Region ON VRegionSales (Reg
ion);
```

How This Works:

- The indexed view, `VRegionSales`, calculates the sum of `SalesAmount` per `Region` and stores the result on disk.
- Each time data in the `Sales` table is inserted, updated, or deleted, SQL Server automatically updates the `VRegionSales` indexed view.
- Queries that access `VRegionSales` can retrieve the precomputed total sales without having to re-aggregate each time, which can significantly boost performance, especially with large data sets.

Querying the Indexed View:

Once the indexed view is created, it can be queried just like a regular view:

```
SELECT * FROM VRegionSales WHERE Region = 'North';
```

Benefits:

- **Improved Query Performance**: Since the aggregation is precomputed and stored, querying this view is faster.
- **Automatic Updates**: Changes to the base table automatically update the indexed view, so data remains consistent.

Limitations:

- **Storage Cost**: Indexed views occupy additional storage as they store a materialized result set.
- **Overhead on DML Operations**: Inserts, updates, or deletes on the base tables cause updates in the indexed view, adding maintenance overhead.

When to Use the Schema Name

Using the schema name (`dbo` in this case) is required in SQL Server under specific conditions to avoid ambiguity and ensure the correct object is accessed. Here are the main situations:

1. **In Scalar Functions**: When calling a scalar function, prefix it with the schema name.
   - **Example**: `SELECT dbo.ScalarFunName(parameter);`
2. **In Indexed Views**: All tables referenced in an indexed view must use the schema name due to the `WITH SCHEMABINDING` option.
   - **Example in View Definition**: `FROM dbo.Employee`
3. **When Accessing Tables Across Databases**: If querying tables from another database, prefix the table name with the database and schema names.
   - **Example**: `SELECT * FROM ITI.dbo.instructor;`

This schema prefixing ensures that SQL Server accesses the correct objects and maintains consistency in queries, especially when dealing with indexed views or cross-database queries.

Property/Feature	Description	Limitations	Exam
Virtual Table	Acts as a virtual table that represents data from one or more tables.	- Does not store data; it only stores the query. Data is fetched at runtime, which may impact performance.	`CREAT` `Fname`
Hides Database Complexity	Hides the complexity of the schema or complex joins, presenting simpler views of data.	- Over-simplification may limit flexibility for users needing access to hidden columns or joins.	`CREAT` `Dname` `JOIN` `e.Dno`
Data Security	Limits access to specific rows or columns, enhancing security by restricting data exposure.	- View definition may still be accessible without encryption. Views may not prevent all indirect data access risks.	`CREAT` `Fname` `> 500`
Simplifies Complex Queries	Allows users to save complex queries for repeated use, simplifying development in applications.	- Frequent use of complex views with large tables can impact performance as queries re-run each time.	`CREAT` `SELEC` `Total` `Regio`
DML Operations	Supports INSERT, UPDATE, DELETE with conditions, allowing data changes via the view.	- DML operations are only allowed if certain rules are met (see below). Cannot use DML in multi-table views unless conditions are met.	`INSER` `('Ali`
Types of Views	Supports standard, partitioned, and indexed views for flexibility.	- Partitioned views require all participating tables to be structured similarly. Indexed views require schema binding and unique indexes.	See e
With Encryption	Encrypts the view definition to protect it from being viewed with tools like `sp_helptext`.	- Once encrypted, the view definition cannot be recovered directly. Be cautious, as there's no native decryption.	`CREAT` `ENCRY` `Emplo`
Schema Binding	Locks the underlying table schema, making it essential for indexed views and preventing schema changes on referenced objects.	- Schema-bound views require `dbo` schema reference for tables. Limits flexibility in modifying base tables (e.g., renaming columns).	`CREAT` `SCHEM` `Salary`
Indexed View	Improves performance by physically storing view data when	- Only available in Enterprise Edition or higher. Requires schema binding	`CREAT` `IX_Vd`

	indexed, optimizing retrieval for specific queries.	and a unique clustered index. Cannot use `MAX` data types.	
With Check Option	Enforces integrity by preventing changes that make data unavailable in the view (e.g., restricting insertions based on view conditions).	- Can be limiting if updates are needed that don't meet the filter criteria. Adds maintenance for adhering to filter rules.	`CREAT` `OPTIO` `WHERE`
No Parameters	Cannot pass parameters directly to a view; instead, views are defined with fixed conditions or filters.	- Limited flexibility. Parameterized stored procedures or inline table-valued functions are alternatives for dynamic conditions.	N/A
Multi-table DML Limitations	DML operations on multi-table views require strict conditions, allowing only single-table changes unless specific criteria are met.	- Cannot delete in multi-table views. Insert/update requires specific columns to allow `NULL`, `DEFAULT`, `IDENTITY`, or be calculated.	See e
Dependent on Base Tables	Views reflect changes in underlying tables automatically, making them dynamic and adaptable as base data changes.	- Changes in the base table schema (e.g., renaming or removing columns) can invalidate the view.	See e

---

Merge

The `MERGE` statement in SQL Server is a powerful way to synchronize two tables by performing `INSERT`, `UPDATE`, or `DELETE` operations in a single statement based on a specified condition. This can be especially useful for handling scenarios where you want to update existing records and insert new records simultaneously.

Example Scenario

Let's consider two tables: `LastTransaction` (which holds the last recorded transactions for employees) and `DailyTransaction` (which holds the current transactions).

Table Structures

LastTransaction:

```
id | name | salary -----|--------|------- 1 | ahmed | 1000 2 | omar | 2000 4
| mona | 3000
```

DailyTransaction:

```
id | name | salary -----|--------|------- 1 | ahmed | 1000 2 | omar | 2100 10
| sayed | 5000
```

Objective

- Update the salary of `ahmed` and `omar` in `LastTransaction`.
- Insert `sayed` into `LastTransaction` as it does not exist in it.
- `mona` should be left unchanged as it's not part of the daily transactions.

Various Formats of MERGE

1. **Basic Merge with Update and Insert:**

```
MERGE INTO LastTransaction AS T USING DailyTransaction AS S ON T.id = S.i
d WHEN MATCHED THEN UPDATE SET T.salary = S.salary WHEN NOT MATCHED THEN
INSERT (id, name, salary) VALUES (S.id, S.name, S.salary);
```

2. **Merge with a Subquery as Source:**

```
MERGE INTO LastTransaction AS T USING (SELECT id, name, salary FROM Daily
Transaction) AS S ON T.id = S.id WHEN MATCHED THEN UPDATE SET T.salary =
S.salary WHEN NOT MATCHED THEN INSERT (id, name, salary) VALUES (S.id, S.
name, S.salary);
```

3. **Conditional Update with Match:**

```
MERGE INTO LastTransaction AS T USING DailyTransaction AS S ON T.id = S.i
d WHEN MATCHED AND S.salary > T.salary THEN UPDATE SET T.salary = S.salar
y WHEN NOT MATCHED THEN INSERT (id, name, salary) VALUES (S.id, S.name,
S.salary);
```

4. **Insert Not Matched by Target:**

```
MERGE INTO LastTransaction AS T USING DailyTransaction AS S ON T.id = S.i
d WHEN MATCHED THEN UPDATE SET T.salary = S.salary WHEN NOT MATCHED BY TA
RGET THEN INSERT (id, name, salary) VALUES (S.id, S.name, S.salary);
```

5. **Insert Not Matched by Source:**

```
MERGE INTO LastTransaction AS T USING DailyTransaction AS S ON T.id = S.i
d WHEN MATCHED THEN UPDATE SET T.salary = S.salary WHEN NOT MATCHED BY SO
URCE THEN INSERT (id, name, salary) VALUES (S.id, S.name, S.salary);
```

Explanation of Each Format:

- **Format 1**: A straightforward `MERGE` that updates salaries for matched records and inserts new records from `DailyTransaction` that do not exist in `LastTransaction`.
- **Format 2**: Uses a subquery to define the source table, providing flexibility in how the source data is selected.
- **Format 3**: Updates salaries only if the new salary is greater than the existing one.
- **Format 4**: Inserts records that exist in `DailyTransaction` but not in `LastTransaction`.
- **Format 5**: Handles the case where records exist in `LastTransaction` but not in `DailyTransaction`, effectively ignoring those in this context.