DBM ASSIGNMENT-2

Q1 Explain clustering and indexing.

1. Clustering

Clustering refers to the physical arrangement of data records that share common characteristics, improving data retrieval efficiency.

Types of Clustering:

- **Clustered Tables:** Multiple tables with related data are stored in the same physical location to reduce disk I/O.
- **Clustered Indexing:** The order of data in the table is determined by the index itself, meaning data rows are stored in sorted order based on the indexed column. Each table can have only one clustered index.

Example of Clustering:

Consider a database with an **Employee** table containing DepartmentID. If employees from the same department are stored together, retrieval becomes faster when searching by DepartmentID.

2. Indexing

Indexing is a technique to speed up data retrieval by creating a data structure (index) that maps keys to their corresponding locations in a table.

Types of Indexing:

- **Primary Indexing:** Created on the primary key, which ensures uniqueness.
- **Secondary Indexing:** Created on non-primary key columns to enhance search performance.
- **Clustered Indexing:** Data is physically arranged based on the indexed column.
- **Non-clustered Indexing:** The index contains pointers to the actual data instead of arranging the table itself.

Example of Indexing:

If a table has a **Student** table with RollNo, an index on RollNo allows the database to quickly find records without scanning the entire table.

Q2. What are views? Give example.

Views in DBMS

A **view** is a virtual table that represents the result of a SQL query. It does not store data itself but provides a way to access and display data from one or more tables.

Example of a View

Consider an **Employee** table:

EmpID	Name	Department	Salary
101	John	HR	5000
102	Alice	IT	7000
103	Bob	Finance	6000

Creating a View

Suppose we want to create a view that displays only the employees from the IT department:

```
CREATE VIEW IT_Employees AS
SELECT EmpID, Name, Salary
FROM Employee
WHERE Department = 'IT';
```

Using the View

Now, we can retrieve data from the view just like a table:

```
SELECT * FROM IT_Employees;
```

Output:

EmpID	Name	Salary
102	Alice	7000

Q3. What are triggers? What are its different types?

Triggers in DBMS

A **trigger** is a special type of stored procedure that automatically executes in response to specific events on a table, such as INSERT, UPDATE, or DELETE. Triggers help enforce business rules, maintain data integrity, and automate processes.

Types of Triggers:

1. **Before Triggers**
    - Executes **before** the triggering event occurs.
    - Used to validate or modify data before changes are made.
    - Example: Preventing negative salary before inserting into an Employee table.
2. **After Triggers**
    - Executes **after** the event occurs.
    - Commonly used for logging changes or maintaining audit records.
    - Example: Keeping track of deleted records in an AuditLog table.

3. **Instead Of Triggers**
    - Executes **instead of** the triggering event.
    - Used mainly in views where direct modifications are restricted.
    - Example: Updating a complex view using an INSTEAD OF UPDATE trigger.
4. **Row-Level Triggers**
    - Executes **once per row** affected by the triggering event.
    - Example: Automatically updating stock quantity after a sale.
5. **Statement-Level Triggers**
    - Executes **once per SQL statement**, regardless of the number of rows affected.
    - Example: Logging an event when an UPDATE query modifies multiple rows.

Example of a Trigger

Scenario: Automatically log deleted employee records.

```
CREATE TRIGGER log_employee_deletion
AFTER DELETE ON Employee
FOR EACH ROW
INSERT INTO Employee_Log (EmpID, Name, Deleted_At)
VALUES (OLD.EmpID, OLD.Name, NOW());
```

Q4.What are the benefits of using exception handling in a DBMS application?

1. **Ensures Data Integrity**
    - Prevents partial updates or inconsistent data by rolling back transactions when errors occur.
2. **Improves Application Stability**
    - Prevents system crashes by handling unexpected database errors gracefully.
3. **Enhances Debugging and Maintenance**
    - Helps identify the root cause of issues by logging detailed error messages.
4. **Provides Better User Experience**
    - Displays meaningful error messages instead of abrupt failures, improving usability.
5. **Supports Transaction Management**
    - Uses TRY...CATCH blocks or EXCEPTION handlers to roll back changes in case of failures.
6. **Reduces Redundant Error Handling Code**
    - Centralized error handling avoids repetitive IF-ELSE checks in multiple queries.

Example in SQL (Using TRY...CATCH in SQL Server)

```
BEGIN TRY
   BEGIN TRANSACTION;
   INSERT INTO Employee (EmpID, Name, Salary) VALUES (101, 'John', -5000);
   COMMIT TRANSACTION;
END TRY
BEGIN CATCH
   ROLLBACK TRANSACTION;
   PRINT 'Error: Salary cannot be negative';
END CATCH;
```