**Database Management Assignment:-**

**Section A: Introduction to SQL/NoSQL**

1. You are working on a project where you need to store large amounts of structured and semi-structured data. Which type of database (SQL or NoSQL) would you choose and why? Explain with a practical example.

The choice between **SQL (relational)** and **NoSQL (non-relational)** databases depends on the nature of the data and scalability requirements.

**When to Choose SQL:**

* Structured data with fixed schema
* Complex relationships between entities
* ACID (Atomicity, Consistency, Isolation, Durability) compliance is required

**When to Choose NoSQL:**

* Semi-structured or unstructured data
* High scalability and flexibility
* Horizontal scaling (distributed architecture)

**Example:**  
If you're building an **e-commerce platform**, you might choose **SQL** (like MySQL or PostgreSQL) for managing transactions, inventory, and user data where relationships are key. However, for **product catalogs** with dynamic attributes (e.g., different specs for electronics vs. clothing), a **NoSQL document database like MongoDB** is a better fit, as it allows flexible schema storage.

1. A company wants to migrate from a relational database to a NoSQL database for better scalability. What challenges might they face? Discuss with an example.

Migrating from a relational database to NoSQL introduces several challenges:

1. **Schema Design Changes**
   * SQL databases have well-defined schemas, while NoSQL is schema-less or flexible.
   * **Example:** A relational database for a CRM system stores customer info across multiple normalized tables, while in NoSQL, customer data might be stored in a single document with nested fields.
2. **Data Consistency vs. Availability**
   * SQL ensures **strong consistency** through ACID transactions, while NoSQL often follows **eventual consistency** for scalability.
   * **Example:** A banking system needs strict transaction integrity, making NoSQL a poor choice for core financial operations.
3. **Query Complexity**
   * SQL supports powerful joins and aggregations, whereas NoSQL often requires data duplication or application-level joins.
   * **Example:** In a social media app, fetching user posts along with comments is straightforward in SQL using joins, but in NoSQL (like MongoDB), denormalization is needed for performance.
4. **Migration Effort and Data Mapping**
   * Data must be transformed and restructured to fit NoSQL models.
   * **Example:** Converting normalized customer data into JSON-based document format requires restructuring and possible duplication.
5. **Application Logic Changes**
   * Existing queries and business logic need updates to work with the NoSQL model.

**Section B: Advantages and Disadvantages of SQL/NoSQL**  
3. You are designing an e-commerce website's database. Explain the advantages and disadvantages of using SQL vs. NoSQL in this scenario.

**Using SQL (Relational Database - MySQL, PostgreSQL)**

**Advantages:**

* Structured Data & Relationships: Ideal for handling structured data like users, orders, and payments.
* ACID Compliance: Ensures data integrity, crucial for transactions (e.g., order payments).
* Standardized Querying: SQL provides powerful joins and aggregations for reporting and analytics.

**Disadvantages:**

* Scalability Challenges: SQL databases typically scale vertically (adding more power to a single server), which can become costly.
* Fixed Schema: Changes in product attributes (e.g., different specs for electronics vs. clothing) require schema alterations, leading to potential downtime.

**Using NoSQL (Non-Relational Database - MongoDB, DynamoDB)**

**Advantages:**

* Schema Flexibility: Ideal for storing dynamic product details (e.g., different categories with varying attributes).
* High Scalability: NoSQL databases scale horizontally (across multiple servers), handling large traffic loads efficiently.
* Faster Read/Writes: Optimized for handling large amounts of user-generated content (e.g., customer reviews, wishlists).

**Disadvantages:**

* No Strong ACID Guarantees: Some NoSQL databases prioritize performance over strict consistency, which may be a risk for transactions.
* Query Complexity: Lacks traditional joins, requiring data duplication or denormalization for efficiency.

4. A banking system requires high consistency and ACID compliance. Which database system (SQL or NoSQL) would you recommend? Justify your answer with a real-world use case.

For a banking system, **SQL (Relational Database)** is the best choice due to:

**ACID Compliance:**

* **Atomicity:** Ensures transactions (e.g., money transfers) either fully complete or roll back if any issue occurs.
* **Consistency:** Guarantees correct balances in accounts even under failures.
* **Isolation:** Prevents concurrent transactions from interfering.
* **Durability:** Ensures data is permanently stored.

**Data Integrity & Security:**

* Banking applications require strict access control, auditing, and regulatory compliance (e.g., PCI DSS, GDPR).

**Real-World Example:**

* JPMorgan Chase & Bank of America use Oracle and PostgreSQL for their core transactional systems.
* Stock exchanges like the New York Stock Exchange (NYSE) use SQL databases to ensure high consistency in financial transactions.

**Why Not NoSQL?**

* NoSQL databases prioritize availability over consistency, which is risky for real-time financial transactions.
* Eventual consistency could lead to issues like incorrect balances or duplicate transactions.

**Final Recommendation**: SQL (MS SQL, MySQL) for high-consistency banking applications.

**Section C: Managing Databases**  
5. You are a database administrator and need to perform routine maintenance on a production database. Describe at least three essential database management tasks you would perform.

As a Database Administrator (DBA), routine maintenance is crucial to ensure performance, security, and data integrity. Here are three essential tasks:

**1. Database Backup & Recovery Planning**

* Regularly schedule full and incremental backups to prevent data loss.
* Test backup restoration to ensure recoverability in case of failure.
* Use automated backup solutions like MySQL dump, PostgreSQL pg\_dump, or cloud snapshots (AWS RDS, Azure SQL Backup).
* Example: A financial institution ensures daily backups with a retention policy to comply with disaster recovery requirements.

**2. Index Optimization & Performance Tuning**

* Analyze slow queries using tools like EXPLAIN (MySQL, PostgreSQL).
* Rebuild or optimize indexes to speed up query execution.
* Identify unused or redundant indexes to reduce storage overhead.

Example: An e-commerce website experiences slow product searches; optimizing indexes on product\_name improves performance.

**3. Security Auditing & User Access Management**

* Regularly review user roles and permissions to prevent unauthorized access.
* Apply encryption for sensitive data (e.g., credit card info).
* Monitor logs for suspicious activities and set up alerts for potential security breaches.
* Example: A healthcare provider ensures HIPAA compliance by encrypting patient records and restricting access.

6. An online streaming service needs to optimize its database performance. What strategies can be used for effective database management in this case?

An online streaming service (like Netflix or Spotify) requires a **highly scalable and optimized database** for storing user preferences, video/audio metadata, and real-time recommendations.

**1. Use a NoSQL Database for Scalability**

* Choose NoSQL (e.g., Cassandra, DynamoDB, MongoDB) for distributed, high-speed data access.
* Ensures horizontal scaling to handle millions of concurrent users.
* Example: Netflix uses Cassandra to store metadata for videos, providing fast retrieval.

**2. Implement Caching for Faster Content Delivery**

* Use Redis or Memcached to cache frequently accessed data like user watch history.
* Reduces direct database queries and improves response times.
* Example: Spotify caches popular playlists to minimize database load.

**3. Database Sharding & Replication**

* Sharding: Splitting large tables across multiple databases to distribute load.
* Replication: Keeping read-only copies for faster queries and fault tolerance.
* 🔹 Example: YouTube shards user data across multiple databases to scale globally.

**4. Optimize Queries & Use Connection Pooling**

* Optimize SQL queries using indexes, proper joins, and avoiding unnecessary selects (SELECT \*).
* Use connection pooling (e.g., PgBouncer for PostgreSQL) to manage multiple user connections efficiently.
* Example: A video recommendation system optimizes queries fetching personalized suggestions.

By implementing these strategies, an online streaming service can ensure seamless user experience, minimize downtime, and handle massive traffic efficiently.

**Section D: Identifying System Databases in SQL Server**  
7. List and describe the system databases in SQL Server. Provide one practical use case for each system database.

SQL Server has **four main system databases** that are essential for managing the database environment.

**1. master**

* **Description:** Stores system-wide configuration settings, including instance-level metadata, logins, linked servers, and system stored procedures.
* **Use Case:** If the **master database is corrupted, SQL Server cannot start**.  
  🔹 **Example:** Restoring a **lost login account** by querying the syslogins table in the master database.

**2. msdb**

* **Description:** Used for **SQL Server Agent jobs, backups, alerts, and scheduling tasks**.
* **Use Case:** Stores backup history, making it essential for **automating database backups**.  
  🔹 **Example:** Retrieving the last backup time using:

sql

CopyEdit

SELECT TOP 1 \* FROM msdb.dbo.backupset ORDER BY backup\_finish\_date DESC;

**3. model**

* **Description:** Acts as a **template** for new databases created on the instance.
* **Use Case:** If you want all new databases to have a **default schema or specific settings** (like recovery model), you configure it in the model database.  
  🔹 **Example:** Setting the default **collation** and recovery model for all new databases.

**4. tempdb**

* **Description:** A temporary database used for **storing temporary tables, query results, and sorting operations**. It resets every time SQL Server restarts.
* **Use Case:** Reduces disk I/O by handling **temporary data storage** for large query processing.  
  🔹 **Example:** Running complex queries with ORDER BY uses tempdb for sorting.

8. You have accidentally deleted a user database in SQL Server. Which system database would you use to recover it, and how?

If a user database is **accidentally deleted**, the **msdb** system database is used to recover it.

**Steps to Recover a Deleted Database**

**1. Check Backup History (Using msdb Database)**  
Since msdb stores all backup records, you can check the latest backup:

SELECT TOP 1 \* FROM msdb.dbo.backupset WHERE database\_name = 'YourDatabase'

ORDER BY backup\_finish\_date DESC;

**2. Restore the Database from Backup**  
If a backup is available, restore it using:

RESTORE DATABASE YourDatabase

FROM DISK = 'C:\Backup\YourDatabase.bak'

WITH RECOVERY;

**3. Use Transaction Log Backup (Point-in-Time Recovery)**  
If you need to restore to a specific point before deletion, apply the transaction log backup:

RESTORE LOG YourDatabase

FROM DISK = 'C:\Backup\YourDatabase.trn'

WITH STOPAT = '2025-03-24 12:00:00', RECOVERY;

**4. Check if the MDF and LDF Files Exist**  
If the database was detached (but not deleted), try reattaching it:

EXEC sp\_attach\_db @dbname = 'YourDatabase',

@filename1 = 'D:\MSSQL\DATA\YourDatabase.mdf',

@filename2 = 'D:\MSSQL\DATA\YourDatabase.ldf';

**Key Takeaway:**

* If backups exist, **restore using msdb backup history**.
* If no backups exist, recovery might not be possible unless a third-party tool is used.
* Always schedule **automated backups** to avoid data loss.

**Section E: Normalization Forms (1NF, 2NF, 3NF, BCNF)**  
9. Given the following unnormalized table:

1. **First Normal Form (1NF) – Eliminate Repeating Groups**

* Each column should have atomic (indivisible) values.
* Ensure that each row contains a single value per attribute.
* If an order contains multiple products, they should be stored in separate rows.

| **OrderID** | **CustomerName** | **Product** | **Quantity** | **SupplierName** | **SupplierContact** |
| --- | --- | --- | --- | --- | --- |
| 101 | John Doe | Laptop | 1 | ABC Ltd. | 1234567890 |
| 102 | Jane Smith | Phone | 2 | XYZ Inc. | 9876543210 |

Now, the table is in 1NF because each attribute has atomic values.

1. **Second Normal Form (2NF) – Remove Partial Dependency**

* Must be in 1NF.
* No partial dependency (i.e., non-key attributes must depend on the whole primary key, not just part of it).
* The primary key here is (OrderID, Product).
* CustomerName depends only on OrderID, not on Product.
* SupplierName & SupplierContact depend only on Product, not on OrderID.

**Conversion (Breaking into two tables):**

**Orders Table**

| **OrderID** | **CustomerName** |
| --- | --- |
| 101 | John Doe |
| 102 | Jane Smith |

**OrderDetails Table**

| **OrderID** | **Product** | **Quantity** | **SupplierName** |
| --- | --- | --- | --- |
| 101 | Laptop | 1 | ABC Ltd. |
| 102 | Phone | 2 | XYZ Inc. |

**Suppliers Table**

| **SupplierName** | **SupplierContact** |
| --- | --- |
| ABC Ltd. | 1234567890 |
| XYZ Inc. | 9876543210 |

Now, the table is in 2NF because all non-key attributes fully depend on the entire primary key.

1. **Third Normal Form (3NF)**

* Must be in 2NF.
* No transitive dependencies (i.e., non-key attributes must depend directly on the primary key).
* SupplierContact depends on SupplierName, not directly on the primary key (OrderID).

**Final Tables in 3NF:**

**Orders Table**

| **OrderID** | **CustomerName** |
| --- | --- |
| 101 | John Doe |
| 102 | Jane Smith |

**OrderDetails Table**

| **OrderID** | **Product** | **Quantity** | **SupplierName** |
| --- | --- | --- | --- |
| 101 | Laptop | 1 | ABC Ltd. |
| 102 | Phone | 2 | XYZ Inc. |

**Suppliers Table**

| **SupplierID** | **SupplierName** | **SupplierContact** |
| --- | --- | --- |
| S1 | ABC Ltd. | 1234567890 |
| S2 | XYZ Inc. | 9876543210 |

Now, the database is in 3NF, removing redundancy and ensuring data integrity.

1. A company is facing redundancy issues in their database. How would applying BCNF help reduce redundancy? Explain with an example.

Boyce-Codd Normal Form (BCNF) is an extension of 3NF, ensuring that every determinant is a candidate key. It helps eliminate redundancy by further breaking down tables where anomalies exist in 3NF.

**Example:** University Course Registration (3NF Issue)

Consider the following table:

| **StudentID** | **Course** | **Instructor** |
| --- | --- | --- |
| 1001 | Math | Dr. Smith |
| 1002 | Math | Dr. Smith |
| 1003 | CS | Dr. Brown |

**Problem:**

* Instructor depends on Course, but Course is not a candidate key (StudentID + Course is the primary key).
* If the university changes the Math instructor, updating multiple rows is needed → redundancy issue.

**BCNF Solution:**  
Break the table into two:

**StudentCourses Table**

| **StudentID** | **Course** |
| --- | --- |
| 1001 | Math |
| 1002 | Math |
| 1003 | CS |

**CourseInstructors Table**

| **Course** | **Instructor** |
| --- | --- |
| Math | Dr. Smith |
| CS | Dr. Brown |

Now, the database is in BCNF, eliminating redundancy and improving consistency.

**End of Question Paper**