Module 3

Q. List and explain the core business drivers behind the NoSQL movement.
=>

1. Introduction

- NoSQL (Not Only SQL) refers to a family of non-relational databases designed to overcome the limitations of traditional relational database management systems (RDBMS).
- The movement emerged as businesses began generating massive, diverse, and fast-changing data that traditional systems struggled to handle efficiently.
- Several business and technical factors have driven organizations to adopt NoSQL solutions.

2. Core Business Drivers

1. Need for Scalability

- **Problem with RDBMS:**
  - Traditional SQL databases scale vertically (adding more CPU, RAM to a single server).
  - Vertical scaling is expensive and has physical limits.
- **NoSQL Solution:**
  - NoSQL databases scale horizontally (adding more commodity servers to share the load).
  - Distributed systems like Cassandra or MongoDB handle billions of records efficiently.
- **Business Impact:**
  - Enables applications like social media platforms, e-commerce, and IoT systems to serve millions of users simultaneously.

2. Handling Big Data and High Velocity

- **Problem with RDBMS:**
    - Structured tables cannot efficiently manage huge, unstructured datasets (videos, images, logs, sensor data).
    - High ingestion rates cause performance bottlenecks.
- **NoSQL Solution:**
    - Designed to store structured, semi-structured, and unstructured data without rigid schemas.
    - Databases like Hadoop HBase and Couchbase can handle high-velocity streaming data.
- **Business Impact:**
    - Companies can capture and process real-time data for analytics, fraud detection, and recommendations.

3. Flexibility and Schema-less Design

- **Problem with RDBMS:**
    - Schema changes (adding new columns or modifying tables) require downtime and migration, which slows development.
- **NoSQL Solution:**
    - Schema-less or dynamic schemas allow storing different attributes in different records.
    - Document-oriented databases like MongoDB let developers add new fields on the fly without breaking existing data.
- **Business Impact:**
    - Faster application development and iteration, critical for startups and agile teams.

4. Cloud and Distributed Computing Adoption

- **Problem with RDBMS:**
  - Traditional databases are not inherently cloud-friendly and require complex manual partitioning.
- **NoSQL Solution:**
  - Designed for distributed architectures across multiple nodes and data centers.
  - Built-in replication and partitioning (sharding) make them naturally suitable for cloud platforms.
- **Business Impact:**
  - Organizations reduce infrastructure costs and gain high availability and geographical distribution.

5. Cost Efficiency Using Commodity Hardware

- **Problem with RDBMS:**
  - Requires high-end servers to support vertical scaling.
- **NoSQL Solution:**
  - Runs on low-cost commodity hardware in clusters.
  - Automatic failover and replication avoid the need for expensive hardware redundancy.
- **Business Impact:**
  - Significant cost savings while still achieving high performance and reliability.

6. Demand for High Availability and Fault Tolerance

- **Problem with RDBMS:**
  - Centralized architecture means single point of failure.

    - Complex clustering and replication mechanisms are required for availability.

- **NoSQL Solution:**

    - Provides automatic replication and eventual consistency across nodes.

    - Systems like Cassandra guarantee always-on service with no downtime.

- **Business Impact:**

    - Businesses can provide 24/7 services even during node failures or maintenance.

7. Support for Modern Application Requirements

- **Problem with RDBMS:**

    - Difficult to handle location-based queries, user personalization, graph relationships, and real-time analytics.

- **NoSQL Solution:**

    - Offers specialized databases:
    - Key-Value stores for caching and session management (Redis).
    - Document stores for content management (MongoDB).
    - Graph databases for social networks (Neo4j).

- **Business Impact:**
    - Enables innovative features like personalized recommendations, fraud detection, and social graph analysis.

Q. What is a Key-Value Store?

=>

Definition

- A **key-value store** is the simplest type of NoSQL database where **data is stored as pairs of keys and values**.

- Each **key** is a unique identifier, and its **value** can be any kind of data (string, number, JSON, image, etc.).

- The database retrieves data **by key only**, ensuring **fast lookups** without complex queries.

How it Works

- Think of it as a **dictionary (hash map)**:

  - Key → "user123"

  - Value → {"name":"Alice", "age":25}

- To fetch the user's data, simply ask for the key "user123" — no need to scan the whole database.

Examples of Key-Value Stores

- **Redis** – in-memory key-value store used for caching and session management.

- **Amazon DynamoDB** – cloud-based key-value store.

- **Riak / BerkeleyDB / Voldemort** – distributed key-value stores for scalable applications.

Benefits of Using a Key-Value Store

1. High Performance

- **O(1) lookup time:** Data retrieval is extremely fast because keys are indexed internally.
- Ideal for **real-time applications** like gaming leaderboards, chat apps, and financial systems.

2. Simplicity

- Data model is straightforward — just keys and values.
- Easy to understand, implement, and integrate into applications without complex schemas.

3. Scalability

- Designed for **horizontal scaling** by adding more nodes.
- Distributed key-value stores (e.g., DynamoDB, Riak) automatically partition and replicate data.
- Handles **massive traffic and big datasets** efficiently.

4. Flexibility of Data Types

- Values can be **any type**: text, JSON, binary files, or serialized objects.
- No fixed schema, so developers can **store varied data without altering the structure**.

5. Fault Tolerance and High Availability

- Many key-value stores have **built-in replication** to keep copies of data on multiple nodes.
- Even if one node fails, the data is still available, making them highly reliable for critical systems.

6. Cost Efficiency

- Works well on **commodity hardware** and cloud-based clusters.
- Minimal operational overhead compared to traditional relational databases.

Q. What is graph store? Give an example where a graph store can be used to effectively solve a particular business problem

=>

Definition

- A **graph database (GDB)** is a type of NoSQL database that stores data using **graph structures** instead of traditional tables or documents.
- Data is represented as:
    - **Nodes:** Entities (e.g., users, products, locations)
    - **Edges:** Relationships between nodes (e.g., friendship, transactions, connections)
    - **Properties:** Attributes of nodes or edges (e.g., name, age, timestamp)
- Graph databases are **designed to efficiently handle relationships** and enable fast queries across connected data.
- **Examples:** Neo4j, Amazon Neptune, ArangoDB

Graph Representation

- Nodes store **data entities**, and edges capture **relationships** between them.
- Example structure for a social network:

Nodes (Users):

id	first name	last name	email	phone
1	Anay	Agarwal	anay@example.net	555-111-5555
2	Bhagya	Kumar	bhagya@example.net	555-222-5555

3	Chaitanya	Nayak	chaitanya@example.net	555-333-5555

Edges (Friendship relationships):

user_id	**friend_id**
3	1
3	2
3	4
3	5

Querying friends of a user in a **graph database** is faster because relationships are **directly stored as edges**.

When Do We Need a Graph Database?

1. **Many-to-Many Relationships**

    ○ Ideal for networks where entities are connected to multiple other entities (e.g., social media friends).

2. **Relationship-Centric Queries**

    ○ When **relationships are more important than individual data**, like tracking user interactions or supply chains.

3. **Low Latency on Large Datasets**

    ○ Graph databases can retrieve connected data in **constant time (O(1) per connection)**, unlike relational joins that can become very slow.

Business Example: Social Network Friend Query

- **Problem:** Chaitanya wants to see all her friends' profiles.

Relational DB Approach:

- Query involves joining **Users** table and **Friendship** table.
- Time complexity: O(M * log(N)) for M queries on N friendships.

Graph DB Approach:

- Locate Chaitanya node, traverse edges to friends.
- Time complexity: O(N) (single-step traversal per connection).

Result:

- Faster retrieval, especially as the network grows large.
- Real-time queries become feasible for millions of users.

Advantages of Graph Databases

- Handles **frequent schema changes** easily.
- Efficient for **managing large volumes of interconnected data**.
- Enables **real-time query responses**.
- Supports **intelligent data activation** like recommendations or fraud detection.

Disadvantages / Limitations

- Not always the best solution for all applications.
- Horizontal scaling can be challenging; may affect performance.
- Updating all nodes with a given parameter can be inefficient.
- May not outperform other NoSQL options in certain scenarios.

Q. CAP vs ACID

=>

Feature/Aspect	ACID (Relational DB)	CAP (NoSQL / Distributed DB)
Definition	Guarantees Atomicity, Consistency, Isolation, Durability in transactions	Guarantees Consistency, Availability, Partition tolerance in distributed systems
Primary Focus	Transaction correctness in a single database	System behavior under network partitions in distributed databases
Consistency	Strong consistency: database always remains valid after transactions	Trade-off: can be strong or eventual consistency depending on choice
Availability	High availability depends on the DB setup, but not a core principle	Must choose between Availability or Consistency in presence of network failures
Partition Tolerance	Not explicitly considered; assumes single node or tightly coupled system	Core requirement: system continues to operate even if network partitions occur
Use Cases	Banking, financial systems, ERP — critical transactions	Social media, large-scale web apps, e-commerce — scalable and distributed data
Implementation	Traditional RDBMS (MySQL, PostgreSQL, Oracle)	Distributed NoSQL (Cassandra, MongoDB, DynamoDB)

Transaction Support	Full ACID transactions supported	Often limited or eventual transaction guarantees
Scalability	Vertical scaling (bigger servers)	Horizontal scaling (add more nodes)
Trade-offs	Prioritizes correctness over availability in failures	Must balance C, A, P — cannot achieve all three simultaneously

Q. List the architectural patterns in NoSQL databases. Discuss the Key-Value and Document-Oriented patterns, focusing on their characteristics, use cases, and examples.

=>

1. Introduction

- NoSQL databases are designed to handle **large-scale, distributed, and unstructured data**.
- Unlike traditional relational databases, NoSQL databases follow **different architectural patterns** depending on the type of data and use case.
- Common architectural patterns include:
    1. **Key-Value Stores**
    2. **Document-Oriented Stores**
    3. **Column-Family Stores**
    4. **Graph Databases**

2. Key-Value Store Pattern

Definition

- Data is stored as **key-value pairs**: each key is unique, and the value can be **any type of data** (text, JSON, binary, etc.).
- The system retrieves data **by key only**, providing **fast lookups**.

Characteristics

- Extremely **simple data model**.
- Highly **scalable** via horizontal partitioning (sharding).
- Designed for **high performance and low-latency reads/writes**.
- Schema-less: no fixed structure for values.

Use Cases

- Caching (e.g., Redis for session data).
- Real-time analytics.
- User profiles, preferences, or settings storage.
- IoT or sensor data ingestion.

Examples

- Redis, DynamoDB, Riak, Voldemort.

3. Document-Oriented Store Pattern

Definition

- Data is stored as **documents**, often in **JSON, BSON, or XML format**.
- Each document contains **self-describing data**, often including nested structures.
- Documents are grouped into **collections**, and each document has a unique key (document ID).

Characteristics

- Supports **flexible, dynamic schemas**.
- Can query based on **document fields** rather than just key.
- Handles **complex data structures** naturally.
- Horizontally scalable through **sharding** and **replication**.

Use Cases

- Content management systems (CMS) storing articles or blog posts.

- E-commerce platforms storing product catalogs.
- Event logging and analytics with semi-structured data.
- Applications requiring **rapid iteration and schema evolution**.

Examples

- MongoDB, CouchDB, ArangoDB, Amazon DocumentDB.

4. Comparison of Key-Value vs Document-Oriented Patterns

Feature	Key-Value Store	Document-Oriented Store
Data Model	Simple key → value mapping	Documents with fields and nested structures
Query Flexibility	Only by key	By key and document fields
Schema	Schema-less, unstructured	Flexible schema, supports complex/nested data
Use Case	Caching, real-time session storage	CMS, product catalogs, event logs
Performance	Extremely fast, low latency	Fast, slightly more complex queries
Examples	Redis, DynamoDB, Riak	MongoDB, CouchDB, ArangoDB

Q. Describe the four ways by which big data problems are handled by NoSQL.

=>

- Big data is characterized by **Volume, Velocity, Variety, and Veracity**.
- Traditional relational databases often struggle with these challenges due to **rigid schemas, limited scalability, and high latency**.
- **NoSQL databases** address big data problems through **innovative architectural approaches** that enable distributed storage, flexible schemas, and high performance.

2. Four Ways Big Data Problems Are Handled by NoSQL

1. Horizontal Scalability

- **Definition:** Distributing data across multiple nodes (servers) instead of relying on a single powerful machine.
- **How it helps:**
  - Handles **huge volumes of data** by adding more nodes to the cluster.
  - Reduces performance bottlenecks by parallelizing reads and writes.
- **Example:**
  - **Cassandra** automatically partitions data across multiple nodes for linear scalability.
- **Benefit:** Avoids the cost and limits of vertical scaling in relational databases.

2. Schema Flexibility

- **Definition:** No fixed schema; data structure can evolve without downtime.
- **How it helps:**
  - Handles **variety of data**: structured, semi-structured (JSON/XML), or unstructured (images, logs).

    - Allows **rapid development and iterative changes** without complex migrations.
- **Example:**
    - **MongoDB** stores JSON-like documents with varying fields.
- **Benefit:** Supports fast-changing business requirements and diverse datasets.

3. High Availability and Fault Tolerance

- **Definition:** Ensures data is accessible even when nodes fail.
- **How it helps:**
    - Uses **replication** to keep multiple copies of data across nodes.
    - Provides **continuous availability** for real-time applications.
- **Example:**
    - **Riak** replicates data across nodes so queries succeed even if some nodes are down.
- **Benefit:** Enables robust systems for critical applications like e-commerce, social media, or banking.

4. Optimized for High-Velocity Data

- **Definition:** Efficient ingestion and retrieval of rapidly changing or streaming data.
- **How it helps:**
    - Handles **millions of writes per second** without slowing down the system.
    - Supports **real-time analytics** and operational monitoring.
- **Example:**
    - **Redis** and **Cassandra** can process real-time logs or sensor data at scale.
- **Benefit:** Supports modern applications like IoT, clickstream analysis, and recommendation engines.