NoSQL Introduction

NoSQL Definition

NoSQL stands for “Not Only SQL”, representing a category of non-relational databases.

Created due to the limitations of relational databases in handling large-scale, unstructured or semi-structured data.

Designed to meet the needs of applications requiring high scalability, flexibility ad real-time data processing.

Comparison to RDBMS

- **Schema Flesibility**: NoSQL databases are schema-less or have flexible schemas. Relational databases require a rigid predefined schema.
- **Data Structure**: NoSQL uses varied data models (document, key-value, column, graph). Relational databases organize data into tables with rows and columns.
- **Scalability**: NoSQL databases are designed to scale horizontally (adding more servers to distribute the load). Relational databases scale vertically (adding more power to a single server).
- **Consistency vs Availability**: NoSQL databases opt for eventual consistency to achieve higher availability and partition tolerance (BASE properties). Relational databases prioritise strong consistency (ACID properties).

Relevance in Modern Applications

- **Big Data and Real-Time Processing**: ideal for handling massive databases and real-time analytics (IoT)
- **Unstructured and Semi-Structured Data**: supports varied data types (JSON, text, multimedia) without a rigid schema.
- **Cloud-Native and Distributed Systems**: enables cloud-based distributed architecture to meet the demands of scalable global applications.

Schemaless or Flexible Schema Structure

- **No Fixed Schema**: no predefined schema, allowing for flexible data structures that can evolve over time.
- **Dynamic Fields**: fields can vary across records within the same collection, making it easy to add or remove fields without disrupting the database.
- **Ideal for Varied Data**: ideal for applications with diverse or evolving data structures (user-generated content, IoT data).

Distributed and Horizontally Scalable

- **Horizontal Scalling**: scale out by adding more servers to the cluster, which allows them to handle larger volumes of data and traffic efficiently.
- **Data Distribution**: data is distributed across multiple nodes, Ena bling higher availability and fault tolerance in the event of hardware failures.
- **Cloud-Frindly**: the distributed architecture makes them well-suited for cloud environments, supporting scalability across geographic regions.

Designed for Specific Data Models and Access Patterns

- **Specialised Data Models**: built to support specific data models (document, key-value, column-family, graph), each suited to different types of applications.
- **Optimized for Access Patters**: by focusing on particular use cases and access patters, it can deliver optimized performance for tasks like real-time analytics, session management or complex relationship querying.
- **Application-Centric**: NoSQL data model enables faster development and more efficient data storage and retrieval.

Types of NoSQL Database

Document Databases

- **Data Storage Format**: JSON-like documents (JSON, BSON, XML) allowing for nested and hierarchical structures.
- **Flexible Schema**: each document can have a unique structure, making it ideal for applicants with varying data requirements.

- **Use**: content management systems, catalogues, applications with complex data relationships (MongoDb, Couchbase).

Key-Value Stores

- **Data Structure**: key-value pairs, where each key is unique and maps to a specific value, which can be any type of data.
- **High Speed and Scalability**: optimized for quick data retrieval, making the ideal for caching, session management and real-time data processing.
- **Use**: applications requiring simple lookup or caching mechanisms (Reid's, Amazon DynamoDB, Riak).

Column-Family Stores

- **Column-Oriented Storage**: organizes data into columns, allowing efficient data retrieval for specific columns in large datasets.
- **Efficient for Aggregates and Analytics**: well-suited for read-heavy applications and big data analytics, as it can retrieve large datasets with minimal I/O.
- **Use**: time-series data, data warehousing, large-scale data analytics (Apache Cassandra, HBase).

Graph Databases

- **Node and Edge Structure**: data is stored in nodes (entities) and edges (relationships), making it easy to represent and query complex relationships.
- **Relationship-Driven**: optimized for application with interconnected data (social networks, recommendation systems, fraud detection).
- **Use**: applications where relationships between data are as important as the data itself (Neo4j, Amazon Neptune, ArangoDB).

Modelling Concepts

CAP Theorem

- **Consistency (C)**: ensures that all nodes in a distributed system see the same data at the same time, meaning any read returns the most recent write.
- **Availability (A)**: guarantees that every request receives a response, whether successful or not, even if some nodes are down.
- **Partition Tolerance (P)**: the system continues to operate despite network partitions, where nodes lose connectivity with each other, ensuring the system doesn't fail as a whole.

Importance o Choosing 2 out of 3

- **Trade-Offs**: according to the CAP theorem, in a distributed system, it's impossible to achieve all three properties (Consistency, Avalability and Partition Tolerance) simultaneously.
- **CA (Consistency + Availability)**: ensures consistent data and availability but sacrifices partition tolerance. Feasible only in systems without network partitioning (within a single location).
- **CP (Consistency + Partition Tolerance)**: maintains data consistency and partition tolerance but sacrifices availability. Typically in systems that prioritize accurate data over response times during network issues.
- **AP (Availability + Partition Tolerance)**: ensures high availability and partition tolerance but sacrifices strict consistency. In databases that prioritize uptime and user experience over immediate data consistency.

NoSQL Approach

- **Focus on AP**: most NoSQL databases are designed with high availability and partition tolerance, accepting eventual consistency to achieve resilience and scalability.
- **Eventual Consistency**: many NoSQL databases allow for eventual consistency, where data may take time to synchronize across nodes, suitable for applications where immediate consistency is not critical.
- **Flexibility**: some NoSQL databases provide configurable consistency levels, allowing developers to adjust the balance between consistency and availability based on application requirements.

BASE Properties

- **Basic Availability (BA)**: ensures that the system is available most of the time, providing responses even in the face of the partial system failures or networks issues.
- **Soft State (S)**: allows data to be in an intermediate state, with changes, that may not be immediately synchronized across all nodes, accepting temporary inconsistencies.
- **Eventual Consistency (E)**: guarantees that, given enough time without further updates, data will eventually become consistent across the system, even if it’s temporarily out of sync.

Contrast with ACID

- **ACID (Atomicity, Consistency, Isolation, Durbility)**: traditional relational databases prioritise strict transactional integrity, ensuring that all operations are reliably consistent and immediately synchronized.
- **BASE vs ACID**: while ACID focuses on strong consistency and immediate accuracy, BASE prioritise availability and performance, accepting temporary inconsistencies for better scalability.
- **Use**: ACID is suitable for applications requiring strict consistency (financial transactions), whereas BASE works well in applications that can tolerate eventual consistency (social media feeds).

BASE Benefits for Large Systems

- **High Availability**: BASE properties enable systems to remain operational and responsive, even during network disruptions or partial failures, essential for user-facing applications.
- **Scalability**: by relaxing strict consistency requirements, BASE allows distributed databases to scale horizontally, accommodating large volumes of data and traffic.
- **Efficient for Big Data**: BASE is well-suited for applications dealing with large datasets and real-time processing, where immediate consistency is less critical than performance and availability (e-commerce, analytics).

NoSQL and ACID. Example

ACID compliance within certain boundaries.

- **MongoDB**: supports multi-document ACID transactions in replica sets and sharded clusters, allowing users to group multiple operations that either all succeed or all fail.
- **Couchbase**: offers ACID transactions within its document-based and key-value architectu, supporting multi-document transactions.
- **FoundationDB**: distributed NoSQL database designed with ACID transactions in mind, used in scenarios where consistency and scalability are equally critical.

Trade Offs

ACID-compliant transactions in NoSQL may have performance or scalability trade-offs, particularly in highly distributed environments, where maintaining consistency across nodes can increase latency.

Many NoSQL databases allow developers to enable or disable ACID compliance based on their specific needs, providing flexibility in balancing performance with data integrity.

Data Modelling in NoSQL

- **Query-Driven Design**: NoSQL data modelling often starts by understanding how data will be accessed, focusing on optimizing for specific queries rather than adhering to a strict schema.
- **Read/Write Optimization**: models are designed to minimize the number of queries required for common access patterns, improving performance by grouping related data together.
- **Trade-Offs**: data modelling decision are guided by the need for quick access and scalability, rather than normalization, which is common in relational databases.

Denormalization and Embedded Documents

- **Denormalization**: data is often Denormalization to reduce the need for complex joins or multiple queries, storing redundant data to improve performance.

- **Embedded Documents**: allows related data to be stored within a single document or collection, enabling faster read operations by reducing the need to retrieve from multiple sources.

Different to RDMS Modelling

- **Schema Flexibility**: NoSQL does not require a fixed schem, allowing data structures to evolve as application needs change.
- **Joins**: NoSQL databases avoid joins by using embedded or denormalized structures, which improves performance but may lead to data redundancy.
- **Application-Specific Models**: while relational modelling is generally designed to be versatile and normalized, NoSQL modelling is tailored to the specific requirements of the application, often sacrificing normalization for performance and scalability.

MongoDB

Introduction to MongoDB

- **Data Model**: MongoDB stores data in flexible, JSON-like documents, making it a document-oriented database that supports complex nested data structures.
- **Schema Flexibility**: each document can have a unique structure, allowing for dynamic data models that can evolve as application requirements change.
- **Ideal for NoSQL Use**: well-suited for applications that benefit from a schema-less design (content management, e-commerce).

Data is Stored in BSON

- **Binary JSON (BSON)**: MongoDB uses BSON to store documents, which extends JSON with additional data types and is optimized for storage and speed.
- **Efficient Data Handling**: BSON’s binary format enables fast data traversal and indexing, enhancing MongoDB’s performance for both small nd large datasets.

- **Supports Rich Data Types**: allows for a variety of data types, including nested arrays and sub-documents, making it highly flexible for complex data models.

Popular Uses

- **Widely Used in Web Development**: MongoDB's scalability, flexible schema and ease of integration make it a popular choice for web applications and APIs.
- **Real-Time Analytics**: frequently used for applications needing real-time analytics or high data throughput (IoT platforms, social media, online gaming).
- **Scalable Architecture**: MongoDB's distributed, horizontally scalable nature makes it ideal for modern, large-scale applications that require robust data handling across multiple servers.

Sharding and Replica Sets

- **Sharding**: MongoDB supports sharding, a method for distributing data across multiple servers (shards), which helps handle large datasets and high transaction volumes by dividing data horizontally.
- **Replica Sets**: MongoDB uses replica sets for redundancy, where multiple copies of data are stored across different nodes to ensure availability and data recovery in case of failure.
- **High Availability**: sharding and replica sets together provide both horizontal scalability and high availability, making MongoDB resilient and capable of handling Large-scale applications.

Data Distribution and Management

- **Automatic Data Distribution**: the sharding mechanism automatically distributes data across shards, balancing the load and optimizing performance across servers.
- **Primary and Secondary Nodes**: in replica sets, data is written to the primary node and replicated to secondary nodes, ensuring data consistency and availability.

- **Fault Tolerance**: if a primary node fails, a secondary node is automatically promoted to primary, ensuring minimal downtime and data durability.

Sharding (Distributed Databases)

Sharding is the practice of splitting data horizontally across multiple nodes or servers, with each shard containing a subset of the data. Each shard functions as an independent database, storing only a portion of the overall dataset.

- **Purpose**: sharding is used to handle high data volumes and distribute the load across multiple machines, which helps with both **scalability** and **fault tolerance**.
- **Use**: distributed relational databases, data warehouses, big data systems.

Horizontal Scling in MongoDB

- **Horizontal Scaling Across Shards**: MongoDB scales horizontally by adding new shards as data grows, allowing the system to handle increasing workloads without major architectural changes.
- **Elastic Scalability**: new servers can be added to MongoDB clusters with minimal disruption, providing a flexible and scalable environment.
- **Application Independence**: MongoDB's approach to scaling is largely managed within the database layer, reducing the need for complex application-level changes to accommodate data growth.

Collections and Documents in MongoDB

- **Collections**: MongoDB organizes data into collections (like tables in relational databases, but do not enforce a strict schema) allowing document structures within the same collection.
- **Documents**: each document is a record stored in BSON format (Binary JSON) and represents a single data entry (like a row in relational databases) with a flexible structure.
- **Schema-Free Collections**: collections can store documents with different fields and structures, enabling MongoDB to accommodate changing data requirements over time.

Flexible Schema Structure

- **Schema Flexibility**: no predefined schema, allowing documents to be customized as needed, making it easy to add or remove fields without database restructuring.
- **Dynamic Data Handling**: the flexible schema structure allows to handle various data types and evolving data models, ideal for applications with frequently changing data needs.
- **Supports Rapid Development**: this flexibility reduces the need for extensive data modelling upfront, enabling quicker development and easier adaptation to new requirements.

Nested Docuemts and Arrays

- **Nested Documents**: documents can contain other documents as values within fields, enabling complex hierarchical data representations.
- **Arrays**: fields in a document can holds arrays, allowing multiple values (such as list of tags, comments, items) to be stored within a single field.

CRUD Operations in MongoDB

- **Create**: inserts new documents into a collection, allowing data to be added dynamically without predefined schema constraints.
- **Read**: retrieves documents from a collection based on specified criteria, supporting both simple lookups and complex queries.
- **Update**: modifies existing documents, either fully replacing them or updating specific fields within a document.
- **Delete**: removes documents from a collection bed on given conditions, helping to manage and clean up data.

```
-- Create
db.collection.insertOne({ name: "Alice", age: 30 }) -- Insert

-- Read
db.collection.find({ age: { $gt: 25 } }) -- Finds documents w

-- Update
db.collection.updateOne({ name: "Alice" }, { $set: { age: 31
```

```
-- Delete
db.collection.deleteOne({ name: "Alice" }) -- Deletes the fir
```

MongoDB Commands vs SQL

- **Document-Based Queries**: queries use a document structure rather than SQLs tabular format.
- **Flexible Schema**: no flexible schema, fields and data types can vary across documents without altering the collection structure.
- **Method-Based Syntax**: MongoDB uses JavaScript-like commands (find(), insertOne()) instead of SQL's declarative language, making MongoDB syntax closer to programming language operations than traditional SQL queries.

Indexing in MongoDB

- **Single-Field Index**: most basic index type, created on a single field, useful for straightforward queries.
- **Compound Index**: index on multiple fields allowing MongoDB to optimize queries that use multiple criteria.
- **Multikey Index**: used for indexing arrays within documents, enabling searches on array values.
- **Text Index**: allows text search within strings, useful for queries that involve keywords or phrases.
- **Geospatial Index**: designed for location-based queries, enabling efficient searching on geographical data like latitude and longitude.

```
-- Create an index
db.collection.createIndex({ field: 1 }) -- create an ascendin

-- Viewing indexes
db.collection.getIndexes() -- displays all indexes on a colle

-- Removing indexes
db.collection.dropIndex("index_name") -- delete an unnecessar
```

```
-- Index management
-- Regularly monitor and optimise indexes based on query patt
```

Aggregation Framework: Stages

- **$match**: filters documents to pass only those that meet specific criteria, similar to SQL's WHERE clause, and is often the first stage to reduce data early in the pipeline.
- **$group**: groups documents by a specified field and allows for aggregating data within each group (calculating sums, averages), similar to SQL's GROUP BY.
- **$sort**: orders the documents by specified fields in ascending or descending order, enabling organized output (like SQL's ORDER BY clause).
- **Other Stages**: MongDB's aggregation framework includes additional stages like $project (reshaping data), $limit (limiting output), $lookup (joins).