<u>Session 11:</u>

- ❖ **Practical Design of NoSQL**
- ❖ **NOSQL**

Topic: Schema structure for NoSQL database

The database schema is the structure of a database system, described in a formal language supported by the database management system (DBMS). In simpler terms, it represents the blueprint of your data environment, defining how data is organized and how the relations among them are associated.

Designing an effective schema is crucial to ensuring efficient data storage, retrieval and manipulation. A poorly designed database schema can negatively impact business operations, affecting the speed and reliability of data-driven decision-making and even hindering scalability.

<u>SQL and NoSQL:</u>

SQL (Structured Query Language) databases use a predefined schema, meaning the structure of the data must be set before data storage. They are based on the relational model and are perfect for complex queries and transactions that require atomicity, consistency, isolation and durability (ACID) properties.

NoSQL databases are non-relational and provide a flexible schema. This flexibility is a strong point in scenarios that require handling large amounts of diverse data or when the data structure is expected to change over time. NoSQL databases include MongoDB, Cassandra and DyanamoDB.

<u>NoSQL database schema design:</u> For NoSQL databases, the following things should be:

1. **Design for your queries**: Unlike SQL, NoSQL databases are designed with the query in mind. Identify your application's query patterns and data access needs first and then design your schema.
2. **Flexible schema**: NoSQL databases are schema-agnostic. This flexibility allows you to store data in the format that best suits your needs, whether it be key-value pairs, wide-column stores, graph databases or document databases.
3. **Consider data duplication**: Unlike SQL databases where normalization is key, NoSQL databases often leverage data duplication to optimize performance.

4. **Scalability**: NoSQL databases are designed for horizontal scalability. Design your database schema with future growth in mind.

5. **Consistent hashing**: NoSQL databases use consistent hashing for data distribution. Understand this mechanism to avoid hotspots and ensure balanced data distribution across your database cluster.

Topic: Changing Document Databases

A document database (also known as a document-oriented database or a document store) is a database that stores information in documents. A Document Data Model is a lot different than other data models because it stores data in JSON or XML documents.

Document databases offer a variety of advantages:

- An intuitive data model that is fast and easy for developers to work with.
- A flexible schema that allows for the data model to evolve as application needs change.
- The ability to horizontally scale out.

Document databases are considered to be non-relational (or NoSQL) databases. Instead of storing data in fixed rows and columns, document databases use flexible documents.

Documents store data in field-value pairs. The values can be a variety of types and structures, including strings, numbers, dates, arrays, or objects. Documents can be stored in formats like JSON, BSON, and XML.

How to create json file:

```
{
  {"id" : "1201", "name" : "satish", "age" : "25"}
  {"id" : "1202", "name" : "krishna", "age" : "28"}
  {"id" : "1203", "name" : "amith", "age" : "39"}
  {"id" : "1204", "name" : "javed", "age" : "23"}
  {"id" : "1205", "name" : "prudvi", "age" : "23"}
  {"id" : "1205", "name" : "prudvi", "age" : "23"}
}
```

Document Database Design: Designing a document database involves several key decisions, including:

1. **Data modelling:** This involves deciding how to organize and structure the data in your database. In document databases, data is stored in the form of documents, which can

include nested data structures. The structure of the documents should be designed to support the queries and use cases of your application.

2. **Indexing:** Indexing allows for fast searching and querying of the data. In document databases, you can index specific fields in the documents to improve query performance. It's important to carefully consider which fields to index, as too many indexes can lead to decreased performance.
3. **Sharding:** Sharding is the process of distributing the data across multiple servers to improve performance and scalability. In a document database, sharding can be done based on the value of a specific field in the documents, such as the user ID.
4. **Replication:** Replication involves creating multiple copies of the data to improve reliability and availability. In document databases, replication can be done in a variety of ways, such as master-slave replication or peer-to-peer replication.
5. **Data validation:** Data validation involves specifying rules for how data is stored in the database. Some document databases support data validation, which can help ensure data consistency and integrity.
6. **Security:** Document databases should have a robust security mechanism to protect against unauthorized access to the data. It's important to consider how to secure data at rest and in transit, as well as how to authenticate and authorize users.
7. **Backup and Recovery:** Document databases should have a robust backup and recovery plan in place to ensure that data is not lost in case of a disaster.

Key features of document databases: Document databases have the following key features:

- **Document model:** Data is stored in documents (unlike other databases that store data in structures like tables or graphs). Documents map to objects in most popular programming languages, which allows developers to rapidly develop their applications.
- **Flexible schema:** Document databases have a flexible schema, meaning that not all documents in a collection need to have the same fields. Note that some document databases support schema validation, so the schema can be optionally locked down.
- **Distributed and resilient**: Document databases are distributed, which allows for horizontal scaling (typically cheaper than vertical scaling) and data distribution. Document databases provide resiliency through replication.

- **Querying through an API or query language**: Document databases have an API or query language that allows developers to execute the CRUD operations on the database. Developers have the ability to query for documents based on unique identifiers or field values.

<u>Applications of document databases?</u>

Document databases are general-purpose databases that serve a variety of use cases for both transactional and analytical applications:

- Single view or data hub
- Customer data management and personalization
- Internet of Things (IoT) and time-series data
- Product catalogs and content management
- Payment processing
- Mobile apps
- Mainframe offload
- Operational analytics
- Real-time analytics

<u>Topic: Schema Evolution in Column-Oriented Databases</u>

Schema evolution in column-oriented databases refers to the ability of the database system to accommodate changes to the database schema without requiring downtime or data migration. Column-oriented databases are designed to store and retrieve data in a column-wise fashion.

Schema evolution keeps track of schema versions and their differences. Query decomposition allows the user to query old and new documents in a single query, as if they all conformed to the latest schema version, using knowledge of the differences between versions.

let’s consider an example table with three columns: “Name,” “Age,” and “City.” In traditional databases that use **<u>row-oriented storage</u>**, the data might be stored like this:

Row1: [“John Smith”, 30, “New York”]

Row2: ["Jane Doe", 25, "Chicago"]

Row3: [“Bob Johnson”, 35, “Miami”]

In a column-oriented database, the same data type would be stored in one column. It will be stored like this:

Name: [“John Smith”, “Jane Doe”, “Bob Johnson”]

Age: [30, 25, 35]

City: [“New York”, “Chicago”, “Miami”]

Advantages of Columnar-oriented Databases: There are three key benefits to using column-oriented storage:

1. **Data compression:** Column-oriented databases allow for better data compression ratios compared to row-oriented databases.
2. **Query performance**

In columnar data storage, the database engine can read and process only the necessary columns, reducing I/O and improving query performance.

3. **Scalability**

Column-oriented databases are highly scalable. They allow data engineers to add more columns to a table without affecting the existing columns.

Columnar Storage Formats:

Columnar storage formats are specific implementations that define how data is organized and stored in a columnar database. These formats optimize storage, compression, and query performance for columnar data.

A. Apache Parquet

Parquet is a popular columnar storage format used in big data processing frameworks like Apache Hadoop and Apache Spark. It offers efficient compression and encoding techniques, enabling high-performance query execution.

B. Apache ORC (Optimized Row Columnar)

Apache ORC is a high-performance columnar format for data processing frameworks. It aims to provide efficient storage, compression, and query execution for analytical workloads.

Deal with Schema in column-oriented databases:

1. **Adding Columns:** One of the most common schema evolution tasks is adding new columns to existing tables. In column-oriented databases, this operation is usually efficient because each column is stored separately, so adding a new column typically does not require moving or rewriting existing data.

2. **Removing Columns:** Similarly, removing columns from a table is generally straightforward in column-oriented databases. Since each column is stored independently, removing a column typically involves updating metadata rather than physically deleting data.
3. **Changing Data Types:** Schema evolution may also involve changing the data type of a column. For example, converting a column from an integer type to a string type. This operation can be more complex, especially if it involves data type conversions that require data transformation or validation.
4. **Renaming Columns:** Renaming columns is another common schema evolution task. This operation usually involves updating metadata and any dependent objects (such as views or stored procedures) but does not typically require moving or altering data.
5. **Compatibility and Versioning:** Column-oriented databases often provide features for managing schema versions and ensuring compatibility between different versions of the schema. This may include tools for automatic schema migration, version control mechanisms, and schema validation during data loading or querying.
6. **Performance Considerations:** While column-oriented databases excel at handling schema changes without major performance impacts, it's essential to consider performance implications for complex schema changes or large datasets. Some operations, such as changing the data type of a heavily indexed column, may require careful planning to avoid performance degradation.
7. **Data Migration and Backward Compatibility:** When making significant schema changes that require data migration, column-oriented databases may offer tools or guidelines for managing data migration efficiently while ensuring backward compatibility with older versions of the schema.

When to use and when not to use a columnar database

A. column store is ideal for:

- **Analytical workloads:** Columnar databases are ideal for OLAP workloads and aggregation queries. It provides fast query performance and efficient data retrieval, making it well-suited for business intelligence, reporting, and data analysis tasks.
- **High data volumes**: If your data size is significant and you want to minimize storage costs and optimize data retrieval speed, a columnar database can be suitable.

- **Selective data access:** A columnar database can provide significant performance improvements if your data processing and analysis primarily involves accessing specific columns or attributes rather than the entire dataset.
- **Time-series analysis: A** columnar database is a good option for time-series analysis, where data is frequently queried and aggregated based on timestamps.
- **Data warehousing:** Columnar databases are widely used in data warehousing environments where efficient storage and rapid query performance are critical. It supports complex analytics and querying of large volumes of structured and semi-structured data.

<u>**B.Columnar database is not suitable for**</u>:

- **Transactional workloads: A** columnar database is not built for transactional operations that involve frequent updates, inserts, or deletes. It cannot be used for Online Transaction Processing (OLTP) workloads that involve incremental data loading.
- **Real-time data ingestion:** A columnar database may not be the best choice if you require real-time data ingestion and immediate availability of the latest data. Using a column store may introduce some latency in data ingestion and updates.
- **Small datasets:** If your dataset is relatively small and query performance is not a significant concern, simpler storage formats can suffice.
- **Random access patterns:** If your workload involves frequent random access to different rows or records within the dataset, row-based storage formats may perform better.
- **Limited storage resources:** Although columnar databases provide efficient storage utilization, they may require additional computational resources. Consider the trade-off between storage savings and processing overhead.

<u>Key challenges and how to manage them</u>

Several potential challenges may arise when using a columnar database:

- **Write performance:** Write operations, like data ingestion and updates, can be slower than row-oriented storage. To overcome this challenge, we can employ batch processing or buffering techniques to optimize the write operations.

- **Schema evolution:** Managing schema evolution can be challenging, especially when dealing with large datasets or complex data structures.
- **Data skew and performance variability:** In some cases, specific columns may have data skew, where the distribution of values across the column is uneven. This can result in performance variability, with queries on skewed columns taking longer to execute.
- **Data updates and deletes:** A columnar database may not perform well when it comes to frequent updates of individual records because they may require rewriting entire column chunks.
- **Null values:** If your dataset has many null values in certain columns, it can lead to inefficient storage and query performance. Null values take up space in a columnar database and may result in storage waste.

Key factors for implementation of columnar storage

To implement a columnar database that meets performance, scalability, and analytical requirements we need to focus on the following facts:

- **Understand your data and workload:** Gain a deep understanding of your data characteristics and the specific analytical workloads. Analyse the query and data access patterns and performance requirements to determine which columns are frequently accessed and should be prioritized.
- **Choose the correct format:** Evaluate different columnar formats' features, performance, and compatibility. Consider compression capabilities, schema evolution support, integration with existing tools, and ecosystem support.
- **Optimize data organization and compression:** Experiment with different compression techniques and configurations to find the best balance between storage efficiency and query performance.
- **Plan for schema evolution:** If your data schema is likely to evolve, plan for schema evolution in advance. Choose a storage format that supports schema evolution and design strategies for managing schema changes while minimizing disruption to existing processes.
- **Leverage indexing:** Continuously analyse the query patterns and create appropriate indexes to ensure accurate query planning and execution.

- **Monitor and optimize performance:** Track query execution times, data ingestion rates, and storage utilization to identify areas for optimization. Regularly review and fine-tune configurations based on evolving data and workload patterns.

- **Data Evolution in Key/Value Stores**

  A key-value database, AKA key-value store, associates a value (which can be anything from a number or simple string to a complex object) with a key, which is used to keep track of the object. In its simplest form, a key-value store is like a dictionary/array/map object.

  When to use a key-value database

  - Real time random data access, e.g., user session attributes in an online application such as gaming or finance.
  - Caching mechanism for frequently accessed data or configuration based on keys.
  - Application is designed on simple key-based queries.

    MongoDB as a key-value store

    MongoDB stores data in collections, which are a group of BSON (Binary JSON) documents   where each document is essentially built from a field-value structure.

    ```
    {
      name: "John",
       age : 35,
       dob : ISODate("01-05-1990"),
       profile_pic : "https://example.com/john.jpg",
       social : {
           twitter : "@mongojohn",
             linkedin : "https://linkedin.com/abcd_mongojohn"
            }
    }
    ```

    Schema design to support key value

    MongoDB documents can be complex objects, applications can use a schema design to minimize index footprints and optimize access for a “key-value” approach. This design pattern is called the Attribute Pattern and it utilizes arrays of documents to store a “key-value” structure.

```
        attributes: [
            {
            key: "USA",
            value: ISODate("1977-05-20T01:00:00+01:00")
            },
            {
            key: "France",
            value: ISODate("1977-10-19T01:00:00+01:00")
            },
            {
            key: "Italy",
            value: ISODate("1977-10-20T01:00:00+01:00")
            },
            {
            key: "UK",
            value: ISODate("1977-12-27T01:00:00+01:00")
            },
            ...
          ]
```

Advantages of key-value databases

key-value databases are NoSQL databases. They allow flexible database schemas and improved performance at scale for certain use cases. The advantages of key-value stores are as following:

1. **Scalability**

   key-value databases provide built-in support for advanced scaling features. They scale horizontally and automatically distribute data across servers to reduce bottlenecks at a single server.

2. **Ease of use**

   Key-value databases follow the object-oriented paradigm that allows developers to map real-world objects directly to software objects. Several programming languages, such as Java, also follow the same paradigm.

Applications of key-value databases

1.Session management

A session-oriented application, such as a web application, starts a session when a user logs in to an application and is active until the user logs out or the session times out.

Shopping cart

2.Metadata storage engine

Your key-value store can act as an underlying storage layer for higher levels of data access. For example, you can scale throughput and concurrency for media and entertainment workloads such as real-time video streaming and interactive content.

3.Caching

You can use a key-value database for storing data temporarily for faster retrieval. For example, social media applications can store frequently accessed data like news feed content. In-memory data caching systems also use key-value stores to accelerate application responses.