1. Introduction to Databases

* **Problem Statement:** Addresses the necessity of databases by using an example of a communication application like WhatsApp, highlighting the inefficiencies of file storage for managing data.
* **Limitations of File Storage:**
  + Lack of concurrent management for users accessing from different locations.
  + Inability to grant different access rights to users.
  + Challenges in scaling and quick content searching.
* **Solution:** Introduction of databases as a structured, easily manageable solution for storing, retrieving, modifying, and deleting data.
* **Types of Databases:**
  + **SQL (Relational Databases):** Structured with a predetermined schema.
  + **NoSQL (Non-relational Databases):** Unstructured, dynamic schema.
* **Advantages of Using Databases:**
  + Efficient management of large data.
  + Ensured data consistency.
  + Easy data updates and high security.
  + Ensured data integrity and high availability.
  + Scalability through data replication and partitioning.

2. Types of Databases

* **Relational Databases (SQL):** Stores data in tables with fixed schemas; ideal for complex queries and ensures data integrity.
* **Non-Relational Databases (NoSQL):** Flexible schema for storing unstructured data; suitable for rapid development and horizontal scaling.
* **Key-Value Stores:** Simple, usually used for caching and storing session information.
* **Document Stores:** Stores data in document-like structures; great for content management systems.
* **Graph Databases:** Efficient in handling data expressed in graphs (networks); useful for social networks or recommendation engines.

Relational databases provide the atomicity, consistency, isolation, and durability [(ACID)](https://www.educative.io/edpresso/what-are-acid-properties-in-a-database) properties to maintain the integrity of the database. ACID is a powerful abstraction that simplifies complex interactions with the data and hides many anomalies (like dirty reads, dirty writes, read skew, lost updates, write skew, and phantom reads) behind a simple transaction abort.

But ACID is like a big hammer by design so that it’s generic enough for all the problems. If some specific application only needs to deal with a few anomalies, there’s a window of opportunity to use a custom solution for higher performance, though there is added complexity.

Let’s discuss ACID in detail:

* **Atomicity:** A transaction is considered an atomic unit. Therefore, either all the statements within a transaction will successfully execute, or none of them will execute. If a statement fails within a transaction, it should be aborted and rolled back.
* **Consistency:** At any given time, the database should be in a consistent state, and it should remain in a consistent state after every transaction. For example, if multiple users want to view a record from the database, it should return a similar result each time.
* **Isolation:** In the case of multiple transactions running concurrently, they shouldn’t be affected by each other. The final state of the database should be the same as the transactions were executed sequentially.
* **Durability:** The system should guarantee that completed transactions will survive permanently in the database even in system failure events.

### **Why relational databases?**

#### **Flexibility**

In the context of SQL, **data definition language (DDL)** provides us the flexibility to modify the database, including tables, columns, renaming the tables, and other changes. DDL even allows us to modify schema while other queries are happening and the database server is running.

#### **Reduced redundancy**

One of the biggest advantages of the relational database is that it eliminates data redundancy. The information related to a specific entity appears in one table while the relevant data to that specific entity appears in the other tables linked through foreign keys. This process is called normalization and has the additional benefit of removing an inconsistent dependency.

#### **Concurrency**

Concurrency is an important factor while designing an enterprise database. In such a case, the data is read and written by many users at the same time. We need to coordinate such interactions to avoid inconsistency in data—for example, the double booking of hotel rooms. Concurrency in a relational database is handled through transactional access to the data. As explained earlier, a transaction is considered an atomic operation, so it also works in error handling to either roll back or commit a transaction on successful execution.

#### **Integration**

The process of aggregating data from multiple sources is a common practice in enterprise applications. A common way to perform this aggregation is to integrate a shared database where multiple applications store their data. This way, all the applications can easily access each other’s data while the concurrency control measures handle the access of multiple applications.

#### **Backup and disaster recovery**

Relational databases guarantee the state of data is consistent at any time. The export and import operations make backup and restoration easier. Most cloud-based relational databases perform continuous mirroring to avoid loss of data and make the restoration process easier and quicker.

## **Why non-relational (NoSQL) databases?**

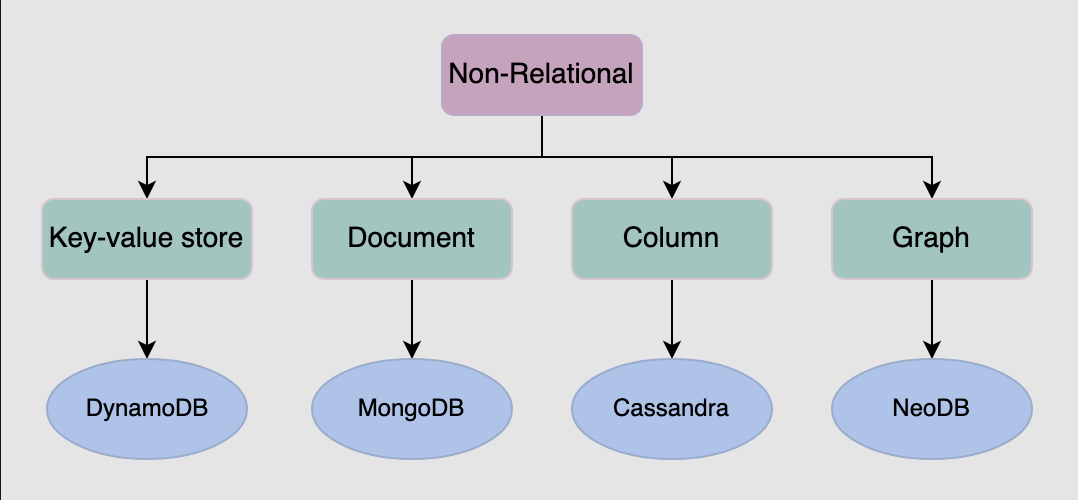
This can be achieved by relaxing some of the data consistency restrictions of other databases. Following are some characteristics of the NoSQL database:

* **Simple design:** Unlike relational databases, NoSQL doesn’t require dealing with the impedance mismatch—for example, storing all the employees’ data in one document instead of multiple tables that require join operations. This strategy makes it simple and easier to write less code, debug, and maintain.
* **Horizontal scaling:** Primarily, NoSQL is preferred due to its ability to run databases on a large cluster. This solves the problem when the number of concurrent users increases. NoSQL makes it easier to scale out since the data related to a specific employee is stored in one document instead of multiple tables over nodes. NoSQL databases often spread data across multiple nodes and balance data and queries across nodes automatically. In case of a node failure, it can be transparently replaced without any application disruption.
* **Availability:** To enhance the availability of data, node replacement can be performed without application downtime. Most of the non-relational databases’ variants support data replication to ensure high availability and disaster recovery.
* **Support for unstructured and semi-structured data:** Many NoSQL databases work with data that doesn’t have schema at the time of database configuration or data writes. For example, document databases are structureless; they allow documents (JSON, XML, BSON, and so on) to have different fields. For example, one JSON document can have fewer fields than the other.
* **Cost:** Licenses for many RDBMSs are pretty expensive, while many NoSQL databases are open source and freely available. Similarly, some RDBMSs rely on costly proprietary hardware and storage systems, while NoSQL databases usually use clusters of cheap commodity servers.

NoSQL databases are divided into various categories based on the nature of the operations and features, including document store, columnar database, key-value store, and graph database. We’ll discuss each of them along with their use cases from the system design perspective in the following sections.

### **Types of NoSQL databases**

Various types of NoSQL databases are described below:



The types of NoSQL databases

#### **Key-value database**

**Key-value databases** use key-value methods like hash tables to store data in key-value pairs. We can see this depicted in the figure a couple of paragraphs below. Here, the key serves as a unique or primary key, and the values can be anything ranging from simple scalar values to complex objects. These databases allow easy partitioning and horizontal scaling of the data. Some popular key-value databases include Amazon DynamoDB, Redis, and Memcached DB.

**Use case**: Key-value databases are efficient for session-oriented applications. Session oriented-applications, such as web applications, store users’ data in the main memory or in a database during a session. This data may include user profile information, recommendations, targeted promotions, discounts, and more. A unique ID (a key) is assigned to each user’s session for easy access and storage. Therefore, a better choice to store such data is the key-value database.

The following figure shows an example of a key-value database. The Product ID and Type of the item are collectively considered as the primary key. This is considered as a key for this key-value database. Moreover, the schema for storing the item attributes is defined based on the nature of the item and the number of attributes it possesses.

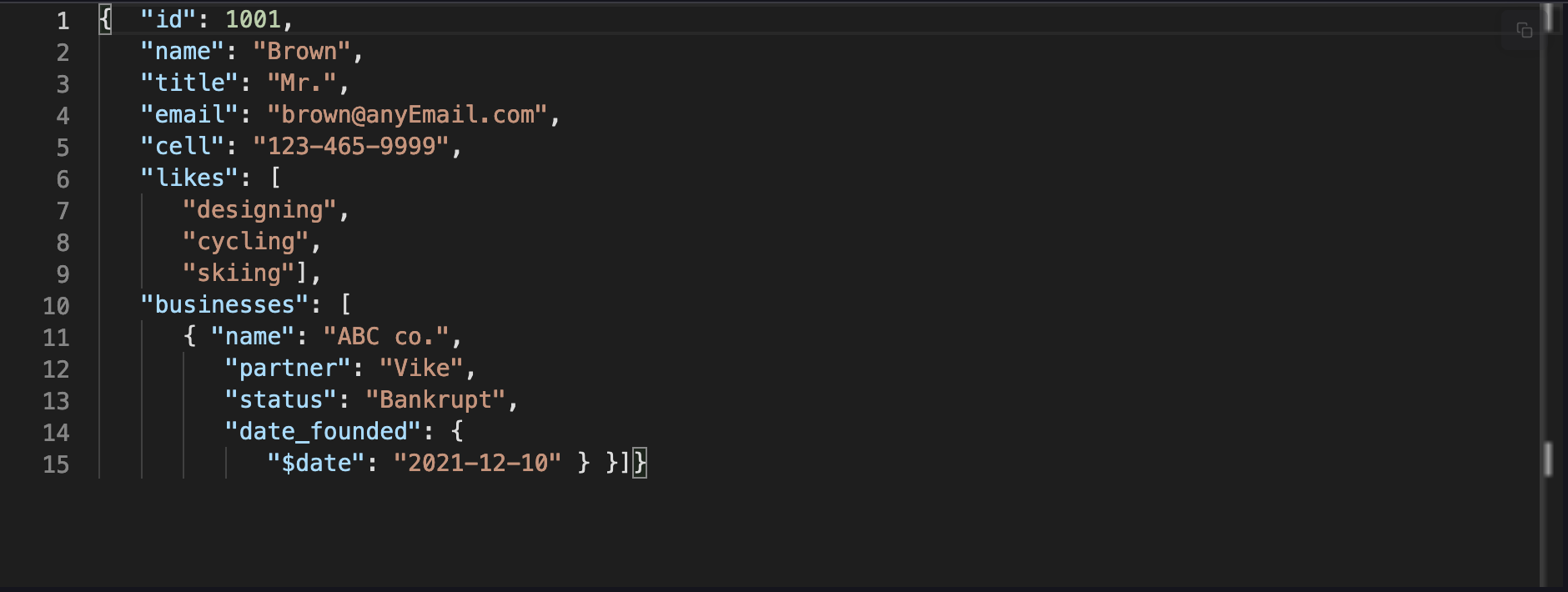
Data stored in the form of key-value pair in DynamoDB, where the key is the combination of two attributes (Product ID and Type)

#### **Document database**

A **document database** is designed to store and retrieve documents in formats like XML, JSON, BSON, and so on. These documents are composed of a hierarchical tree data structure that can include maps, collections, and scalar values. Documents in this type of database may have varying structures and data. MongoDB and Google Cloud Firestore are examples of document databases.

**Use case**: Document databases are suitable for unstructured catalog data, like JSON files or other complex structured hierarchical data. For example, in e-commerce applications, a product has thousands of attributes, which is unfeasible to store in a relational database due to its impact on the reading performance. Here comes the role of a document database, which can efficiently store each attribute in a single file for easy management and faster reading speed. Moreover, it’s also a good option for content management applications, such as blogs and video platforms. An entity required for the application is stored as a single document in such applications.

The following example shows data stored in a JSON document. This data is about a person. Various attributes are stored in the file, including id, name, email, and so on.

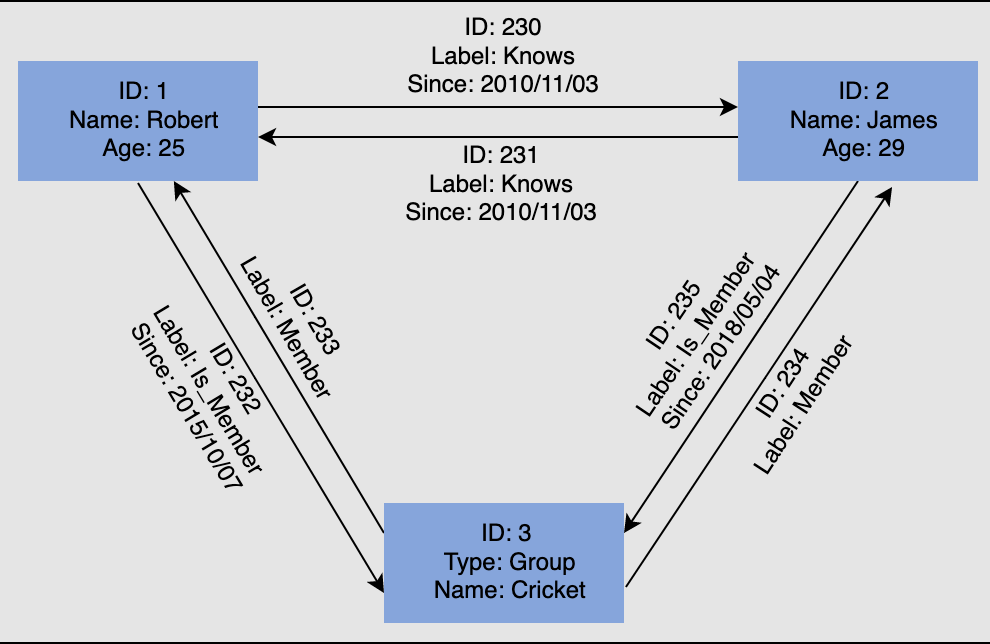


A JSON file containing data of a businessman

#### **Graph database#**

**Graph databases** use the graph data structure to store data, where nodes represent entities, and edges show relationships between entities. The organization of nodes based on relationships leads to interesting patterns between the nodes. This database allows us to store the data once and then interpret it differently based on relationships. Popular graph databases include Neo4J, OrientDB, and InfiniteGraph. Graph data is kept in store files for persistent storage. Each of the files contains data for a specific part of the graph, such as nodes, links, properties, and so on.

In the following figure, some data is stored using a graph data structure in nodes connected to each other via edges representing relationships between nodes. Each node has some properties, like Name, ID, and Age. The node having ID: 2 has the Name of James and Age of 29 years.



A graph consists of nodes and links. This graph captures entities and their relationships with each other

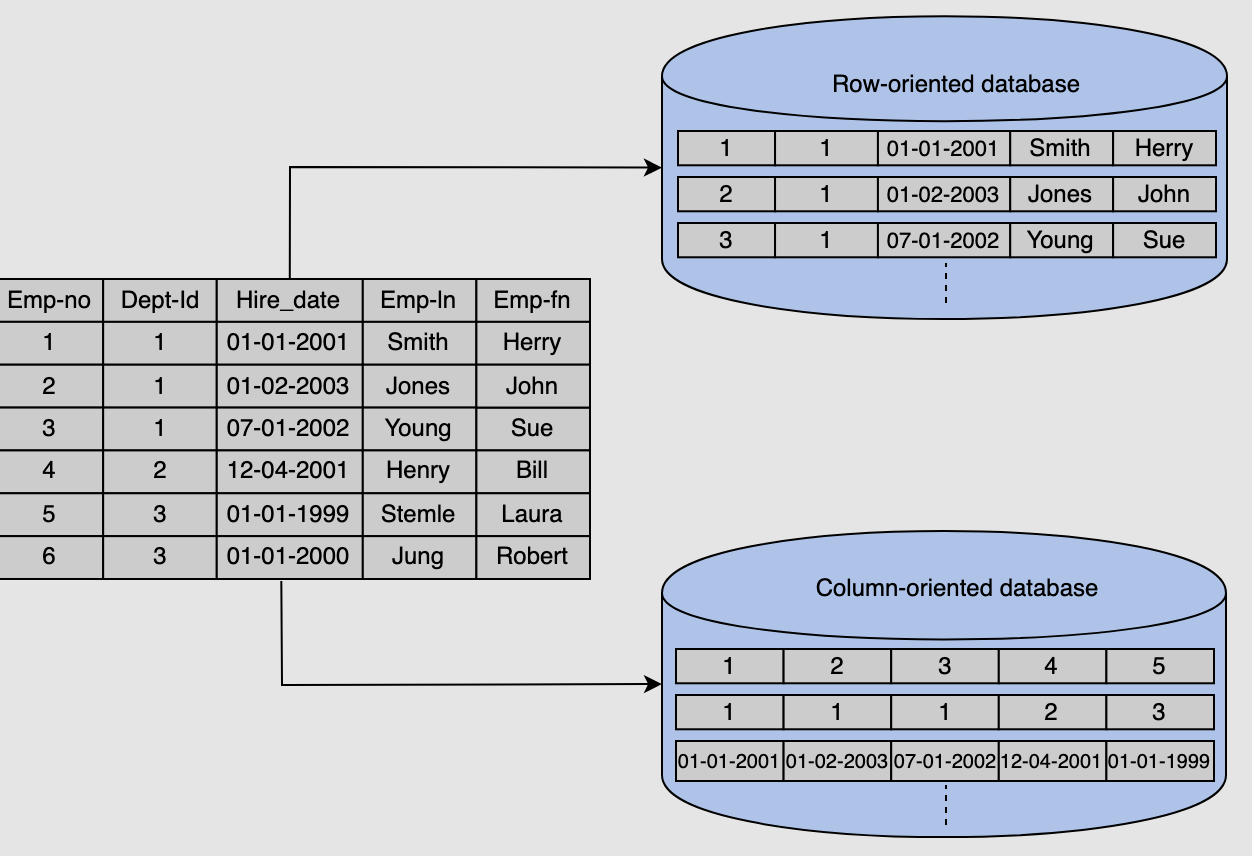
**Use case**: Graph databases can be used in social applications and provide interesting facts and figures among different kinds of users and their activities. The focus of graph databases is to store data and pave the way to drive analyses and decisions based on relationships between entities. The nature of graph databases makes them suitable for various applications, such as data regulation and privacy, machine learning research, financial services-based applications, and many more.

#### **Columnar database**

**Columnar databases** store data in columns instead of rows. They enable access to all entries in the database column quickly and efficiently. Popular columnar databases include Cassandra, HBase, Hypertable, and Amazon SimpleDB.

**Use case**: Columnar databases are efficient for a large number of aggregation and data analytics queries. It drastically reduces the disk I/O requirements and the amount of data required to load from the disk. For example, in applications related to financial institutions, there’s a need to sum the financial transaction over a period of time. Columnar databases make this operation quicker by just reading the column for the amount of money, ignoring other attributes of customers.

The following figure shows an example of a columnar database, where data is stored in a column-oriented format. This is unlike relational databases, which store data in a row-oriented fashion:



Column-oriented and row-oriented database

### **Drawbacks of NoSQL databases**

#### **Lack of standardization**

NoSQL doesn’t follow any specific standard, like how relational databases follow relational algebra. Porting applications from one type of NoSQL database to another might be a challenge.

#### **Consistency**

NoSQL databases provide different products based on the specific trade-offs between consistency and availability when failures can happen. We won’t have strong data integrity, like primary and referential integrities in a relational database. Data might not be strongly consistent but slowly converging using a weak model like eventual consistency.

# 3. Data Replication

### **Data Replication Fundamentals**

* **Purpose of Data Replication**: Essential for ensuring data availability, scalability, and performance across distributed systems. It involves duplicating data across multiple nodes, often geographically dispersed, to maintain system operation during partial failures such as disk, node, or network outages.

### **Replication Strategies**

* **Synchronous vs. Asynchronous Replication**:
  + **Synchronous Replication**: Requires acknowledgments from all secondary nodes before confirming data updates to clients, ensuring all replicas are up-to-date but at the cost of higher latency if a secondary node fails.
  + **Asynchronous Replication**: Primary node updates its state and acknowledges the client without waiting for secondary nodes, reducing latency but risking data loss if the primary node fails before all changes propagate.

### **Models of Data Replication**

* **Single Leader/Primary-Secondary Replication**: A single primary node handles writes and coordinates updates across secondary nodes. This model is effective for read-heavy workloads but can become a bottleneck with high write volumes.
* **Multi-Leader Replication**: Multiple primary nodes exist, allowing simultaneous writes that are synchronized across nodes, enhancing throughput and resilience but introducing complexities like conflict resolution.
* **Leaderless/Peer-to-Peer Replication**: All nodes equally handle reads and writes, optimizing write scalability and fault tolerance by distributing operations but potentially leading to inconsistencies due to concurrent writes.

### **Handling Replication Challenges**

* **Conflict Resolution**: Necessary in multi-leader setups where simultaneous updates by different primaries can lead to data conflicts.
  + **Conflict Avoidance**: Ensures all writes for a record are routed through the same primary.
  + **Last-Write-Wins**: Resolves conflicts using timestamps, though issues with clock skew can complicate this approach.
  + **Custom Logic**: Allows developers to implement application-specific conflict resolution strategies.

### **Replication Techniques**

* **Statement-Based Replication**: Executes the same SQL statements across all nodes, potentially leading to issues with non-deterministic functions.
* **Write-Ahead Log (WAL) Shipping**: Logs changes before execution, ensuring consistency but tightly coupled with database internals.
* **Logical (Row-Based) Log Replication**: Changes are replicated at the row level, providing a more granular and flexible approach.

### **Considerations for System Designers**

* The choice between replication strategies involves trade-offs between consistency, availability, performance, and complexity. System designers must consider the specific requirements and constraints of their applications to select the most appropriate replication model and techniques.

This section provides critical insights into designing robust distributed systems by effectively leveraging various data replication strategies to balance system demands and performance.

4. Data Partitioning

### **Overview of Data Partitioning**

Data partitioning is essential for managing scalability pressures on databases due to increasing concurrent read/write traffic and data volume. It involves dividing data across multiple nodes, balancing partitions and load to enhance performance.

### **Types of Sharding**

* **Vertical Sharding**: Different tables or columns are stored on different nodes or split among various tables to optimize data retrieval, especially for large text or blob columns.
* **Horizontal Sharding**: Data is divided row-wise across different tables or nodes. It can be implemented using:
  + **Key-range Based Sharding**: Each partition holds a continuous range of keys, suitable for range queries but limited to specific key usage.
  + **Hash-based Sharding**: Uses a hash function on keys to distribute them across nodes, ensuring uniform distribution but not supporting range queries.

### **Consistent Hashing**

This method places servers or items on a conceptual ring, facilitating scalable and performance-efficient data management.

### **Rebalancing Partitions**

Strategies for rebalancing include avoiding hash mod n (which can lead to extensive data movement) and maintaining a fixed number of partitions or dynamically adjusting them based on load and data volume.

### **Partitioning with Secondary Indexes**

* **By Document**: Each partition manages its indexes independently, leading to potential high latency if any partition performs poorly.
* **By Term**: Global indexes are created for terms across all partitions, enhancing read efficiency but increasing write complexity.

### **Request Routing**

Discusses mechanisms for directing client requests to the appropriate node, which can be based on direct client knowledge, a routing tier, or forwarding by nodes.

### **Role of ZooKeeper**

ZooKeeper is utilized for tracking cluster configuration changes, helping manage the allocation and information regarding node and partition statuses.

### **Conclusion**

Partitioning is fundamental in distributed system design to enhance data access speed, system scalability, and overall performance. The document covers theoretical aspects, practical scenarios like designing systems for YouTube, Quora, and Uber, and also discusses common pitfalls and successful strategies in partition design.

This detailed examination helps engineers prepare for system design interviews by understanding and applying data partitioning principles effectively.

5. Trade-offs in Databases

### **Introduction to Databases**

* The document begins by outlining the basic concepts and types of databases, such as key-value stores, content delivery networks (CDN), and distributed monitoring systems.
* It discusses the roles of components like sequencers, pub-sub systems, rate limiters, blob stores, and task schedulers in modern databases.

### **Centralized vs. Distributed Databases**

* **Centralized Databases:**
  + **Advantages:** Easier data maintenance, stronger consistency, simpler programming model, and efficiency in handling smaller data sets.
  + **Disadvantages:** Potential for high latency as query volume approaches the limits of a single node, and a higher risk of being a single point of failure.
* **Distributed Databases:**
  + **Advantages:** Faster data access by retrieving data from the nearest or most frequently used shard, ability to handle intensive queries by dividing them into multiple optimized subqueries processed in parallel.
  + **Disadvantages:** Increased complexity and time consumption when data from multiple sites is needed, especially for operations like joins which can become more expensive and complex. Maintaining data consistency across sites is also challenging and requires extra measures.

### **Sharding: Horizontal vs. Vertical**

* The document discusses the decision-making process involved in choosing between horizontal and vertical sharding based on the organization's needs, aiming to prevent downtime and reduce latency while scaling resources like CPU, memory, disk space, and network bandwidth.

### **Query Optimization in Distributed Databases**

* It delves into the specifics of query optimization, using an example involving a query that accesses tables located on different sites. The example demonstrates the computation of total communication time and compares different approaches (moving data between sites, projecting subsets of data) to optimize query processing speed.

### **Practical Examples and System Design**

* The latter part of the document applies these concepts to the design of various systems such as YouTube, Quora, Google Maps, Uber, Twitter, and Instagram. It also covers specialized designs like a proximity service (similar to Yelp), URL shortening services, web crawlers, and collaborative document editing.

### **Conclusion**

* The conclusion emphasizes the importance of considering reliability, performance, and cost when distributing data across multiple nodes. It suggests that the choice between centralized and distributed databases should be tailored to the application's specific needs.