NOSQL

Module-1 Notes

Value of Relational Databases

Relational databases have become such an embedded part of our computing culture that it's easy to take them for granted. It's therefore useful to revisit the benefits they provide. Relational databases provide robust mechanisms for ensuring persistency, consistency, concurrency, and integration. Their ability to permanently store data, maintain data integrity, manage concurrent access, and seamlessly integrate with other systems makes them invaluable in many organizational contexts.

- **Persistency-** Relational databases ensure that once a transaction is committed, the data is stored permanently, even in the event of a power failure or system crash. This durability is a fundamental aspect of the ACID properties. They provide robust backup and recovery mechanisms, allowing data to be restored to a previous state if needed. This ensures long-term data retention and reliability.
- **Consistency -** Relational databases enforce data integrity through constraints such as primary keys, foreign keys, unique constraints, and check constraints. This ensures that the data remains accurate and consistent. Transactions in relational databases adhere to the consistency property of ACID, ensuring that each transaction brings the database from one valid state to another. This prevents data corruption and maintains database rules and constraints.
- **Concurrency -** Relational databases support various isolation levels (e.g., read uncommitted, read committed, repeatable read, serializable) to manage concurrent access to the data. These isolation levels help prevent issues like dirty reads, non-repeatable reads, and phantom reads. They use sophisticated locking mechanisms to manage concurrent access to data, ensuring that multiple transactions can occur simultaneously without causing inconsistencies or data corruption. This is crucial for maintaining data integrity in multi-user environments.
- **Integration -** SQL provides a standardized way to interact with relational databases, facilitating integration with various applications, tools, and platforms. This standardization ensures that different systems can communicate with the database effectively. Relational databases can be integrated with other databases and systems through various methods such as ETL (Extract, Transform, Load) processes, database links, and APIs. This allows for seamless data exchange and integration across different platforms.
- **A (Mostly) Standard Model -** Relational databases have succeeded because they provide the core benefits we outlined earlier in a (mostly) standard way. As a result, developers and database professionals can learn the basic relational model and apply it in many projects. Although there are differences between different relational databases, the core mechanisms remain the same: Different vendors' SQL dialects are similar, transactions operate in mostly the same way.

Impedance mismatch issue of RDBMS

- The impedance mismatch issue in RDBMS refers to the conceptual and practical difficulties that arise when trying to bridge the gap between the relational data model and the object-oriented programming (OOP) paradigm. In relational databases, data is organized into tables with rows and columns, and relationships between data are represented using foreign keys. On the other hand, object-oriented programming languages like Java, C++, and Python organize data into objects, which encapsulate both data and behavior. This fundamental difference in data representation leads to challenges in translating complex data structures and relationships between the two models.

- The impedance mismatch manifests in several ways, such as difficulty in mapping object hierarchies to relational tables, managing object identity versus primary keys, and handling relationships like inheritance and polymorphism. For example, a complex object graph with nested objects and collections must be flattened into a tabular structure for storage in an RDBMS, and vice versa when retrieving data. This often requires additional code for conversion, leading to increased complexity and potential performance issues. Moreover, maintaining data consistency and integrity between the object model and the relational model can be challenging, necessitating the use of Object-Relational Mapping (ORM) frameworks, which introduce their own learning curves and potential performance overheads. These discrepancies complicate development and can impact application performance and maintainability.
- if you want to use a richer in memory data structure, you have to translate it to a relational representation to store it on disk. Hence the impedance mismatch—two different representations that require translation as shown in the following figure:

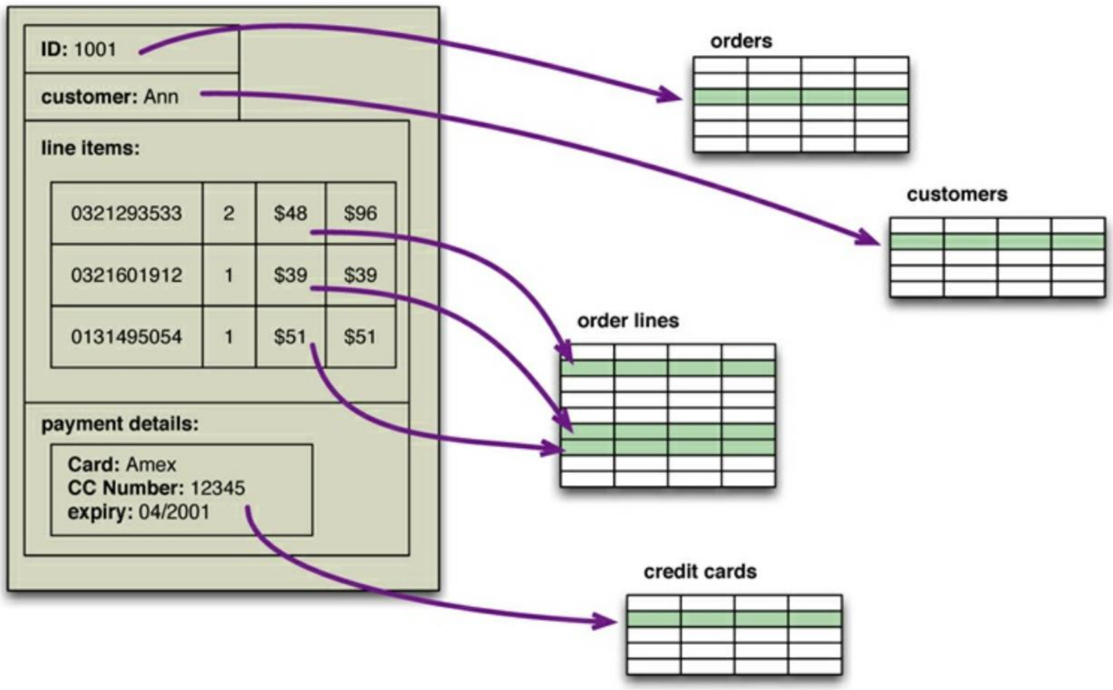


Figure 1.1. An order, which looks like a single aggregate structure in the UI, is split into many rows from many tables in a relational database

Application and Integration databases

- An integration database is a centralized repository designed to store and manage data from multiple applications, often developed by separate teams. This database serves as a common data source, ensuring that different applications within an organization can access and share consistent and up-to-date information. By consolidating data from various systems, the integration database eliminates data silos and facilitates seamless data flow between applications.

- This setup supports efficient data integration, enabling cross-functional teams to collaborate more effectively and make informed decisions based on a unified dataset. Moreover, it enhances data integrity and consistency by providing a single point of truth, while also simplifying data governance and security management. The integration database plays a critical role in complex IT environments, where interoperability and real-time data access are essential for operational efficiency and strategic planning.
- An application database is designed to support a specific application, serving as the dedicated repository for all data that the application needs to function. This database is typically accessed exclusively by the application's codebase, ensuring a tightly coupled relationship between the application and its data. The single-team management approach allows for highly customized schema design and optimization tailored to the application's unique requirements.
- This includes structuring the data in a way that maximizes performance for the application's most frequent queries and transactions. The dedicated focus also enables quick iterations and adjustments to the database schema and indexing strategies in response to evolving application needs, ensuring that the database remains aligned with the application's performance and functionality goals.
- By being managed by a single team, the application database benefits from streamlined communication and coordination, reducing the complexity that often arises when multiple teams are involved.
- Furthermore, this dedicated approach simplifies security management, as the team can implement and enforce specific access controls and data protection measures tailored to the application's requirements. Overall, the application database's exclusive access and single-team management create a highly efficient, responsive, and secure data environment that is closely aligned with the application's operational needs.

Attack of the clusters

- This increase in scale was happening along many dimensions. Websites started tracking activity and structure in a very detailed way. Large sets of data appeared: links, social networks, activity in logs, mapping data**.**
- Approaches to handle this large traffic and dataset: Scaling up, also known as vertical scaling, involves adding more resources to an existing server or machine to handle increased load. This means upgrading the server's hardware capabilities, such as increasing the CPU power, adding more RAM, or enhancing storage capacity.
- Scaling out, also known as horizontal scaling, involves adding more machines or nodes to a system, distributing the load across multiple servers. This approach typically requires a distributed architecture where the workload is shared among various servers.
- A cluster of small machines can use commodity hardware and ends up being cheaper at these kinds of scales. It can also be more resilient—while individual machine failures are common, the overall cluster can be built to keep going despite such failures, providing high reliability.
- Relational databases are traditionally designed for single-machine environments, which makes scaling them across clusters challenging. The primary issue lies in maintaining ACID properties—ensuring atomicity, consistency, isolation, and durability—across multiple nodes. Distributing data via sharding adds complexity, as it can lead to data hotspots and uneven load distribution. Ensuring transactional integrity and consistency in a distributed setup often requires complex coordination mechanisms, which can introduce latency and reduce overall system performance. Additionally, the inherent design of relational databases to use joins and complex queries becomes less efficient

when data is spread across different nodes, further complicating the use of relational databases in clustered environments.

- This mismatch between relational databases and clusters led some organization to consider anlternative route to data storage. Two companies in particular—Google and Amazon—have been very influential. Both were on the forefront of running large clusters of this kind; furthermore, they were capturing huge amounts of data.

Emergence of NOSQL

- The emergence of NoSQL databases addresses many of the limitations of traditional relational databases, especially in handling large-scale, distributed, and unstructured data. NoSQL databases are designed to provide flexible schemas and horizontal scalability, making them well-suited for modern applications that require quick access to large volumes of diverse data. Unlike relational databases, which rely on a rigid schema and ACID transactions, NoSQL databases often sacrifice some aspects of consistency to achieve higher availability and partition tolerance, as per the CAP theorem.
- NoSQL databases come in various types, including document stores (e.g., MongoDB), key-value stores (e.g., Redis), column-family stores (e.g., Cassandra), and graph databases (e.g., Neo4j), each optimized for specific use cases. They allow for efficient storage and retrieval of unstructured or semi-structured data and are particularly effective in handling large-scale, distributed data environments typical of big data applications. The flexibility, scalability, and performance of NoSQL databases have made them popular in industries such as social media, e-commerce, and real-time analytics, where traditional RDBMS solutions struggle to meet the demands of modern, high-velocity data processing.
- The term "NoSQL" originally emerged to describe databases that do not adhere to the traditional relational database model, particularly those that do not use Structured Query Language (SQL) for data management. While "NoSQL" is often interpreted as "no SQL," it more accurately conveys "not only SQL," highlighting that these databases can support various data models and query languages beyond the standard relational approach. The name reflects a broad category of database systems designed to handle unstructured, semi-structured, and rapidly changing data with greater flexibility and scalability than traditional relational databases.

Features of NoSQL

The key characteristics of NoSQL databases that highlight their distinct advantages over traditional relational databases:

1. Schema Flexibility

NoSQL databases allow for dynamic and flexible schemas, enabling users to store unstructured, semi-structured, or structured data without predefined schemas. This flexibility allows for easier adaptation to changing data requirements and makes it simpler to incorporate new data types without extensive modifications to the database.

2. Horizontal Scalability

NoSQL databases are designed to scale out horizontally by adding more servers or nodes to distribute data across multiple machines. This scalability allows them to handle large volumes of data and high levels of concurrent user requests, making them well-suited for applications with rapidly growing datasets.

3. High Availability and Fault Tolerance

Many NoSQL databases are built to ensure high availability and fault tolerance. They often employ data replication across multiple nodes, enabling continuous operation even if some nodes fail. This design ensures that the database remains accessible and resilient in the face of hardware failures or other disruptions.

4. Support for Various Data Models

NoSQL databases support multiple data models, including document, key-value, column-family, and graph models. This variety allows developers to choose the most appropriate model based on their specific use cases, optimizing data storage and retrieval according to the needs of their applications.

5. Eventual Consistency

Unlike traditional relational databases that emphasize strong consistency through ACID transactions, many NoSQL databases adopt an eventual consistency model. This approach allows for higher availability and partition tolerance, as data updates may not be immediately consistent across all nodes. Instead, the system guarantees that, given enough time, all updates will propagate throughout the database, making it suitable for distributed environments where immediate consistency is less critical.

Polyglot persistence refers to the practice of using multiple data storage technologies, each optimized for specific use cases, within a single application or system architecture. In the context of NoSQL databases, this approach allows developers to leverage the strengths of various NoSQL database types—such as document stores, key-value stores, column-family stores, and graph databases—to handle diverse data needs efficiently.

Aggregate data models

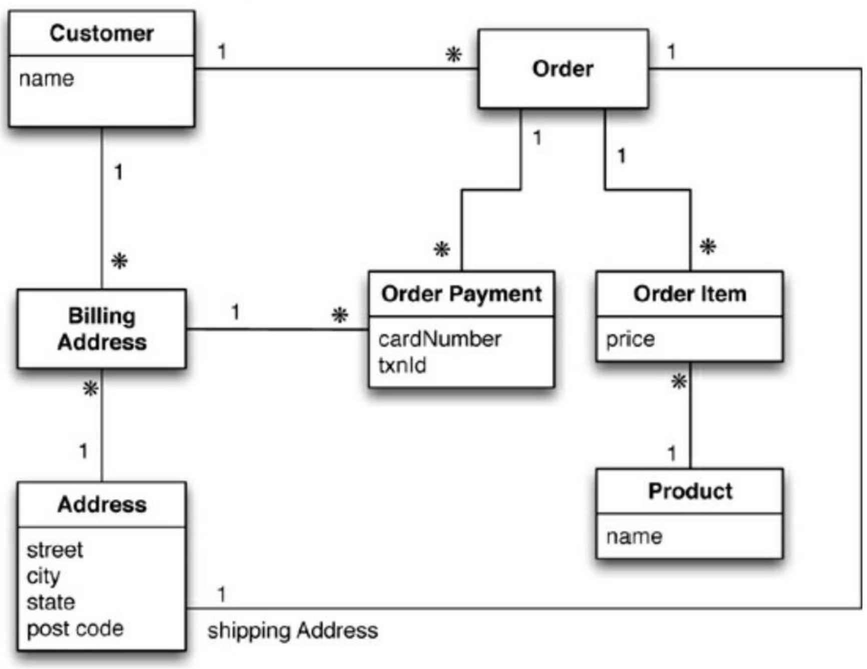
Aggregate data models are a foundational concept in NoSQL databases, particularly in document-oriented and key-value stores, where they enable the grouping of related data into cohesive units known as aggregates. An aggregate typically encompasses all the data related to a specific entity or concept, encapsulated in a single structure to ensure data integrity and simplify retrieval. For example, in a document database, a user profile might be stored as a single document containing all relevant information—such as personal details, preferences, and activity history—rather than distributing it across multiple tables or collections. This approach enhances performance by reducing the number of database calls needed to access related data, thereby minimizing latency.

Aggregates

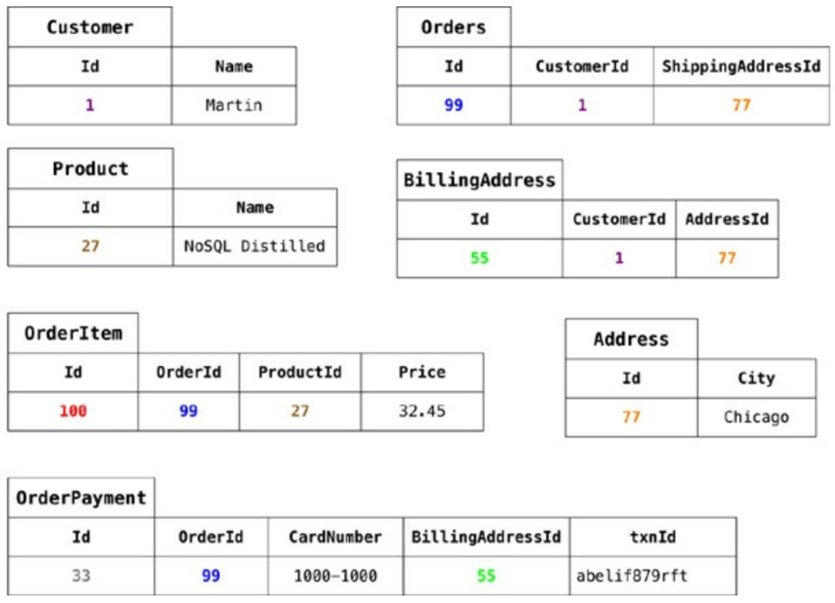
- Aggregate is a collection of related objects that we wish to treat as a unit. In particular, it is a unit for data manipulation and management of consistency.
- Typically, we like to update aggregates with atomic operations and communicate with our data storage in terms of aggregates. This definition matches really well with how key-value, document, and column-family databases work.
- Dealing in aggregates makes it much easier for these databases to handle operating on a cluster, since the aggregate makes a natural unit for replication and sharding.

Explanation of the need of aggregate data model :

Consider an e-commerce website; we are going to be selling items directly to customers over the web, and we will have to store information about users, our product catalog, orders, shipping addresses, billing addresses, and payment data. A pure RDBMS model would look like following figure:



Data as per the RDBMS design is shown in the following example figure:



Customer

Id	Name
1	Martin

Orders

Id	CustomerId	ShippingAddressId
99	1	77

Product

Id	Name
27	NoSQL Distilled

BillingAddress

Id	CustomerId	AddressId
55	1	77

OrderItem

Id	OrderId	ProductId	Price
100	99	27	32.45

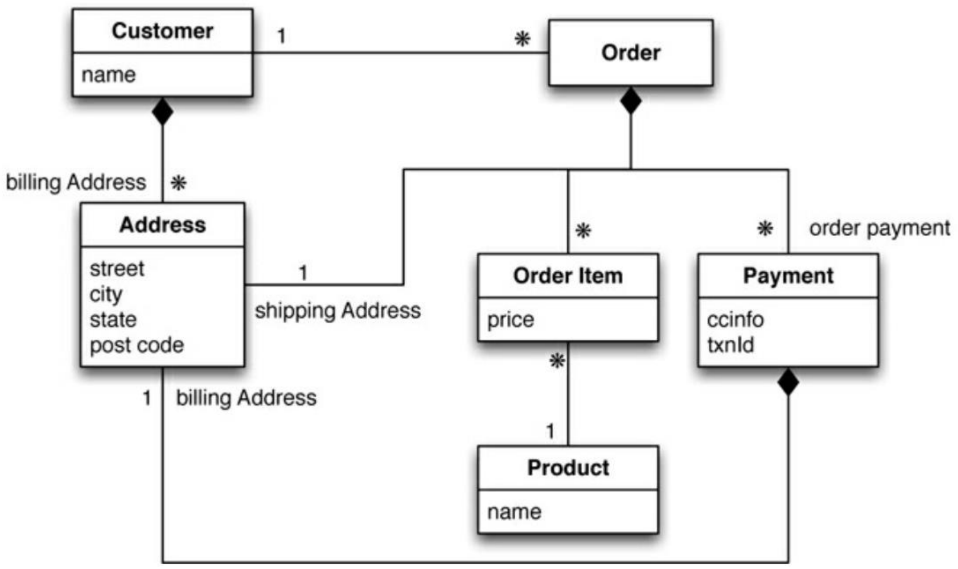
Address

Id	City
77	Chicago

OrderPayment

Id	OrderId	CardNumber	BillingAddressId	txnId
33	99	1000-1000	55	abelif879rft

everything is properly normalized, so that no data is repeated in multiple tables. We also have referential integrity.
An aggregate model for the same example is shown in the following figure:



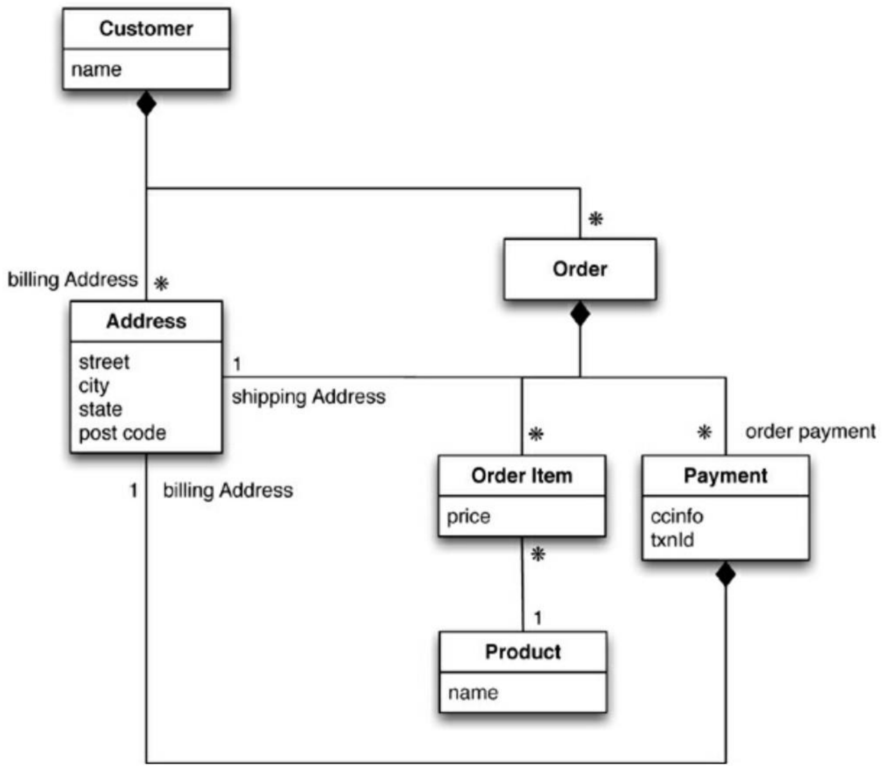
A single logical address record appears three times in the example data, but instead of using IDs it's treated as a value and copied each time. This fits the domain where we would not want the shipping address, nor the payment's billing address, to change. In a relational database, we would ensure that the address rows aren't updated for this case, making a new row instead. With aggregates, we can copy the whole address structure into the aggregate as we need to.

A typical NoSQL code for the same is:

```
// in customers
{
"id":1,
"name":"Martin",
"billingAddress":[{"city":"Chicago"}]
}

// in orders
{
"id":99,
"customerId":1,
"orderItems":[
  {
  "productId":27,
  "price": 32.45,
  "productName": "NoSQL Distilled"
    }
  ],
"shippingAddress":[{"city":"Chicago"}]
"orderPayment":[
  {
    "ccinfo":"1000-1000-1000-1000",
    "txnId":"abelif879rft",
    "billingAddress": {"city": "Chicago"}
  }
  ],
}
```

The link between the customer and the order isn't within either aggregate—it's a relationship between aggregates. Similarly, the link from an order item would cross into a separate aggregate structure for products. It is more common with aggregates because we want to minimize the number of aggregates we access during a data interaction. The more embedded aggregate model is shown in the following figure:



Updated NOSQL code is:

```
// in customers
{
"customer": {
"id": 1,
"name": "Martin",
"billingAddress": [{"city": "Chicago"}],
"orders": [
  {
    "id":99,
    "customerId":1,
    "orderItems":[
    {
    "productId":27,
    "price": 32.45,
    "productName": "NoSQL Distilled"
    }
  ],
  "shippingAddress":[{"city":"Chicago"}]
  "orderPayment":[
    {
    "ccinfo":"1000-1000-1000-1000",
    "txnId":"abelif879rft",
    "billingAddress": {"city": "Chicago"}
    }],
```

```
  }]
}
}
```

Consequences of Aggregate Orientation

Consequences of Aggregate Orientation for RDBMS:

- Relational databases typically require a fixed schema, which can lead to complex designs when trying to model aggregates. This can result in numerous tables and relationships, making it difficult to manage and evolve the schema over time.
- Aggregates often span multiple tables in RDBMS, necessitating complex JOIN operations to retrieve related data. This can lead to performance bottlenecks, especially when dealing with large datasets or frequent queries.
- The need for multiple queries to fetch related data can degrade performance in relational databases, particularly in scenarios where quick access to aggregate data is essential, such as in real-time applications.
- Maintaining ACID properties across multiple tables can introduce additional overhead, making transactions more complex and potentially impacting throughput and latency.

Consequences of Aggregate Orientation for NoSQL:

- NoSQL databases allow for storing aggregates in a single structure (e.g., a document), enabling straightforward and efficient retrieval of all related data in one query, thus improving performance.
- The schema-less nature of many NoSQL databases accommodates dynamic data models, making it easier to adapt to changing requirements without extensive schema alterations.
- NoSQL databases are designed to handle large volumes of data and high levels of concurrent requests, allowing them to scale efficiently while managing aggregates.
- By encapsulating related data within a single aggregate, NoSQL databases minimize or eliminate the need for complex JOIN operations, leading to faster query performance.
- Many NoSQL systems adopt an eventual consistency model, which can enhance performance and availability but may introduce challenges in maintaining consistency across aggregates in distributed environments.

Key-value and Document Data models

- In a key-value data model, data is stored as a collection of key-value pairs, where each key is unique and maps to a single value. The value can be simple data types (like strings or integers) or complex data structures (like lists or sets), but it is not directly accessible by structure—only by the key.
- Key-value stores offer high performance and low latency for read and write operations, making them ideal for caching, session management, and real-time analytics, where quick access to data is crucial.
- In a document data model, data is stored in documents, typically in JSON or BSON format, which allows for nested structures and varying fields within each document. Each document is self-describing, making it easier to understand and manipulate.
- The document model supports complex data types, including arrays and nested objects, allowing for richer data representation and better alignment with object-oriented programming paradigms.
- Document stores provide more advanced querying capabilities than key-value stores, allowing for searches based on the contents of documents, indexing on specific fields, and aggregation operations. This makes them suitable for applications that require complex queries and analytics.

Column Family stores

- Column family stores are a type of NoSQL database designed to store data in a format optimized for distributed data architectures, particularly in handling large volumes of structured data.
- In column family stores, data is organized into column families, which are collections of related columns that are grouped together. Each row within a column family can have a different set of columns, allowing for a more flexible schema compared to traditional relational databases.
- Column family stores are particularly effective at storing sparse data, where not all rows have the same set of columns.
- Column family stores are optimized for read and write performance, especially for queries that involve large datasets
- Popular column family stores include Apache Cassandra, HBase (built on top of Hadoop), and ScyllaDB. Each of these systems leverages the column-family data model to deliver high performance and scalability for big data applications.
- Row-oriented and column-oriented stores are two fundamental approaches to organizing and storing data in databases, each optimized for different use cases. Row-oriented stores, like traditional relational databases, store data in rows, meaning all attributes of a single record are stored together. This format is particularly efficient for transactional workloads that require quick access to complete records, making it ideal for online transaction processing (OLTP) applications.
- Conversely, column-oriented stores organize data by columns, storing all values for a single attribute together. This structure is optimized for analytical queries that often require aggregating or filtering large datasets by specific columns, making it suitable for online analytical processing (OLAP) applications. As a result, while row-oriented databases excel in read-write operations involving complete records, column-oriented stores provide better performance for complex queries and reporting, particularly when only a subset of columns is needed.
- Skinny rows refer to database rows that contain a small number of columns or attributes, resulting in a compact data structure. This design typically means that each record stores only essential information, which can lead to efficient storage and quick retrieval.
- Wide rows, on the other hand, refer to database rows that contain a large number of columns or attributes. This design is common in scenarios where a single entity or record needs to store a significant amount of related information, often leading to rows that are considerably wider than typical.

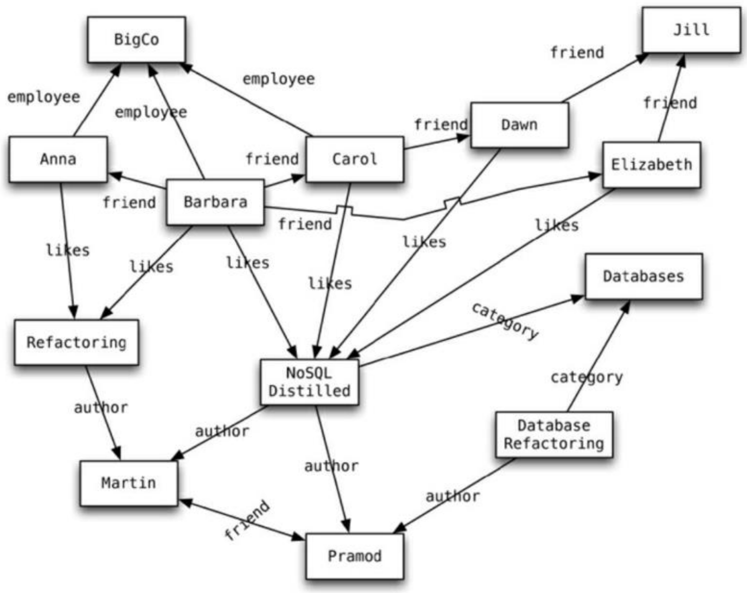
Aggregate Relationships

- Aggregate relationships are a key concept in NoSQL databases, particularly in document-oriented and key-value stores, where data is grouped into aggregates that represent a cohesive unit of related information.
- Aggregate relationships define how different pieces of data within an aggregate relate to one another. An aggregate is a cluster of related data that can be treated as a single unit, encapsulating all the necessary information needed to describe a specific entity or concept, such as a user profile, order, or product.
- In the context of aggregate relationships, an **aggregate root** is the primary entity through which the aggregate is accessed and modified. It serves as the entry point for operations and ensures that all changes to the aggregate maintain its integrity.
- Aggregate oriented databases treat the aggregate as the unit of data-retrieval. Consequently, atomicity is only supported within the contents of a single aggregate. If

you update multiple aggregates at once, you have to deal yourself with a failure partway through. Relational databases help you with this by allowing you to modify multiple records in a single transaction, providing ACID guarantees while altering many rows.

Graph Databases

- Graph databases are a type of NoSQL database specifically designed to represent and store data in graph structures, which consist of nodes (entities) and edges (relationships) connecting them.
- This model allows for the representation of complex relationships and interconnected data in a way that is both intuitive and efficient. Unlike traditional relational databases, which rely on tables and foreign keys to establish relationships, graph databases treat relationships as first-class citizens.
- They excel in scenarios where relationships are as important as the data itself, such as social networks, recommendation systems, fraud detection, and network analysis.
- One of the key advantages of graph databases is their ability to perform complex queries on relationships with high efficiency. By leveraging graph traversal algorithms, they can quickly retrieve interconnected data without the need for expensive JOIN operations typical in relational databases.
- This capability allows for real-time analytics and insights into data relationships, enabling applications to provide richer, context-driven user experiences. Popular graph databases include Neo4j, Amazon Neptune, and ArangoDB, each offering unique features and optimizations for handling graph data.
- As data continues to grow in complexity and interconnectivity, graph databases are increasingly recognized as essential tools for managing and analyzing relational data at scale.
- An example Graph structure is shown in the following figure:



- With this structure, we can ask questions such as “find the books in the Databases category that are written by someone whom a friend of mine likes.” Graph databases specialize in capturing this sort of information—but on a much larger scale than a readable diagram could capture. This is ideal for capturing any data consisting of complex relationships such as social networks, product preferences.

- The data model of a graph database is fundamentally centered around two primary components: **nodes** and **edges**. Nodes represent entities or objects, such as users, products, or locations, while edges define the relationships between these entities, indicating how they are interconnected.
- Each node and edge can have associated properties, which are key-value pairs that provide additional context or attributes to the entities and relationships. For example, in a social network graph database, a node might represent a user with properties like name and age, while an edge could represent a "follows" relationship with properties such as the date the connection was made.
- This flexible structure allows graph databases to efficiently model complex, interconnected data and perform sophisticated queries that explore relationships, making them particularly effective for applications requiring insights into data relationships, such as social networks, recommendation engines, and fraud detection.

Schemaless databases

- Schemaless databases, often associated with NoSQL systems, are designed to allow for flexible data storage without the constraints of a predefined schema. This characteristic enables users to store unstructured, semi-structured, or structured data in a way that can evolve over time without requiring significant alterations to the database design. Here are some key features and advantages of schemaless databases:
- They allow developers to add or modify fields within data records without needing to update the entire database schema.
- Because there are no strict schema requirements, schemaless databases can easily integrate various data types and formats. This characteristic makes them suitable for handling data from multiple sources, such as JSON, XML, or even binary data, allowing for the aggregation of diverse datasets within a single system.
- The lack of a fixed schema facilitates faster development cycles, enabling teams to iterate quickly as they build and refine applications.
- Schemaless databases are often designed to scale horizontally, meaning they can distribute data across multiple servers or nodes. This scalability is especially beneficial for applications experiencing rapid growth or fluctuating workloads.
- Popular examples of schemaless databases include document stores like MongoDB, key-value stores like Redis, and wide-column stores like Cassandra. These databases leverage their schemaless nature to provide developers with the flexibility needed to handle a wide range of applications, from content management systems to real-time analytics.
- A schemaless store also makes it easier to deal with **nonuniform data**: data where each record has a different set of fields. Eg code to parse such schemaless stores:

```
//pseudo code
  foreach (Record r in records) {
  foreach Field f in r.fields) {
     print(f.name, f.value)
  }}
```

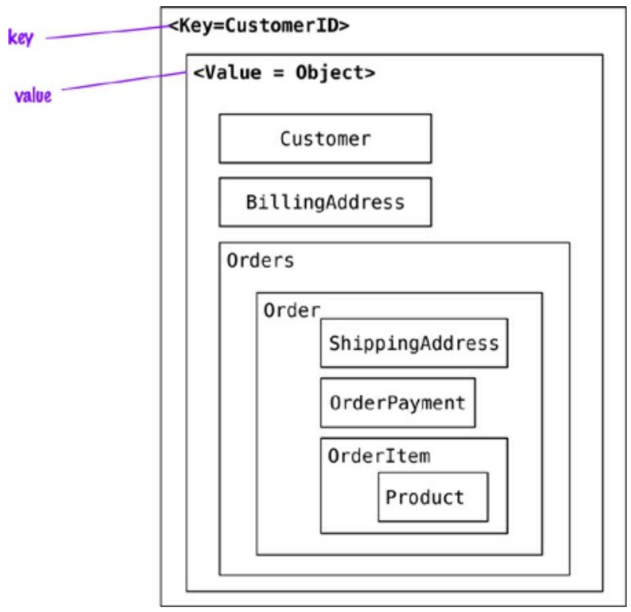
Materialized views

- Materialized views in RDBMS are pre-computed query results stored as database objects. Unlike regular views, which are virtual and calculated on-the-fly each time they are accessed, materialized views store the results physically on disk, allowing for faster query performance, especially for complex aggregations or joins that involve large datasets.

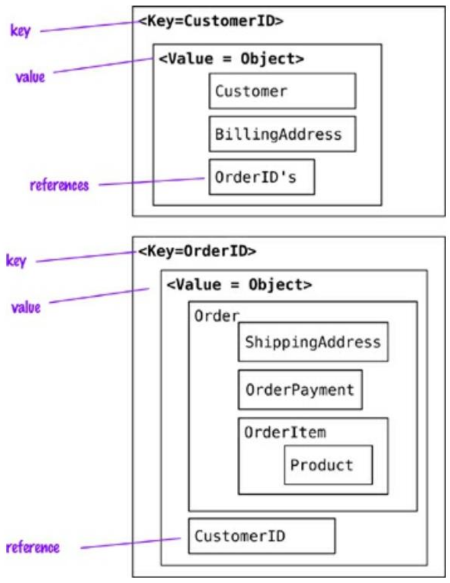
- By refreshing these views at specified intervals, either on demand or automatically, RDBMS can optimize read operations, significantly reducing the load on the underlying tables during heavy read queries.
- Materialized views are particularly useful in scenarios where data is frequently queried but not frequently updated, such as in reporting applications or data warehousing, where performance is critical, and users benefit from quick access to aggregated or summarized data.
- In NoSQL databases, the concept of materialized views can vary significantly, as these databases often prioritize flexibility and scalability over strict adherence to relational concepts.
- Some NoSQL systems provide mechanisms to create materialized views that automatically maintain and update summarized data based on the underlying data changes.
- This approach enables real-time analytics and reporting without the overhead of recalculating data on-the-fly. However, the implementation of materialized views in NoSQL databases may not be as standardized as in RDBMS, requiring developers to design their own strategies for maintaining and updating these views based on the specific requirements of their applications.

Modelling for Data Access

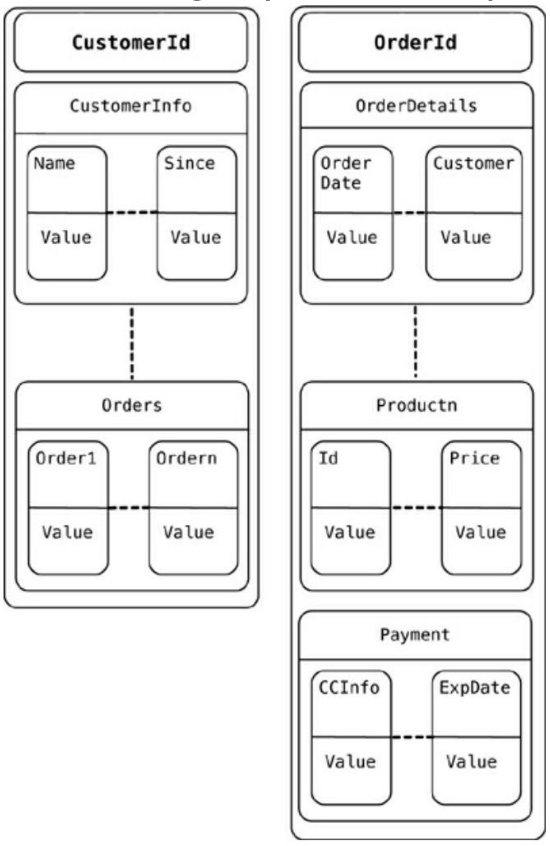
Embedding of customers and their orders using key-value stores



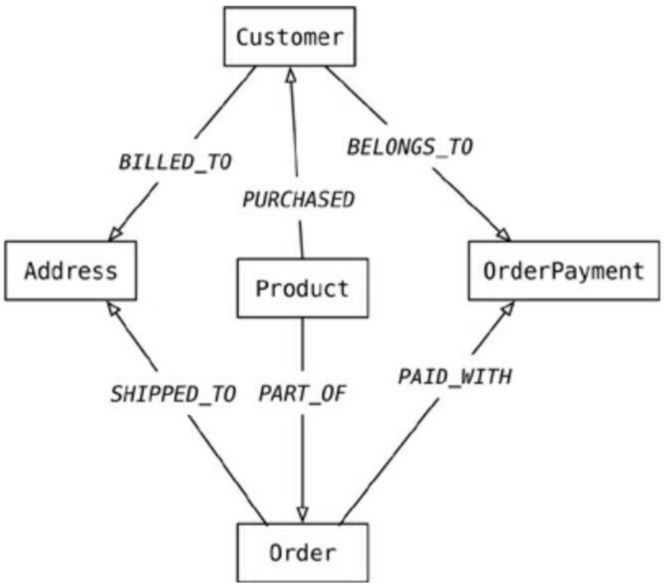
Storing of customers and orders separately



When modeling for column-family stores, we have the benefit of the columns being ordered, allowing us to name columns that are frequently used so that they are fetched first.



When using graph databases to model the same data, we model all objects as nodes and relations within them as relationships; these relationships have types and directional significance.


Customer
BILLED_TO
PURCHASED
BELONGS_TO
Address
Product
OrderPayment
SHIPPED_TO
PART_OF
PAID_WITH
Order

Module II – Notes

Distribution Models

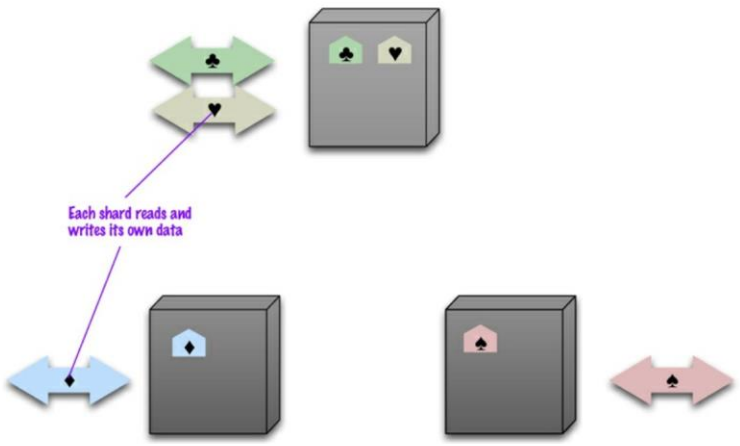
NoSQL databases employ various distribution models to manage data across multiple nodes or servers effectively. These models are designed to enhance scalability, availability, and performance, allowing NoSQL systems to handle large volumes of data and high levels of concurrent requests. Broadly, there are two paths to data distribution: replication and sharding. Replication takes the same data and copies it over multiple nodes. Sharding puts different data on different nodes. Replication and sharding are orthogonal technique. Replication comes into two forms: master-slave and peer-to-peer.

Single Server model

- The single-server model in database systems refers to a deployment architecture where all data and processing operations occur on a single server instance. This model contrasts with distributed architectures that involve multiple servers working together to manage data and handle requests.
- Although a lot of NoSQL databases are designed around the idea of running on a cluster, it can make sense to use NoSQL with a single-server distribution model if the data model of the NoSQL store is more suited to the application. Graph databases are the obvious category here—these work best in a single-server configuration.

Sharding

- Sharding is a database architecture pattern that involves partitioning data across multiple servers or nodes, enabling horizontal scaling and improving performance for large datasets. In a sharded database, data is divided into smaller, more manageable pieces called "shards," which are distributed across different servers.
- Each shard contains a subset of the data, allowing the database to handle a higher volume of read and write operations simultaneously.
- This model is particularly beneficial for applications with large datasets, as it helps mitigate performance bottlenecks and ensures that no single server is overwhelmed by excessive requests.
- A sharding example is shown in the following figure. Sharding puts different data on separate nodes, each of which does its own reads and writes

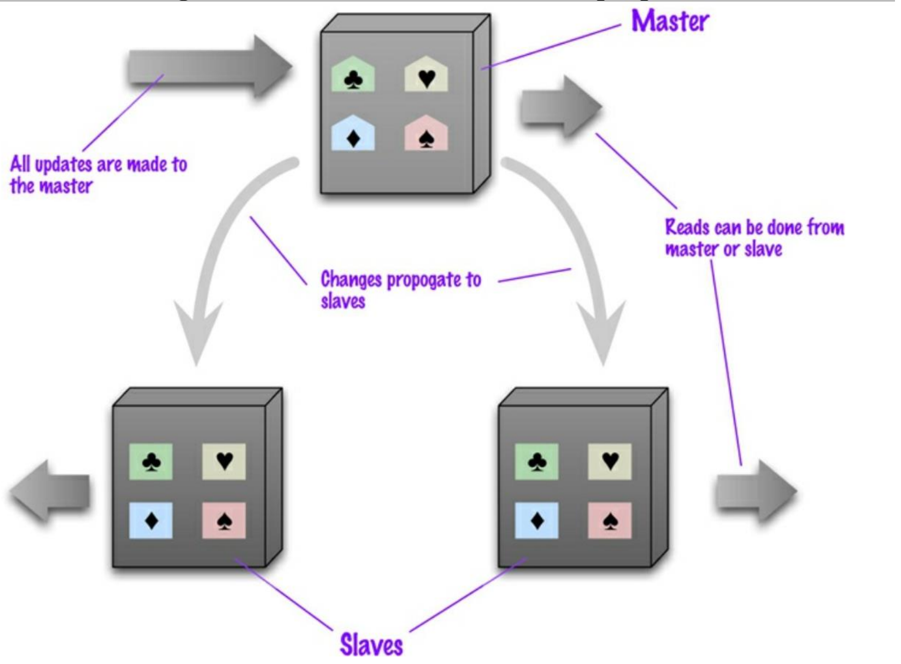


-

- Many NoSQL databases offer **auto-sharding**, where the database takes on the responsibility of allocating data to shards and ensuring that data access goes to the right shard.
- Sharding does little to improve resilience when used alone. Although the data is on different nodes, a node failure makes that shard's data unavailable just as surely as it does for a single-server solution. The resilience benefit it does provide is that only the users of the data on that shard will suffer; however, it's not good to have a database with part of its data missing.

Maste-Slave Replication

- With master-slave distribution, you replicate data across multiple nodes. One node is designated as the master, or primary. This master is the authoritative source for the data and is usually responsible for processing any updates to that data. The other nodes are slaves, or secondaries.
- An example master-slave process is shown in the following figure:

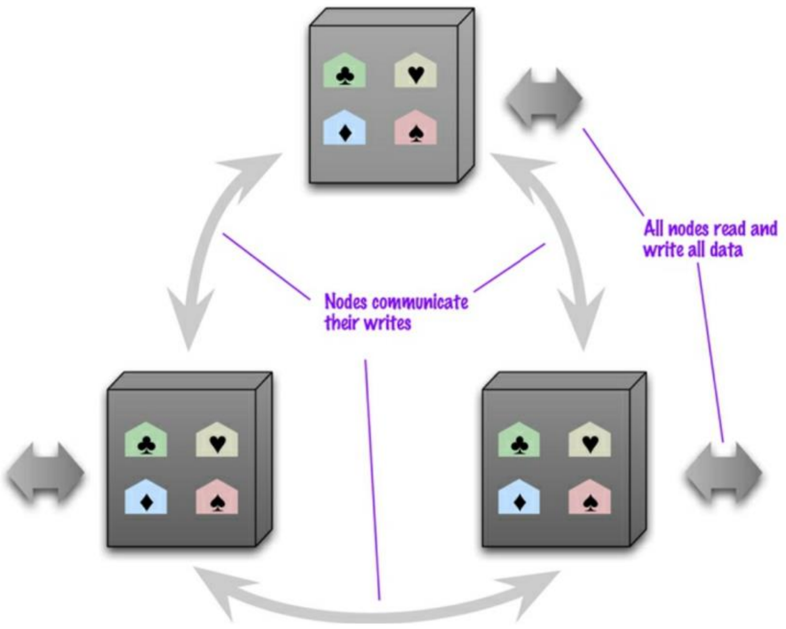

Master
All updates are made to the master
Reads can be done from master or slave
Changes propogate to slaves
Slaves

- Master-slave replication is most helpful for scaling when you have a read-intensive dataset. You can scale horizontally to handle more read requests by adding more slave nodes and ensuring that all read requests are routed to the slaves
- Read resilience in a master-slave database architecture refers to the ability of the system to effectively handle read operations even in the presence of failures or high load, leveraging the characteristics of replication and redundancy inherent in this setup.
- The ability to appoint a slave to replace a failed master means that master-slave replication is useful even if you don't need to scale out. Masters can be appointed manually or automatically. Manual appointing typically means that when you configure your cluster, you configure one node as the master. With automatic appointment, you create a cluster of nodes and they elect one of themselves to be the master.
- Replication comes with some alluring benefits, but it also comes with an inevitable dark side— inconsistency. You have the danger that different clients, reading different slaves, will see different values because the changes haven't all propagated to the slaves.

Peer-to-Peer Replication

- Peer-to-peer (P2P) replication is a distributed database architecture where each node, or peer, in the system acts as both a data source and a data sink, allowing for bidirectional data synchronization between nodes.
- Unlike traditional master-slave configurations, where one server is designated for writes and others only replicate that data, every peer in a P2P system can accept write operations and propagate changes to other peers.
- This model enhances availability and fault tolerance, as the failure of any single node does not disrupt the overall system; other peers can continue to operate and share data. P2P replication is particularly beneficial in scenarios requiring high availability and distributed access, such as in decentralized applications and environments with a large number of geographically dispersed users.

An example scenario of peer-to-peer replication is shown in the following figure:


All nodes read and write all data
Nodes communicate their writes

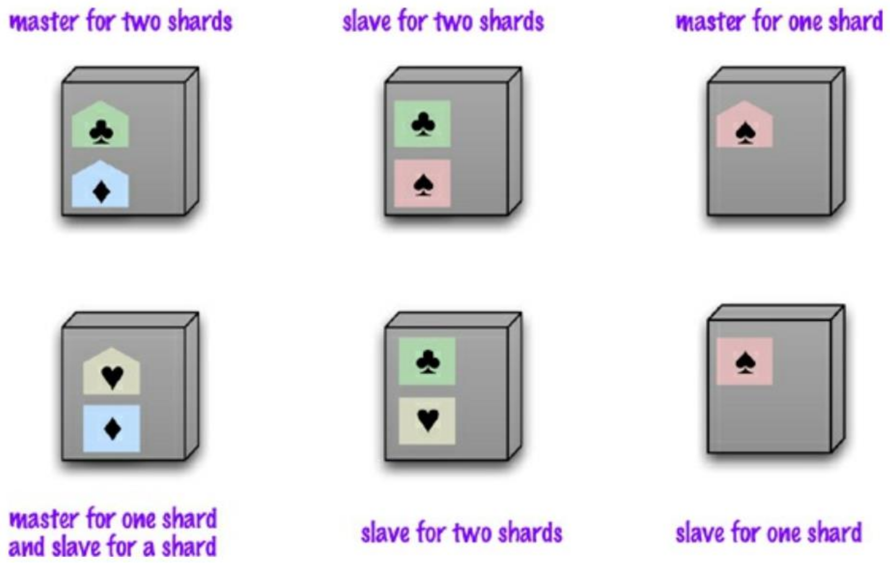
The biggest complication even with P2P replication is consistency. When you can write to two different places, you run the risk that two people will attempt to update the same record at the same time—a write-write conflict. Inconsistencies on read lead to problems but at least they are relatively transient.

Combining Sharding and Replication

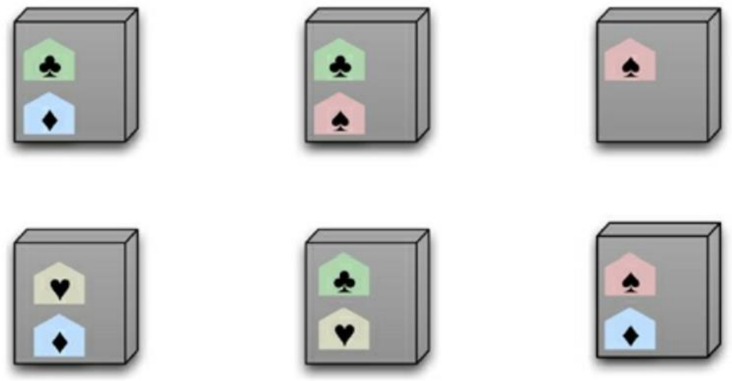
- Combining sharding and replication in a database architecture creates a robust solution that enhances both scalability and availability.
- In this hybrid model, data is partitioned into smaller, manageable shards that are distributed across multiple servers, enabling horizontal scaling to accommodate large datasets and high volumes of concurrent read and write operations. E
- ach shard can then have its own set of replicas, or slave nodes, that maintain copies of the data for redundancy. This setup allows the system to efficiently handle read requests by distributing them across multiple replicas while also ensuring that write operations are directed to the master node of each shard.
- As a result, the combination of sharding and replication effectively mitigates performance bottlenecks and improves response times, particularly for applications with varying workloads.
- If we use both master-slave replication and sharding, we have multiple masters, but each data item only has a single master. Using peer-to-peer replication and sharding is

a common strategy for column-family databases. In a scenario like this you might have tens or hundreds of nodes in a cluster with data sharded over them.

- Combining Master-Slave replication with sharding is shown in the following figure:



Combining Peer-to-peer replication with sharding is shown in the following figure:



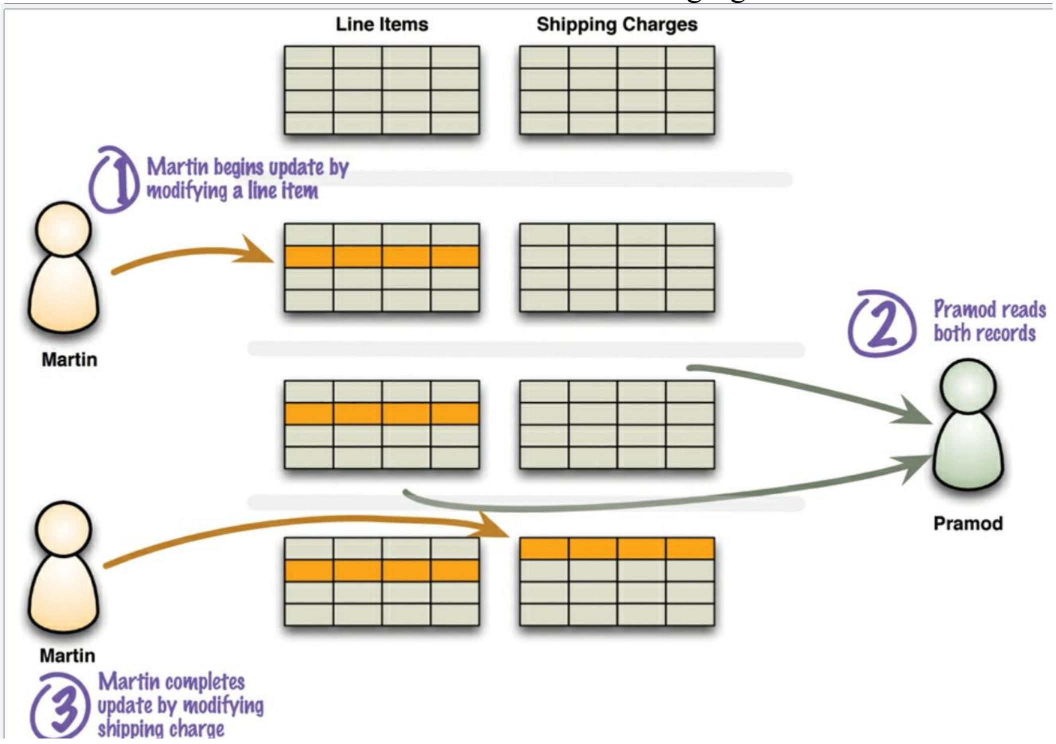
Updating Consistency

- Coincidentally, Martin and Pramod are looking at the company website and notice that the phone number is out of date. Implausibly, they both have update access, so they both go in at the same time to update the number. This issue is called a write-write conflict: two people updating the same data item at the same time.
- When the writes reach the server, the server will serialize them—decide to apply one, then the other. Let's assume it uses alphabetical order and picks Martin's update first, then Pramod's. Without any concurrency control, Martin's update would be applied and immediately overwritten by Pramod's. In this case Martin's is a lost update.
- Approaches for maintaining consistency in the face of concurrency are often described as pessimistic or optimistic. A pessimistic approach works by preventing conflicts from occurring; an optimistic approach lets conflicts occur, but detects them and takes action to sort them out.
- The most common pessimistic approach is to have write locks, so that in order to change a value you need to acquire a lock, and the system ensures that only one client can get a lock at a time.
- A common optimistic approach is a conditional update where any client that does an update to test the value just before updating it to see if it's changed since his last read.

- There is another optimistic way to handle a write-write conflict—save both updates and record that they are in conflict. Pessimistic approaches often severely degrade the responsiveness of a system to the degree that it becomes unfit for its purpose. This problem is made worse by the danger of errors—pessimistic concurrency often leads to deadlocks, which are hard to prevent and debug.

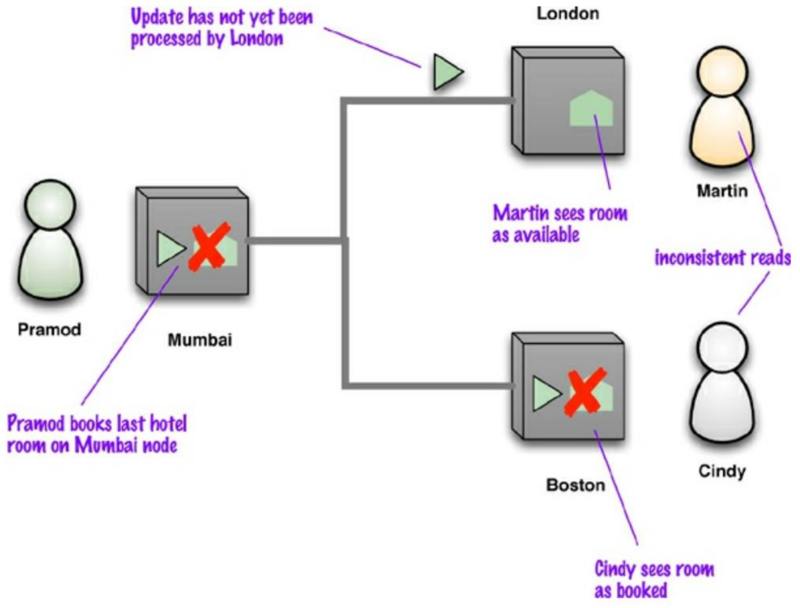
Read Consistency

Let's imagine we have an order with line items and a shipping charge. The shipping charge is calculated based on the line items in the order. If we add a line item, we thus also need to recalculate and update the shipping charge. In a relational database, the shipping charge and line items will be in separate tables. The danger of inconsistency is that Martin adds a line item to his order, Pramod then reads the line items and shipping charge, and then Martin updates the shipping charge. This is an **inconsistent read** or **read-write conflict.**
The scenario of read-write conflict is shown in the following figure:


Line Items
Shipping Charges
1 Martin begins update by modifying a line item
Martin
2 Pramod reads both records
Pramod
Martin
3 Martin completes update by modifying shipping charge

- NoSQL databases don't support transactions and thus can't be consistent. Such claim is mostly wrong because lack of transactions usually only applies to some NoSQL databases, in particular the aggregate-oriented ones. In contrast, graph databases tend to support ACID transactions just the same as relational databases. Secondly, aggregate-oriented databases do support atomic updates, but only within a single aggregate. This means that you will have logical consistency within an aggregate but not between aggregates.
- Not all data can be put in the same aggregate, so any update that affects multiple aggregates leaves open a time when clients could perform an inconsistent read. The length of time an inconsistency is present is called the **inconsistency window**. A NoSQL system may have a quite short inconsistency window.
- Replication consistency refers to the degree to which data remains consistent across multiple replicas in a distributed database system. In architectures utilizing replication—whether master-slave, peer-to-peer, or multi-master—ensuring that all copies of the data

reflect the same state is crucial for maintaining data integrity and application reliability. An example for the replication consistency is depicted in the following figure:


Update has not yet been processed by London
London
Martin
Martin sees room as available
inconsistent reads
Pramod
Mumbai
Pramod books last hotel room on Mumbai node
Boston
Cindy
Cindy sees room as booked

- **With replication, there can be eventually consistency**, meaning that at any time nodes may have replication inconsistencies but, if there are no further updates, eventually all nodes will be updated to the same value.
- One can tolerate reasonably long inconsistency windows, but you need **read your-writes consistency** which means that, once you've made an update, you're guaranteed to continue seeing that update.
- One way to get this in an otherwise eventually consistent system is to provide **session consistency**: Within a user's session there is read-your-writes consistency. This does mean that the user may lose that consistency should their session end for some reason or should the user access the same system simultaneously from different computers, but these cases are relatively rare.
- There are a couple of techniques to provide session consistency. A common way, and often the easiest way, is to have a **sticky session**: a session that's tied to one node (this is also called **session affinity**).
- A sticky session allows you to ensure that as long as you keep read-your-writes consistency on a node, you'll get it for sessions too. The downside is that sticky sessions reduce the ability of the load balancer to do its job.

Relaxing consistency

- Relaxing consistency refers to the practice of intentionally allowing some degree of inconsistency in a distributed database system to improve performance, availability, and scalability.
- In traditional database systems, strong consistency is often enforced, ensuring that all replicas reflect the same state at all times.
- However, in distributed environments, maintaining this strict consistency can introduce latency and bottlenecks, particularly during network partitions or high-traffic scenarios. By relaxing consistency, systems can achieve greater responsiveness and fault tolerance while still meeting the needs of many applications.

CAP Theorem

The CAP theorem, also known as Brewer's theorem, is a fundamental principle in distributed systems that states that it is impossible for a distributed data store to simultaneously provide all three of the following guarantees:

1. Consistency (C)

Consistency ensures that every read operation returns the most recent write for a given piece of data. In other words, all nodes in the distributed system view the same data at the same time.

2. Availability (A)

Availability guarantees that every request to the system receives a response, regardless of whether it is successful or contains the latest data. This means that the system is operational and accessible, even if some nodes are down or unreachable

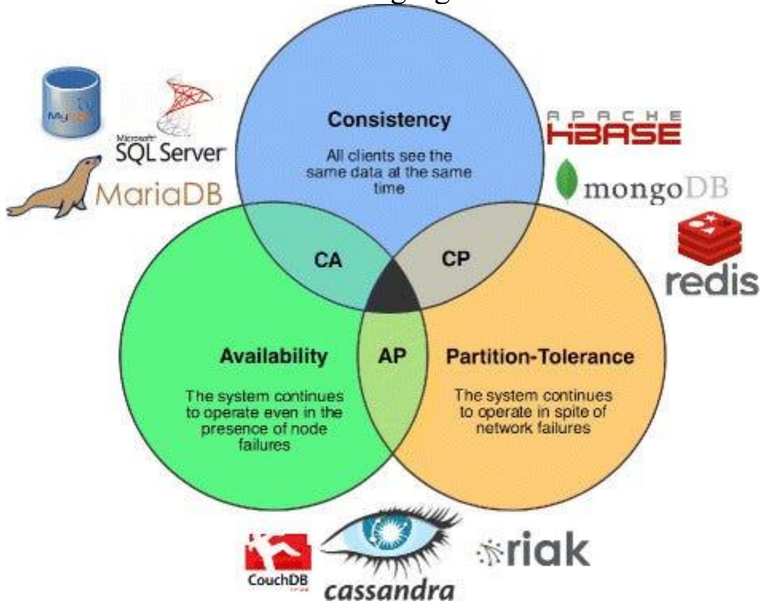
3. Partition Tolerance (P)

Partition tolerance ensures that the system continues to operate even in the presence of network partitions, where communication between some nodes is lost. In a distributed environment, network failures can occur, and partition tolerance guarantees that the system can still function, allowing for either consistent or available responses.

According to the CAP theorem, a distributed database can only achieve two of the three guarantees at any given time:

- CP (Consistency and Partition Tolerance): Systems that prioritize consistency and partition tolerance may sacrifice availability during network partitions. An example is a system that returns errors or unavailable responses if it cannot ensure that all nodes are consistent.
- AP (Availability and Partition Tolerance): Systems that focus on availability and partition tolerance may allow for eventual consistency, meaning that some reads may return stale data while the system continues to operate. Examples include systems like Cassandra and DynamoDB, which prioritize availability.
- CA (Consistency and Availability): It is impossible to achieve consistency and availability in the presence of network partitions. Systems that claim to provide both will inevitably fail during network issues.

CAP theorem can be summarized with following figure:


Consistency
All clients see the same data at the same time
CA
CP
Availability
The system continues to operate even in the presence of node failures
AP
Partition-Tolerance
The system continues to operate in spite of network failures
SQL Server
MariaDB
APACHE HBASE
mongoDB
redis
CouchDB
cassandra
riak

Relaxing durability

- Durability is a key property of database systems that ensures once a transaction has been committed, it remains permanently recorded in the system, even in the event of a failure such as a power outage or system crash. This property is typically achieved through mechanisms like write-ahead logging and data replication, which safeguard the integrity and availability of data.
- If a database can run mostly in memory, apply updates to its in-memory representation, and periodically flush changes to disk, then it may be able to provide substantially higher responsiveness to requests.
- A big website may have many users and keep temporary information about what each user is doing in some kind of session state. There's a lot of activity on this state, creating lots of demand, which affects the responsiveness of the website.
- Another example of relaxing durability is capturing telemetric data from physical devices. It may be that you'd rather capture data at a faster rate, at the cost of missing the last updates should the server go down.
- Replication durability refers to the assurance that data changes made in a distributed system will persist even in the face of failures, thanks to the mechanisms in place to replicate data across multiple nodes or servers.

Quorums

A quorum is a minimum number of votes or acknowledgments required from nodes in a distributed system to consider a read or write operation valid and successful. This mechanism helps ensure consistency and availability in the face of network partitions or node failures.
There are typically two types of quorums used in distributed systems:

- Write Quorum (W): The minimum number of replicas that must acknowledge a write operation before it is considered successful. This ensures that the data is sufficiently replicated across the system to maintain consistency.
- Read Quorum (R): The minimum number of replicas that must be accessed for a read operation to return a valid result. This ensures that the read operation reflects the most recent write, maintaining consistency from the user's perspective.

This relationship between the number of nodes you need to contact for a read (R), those confirming a write (W), and the replication factor (N) can be captured in an inequality:
$W>N/2$
$R+W>N$

Business and System Transactions

- Business transactions refer to meaningful activities or interactions that occur in the context of a business operation. These transactions typically involve the exchange of goods, services, or information and have a direct impact on the organization's financials or operational processes. Examples include sales transactions, purchase orders, inventory management, and customer interactions.
- System transactions, on the other hand, refer to the operations performed by a database management system (DBMS) to ensure the integrity and consistency of data during business transactions. These transactions are fundamental to the functioning of the database and follow the ACID (Atomicity, Consistency, Isolation, Durability) properties to guarantee reliable processing.
- Managing transactions is achieved through **version stamp**: a field that changes every time the underlying data in the record changes. When you read the data you keep a note of the version stamp, so that when you write data you can check to see if the version has changed.

- There are various ways you can construct your version stamps. You can use a counter, always incrementing it when you update the resource. Counters are useful since they make it easy to tell if one version is more recent than another. On the other hand, they require the server to generate the counter value, and also need a single master to ensure the counters aren’t duplicated.
- Another approach is to create a GUID, a large random number that’s guaranteed to be unique. These use some combination of dates, hardware information, and whatever other sources of randomness they can pick up. The nice thing about GUIDs is that they can be generated by anyone and you’ll never get a duplicate; a disadvantage is that they are large and can’t be compared directly for recentness.
- A third approach is to make a hash of the contents of the resource. With a big enough hash key size, a content hash can be globally unique like a GUID and can also be generated by anyone; the advantage is that they are deterministic—any node will generate the same content hash for same resource data. However, like GUIDs they can’t be directly compared for recentness, and they can be lengthy.
- A fourth approach is to use the timestamp of the last update. Like counters, they are reasonably short and can be directly compared for recentness, yet have the advantage of not needing a single master. Multiple machines can generate timestamps—but to work properly, their clocks have to be kept in sync. One node with a bad clock can cause all sorts of data corruptions.

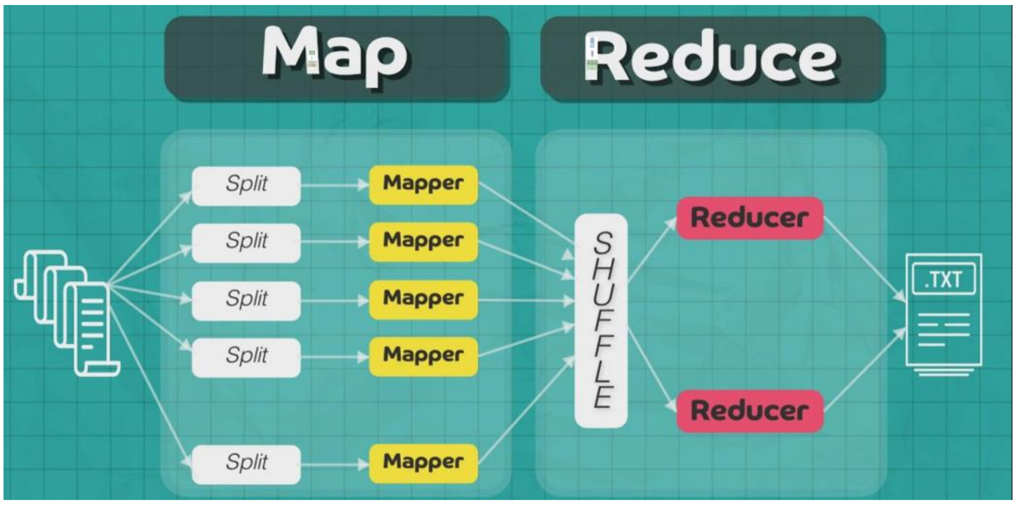
Version stamps on Multiple Nodes

- Version stamps on multiple nodes are a crucial mechanism in distributed systems, enabling the tracking and management of data versions across different replicas or nodes. This approach helps maintain consistency, resolve conflicts, and ensure data integrity in environments where concurrent updates may occur.
- In a distributed system, each node maintains its own version stamp for data items, which can be either logical timestamps, vector clocks, or physical timestamps. When a node performs an update to a data item, it associates the update with its current version stamp. This versioning allows the system to track changes and determine the most recent version of the data across all nodes.
- Logical Timestamps: Each update is assigned a unique logical timestamp based on the order of operations. If two nodes perform updates concurrently, their timestamps help in determining the sequence of updates.
- Vector Clocks: Each node maintains a vector clock that counts the number of updates it has made and tracks updates from other nodes. When a node updates a data item, it increments its own entry in the vector clock and may also compare its vector with others to resolve conflicts.
- Physical Timestamps: In systems using physical timestamps, each update is time-stamped based on real-time. However, this approach can face challenges related to clock synchronization across nodes.

Module-3 Notes

Map-Reduce

MapReduce is a programming model and processing framework for distributed computing, invented by Google, that allows for processing large data sets with a distributed algorithm on a cluster. The MapReduce framework operates primarily in two phases: the Map phase and the Reduce phase.



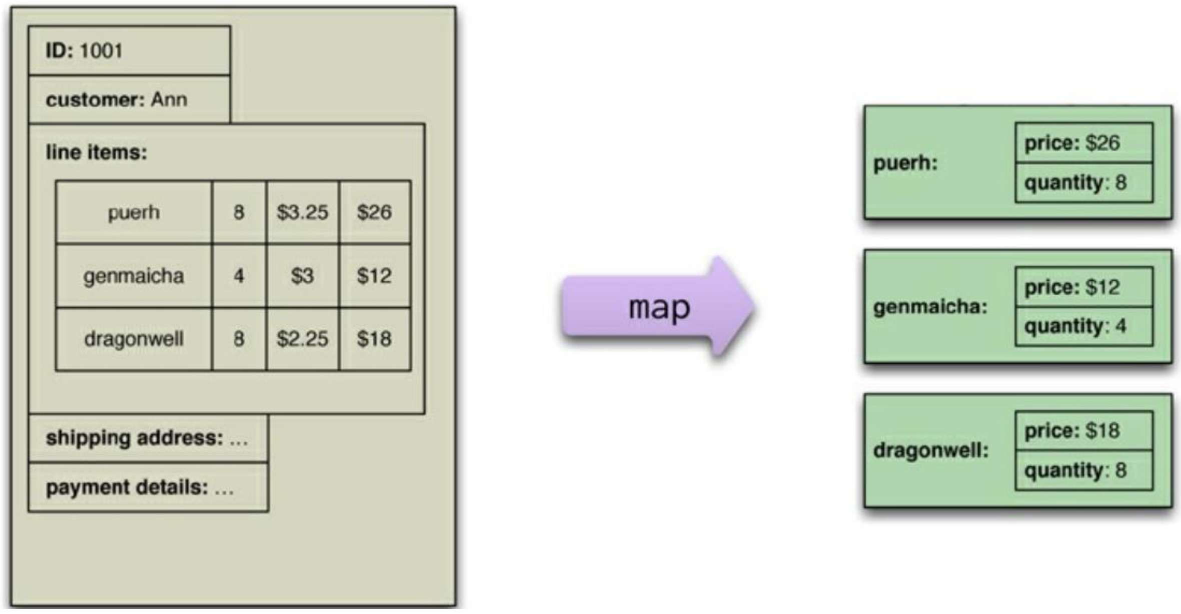
- Map Function: The Map function processes a key/value pair to generate a set of intermediate key/value pairs. It is applied to each input record in parallel.
- Reduce Function: The Reduce function merges all intermediate values associated with the same intermediate key. It processes each group of intermediate key/value pairs to generate the final output.

Workflow:

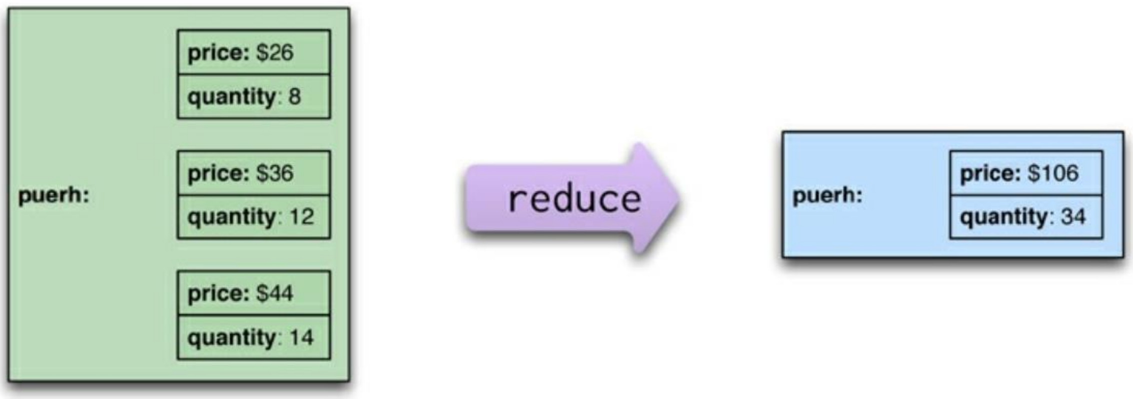
1. **Splitting**: The input data is split into manageable chunks, usually by a distributed file system like Hadoop Distributed File System (HDFS).
2. **Mapping**: The Map function is applied to each chunk, generating intermediate key/value pairs.
3. **Shuffling and Sorting**: The framework sorts and groups the intermediate data by key.
4. **Reducing**: The Reduce function is applied to each group of intermediate data to produce the final output.
5. **Output**: The final results are written to an output store, which can be HDFS or another distributed storage system.

Basic Map-Reduce

The first stage in a map-reduce job is the map. A map is a function whose input is a single aggregate and whose output is a bunch of keyvalue pairs. In this case, the input would be an order. The output would be key-value pairs corresponding to the line items. Each one would have the product ID as the key and an embedded map with the quantity and price as the values



A map function reads records from the database and emits key-value pairs

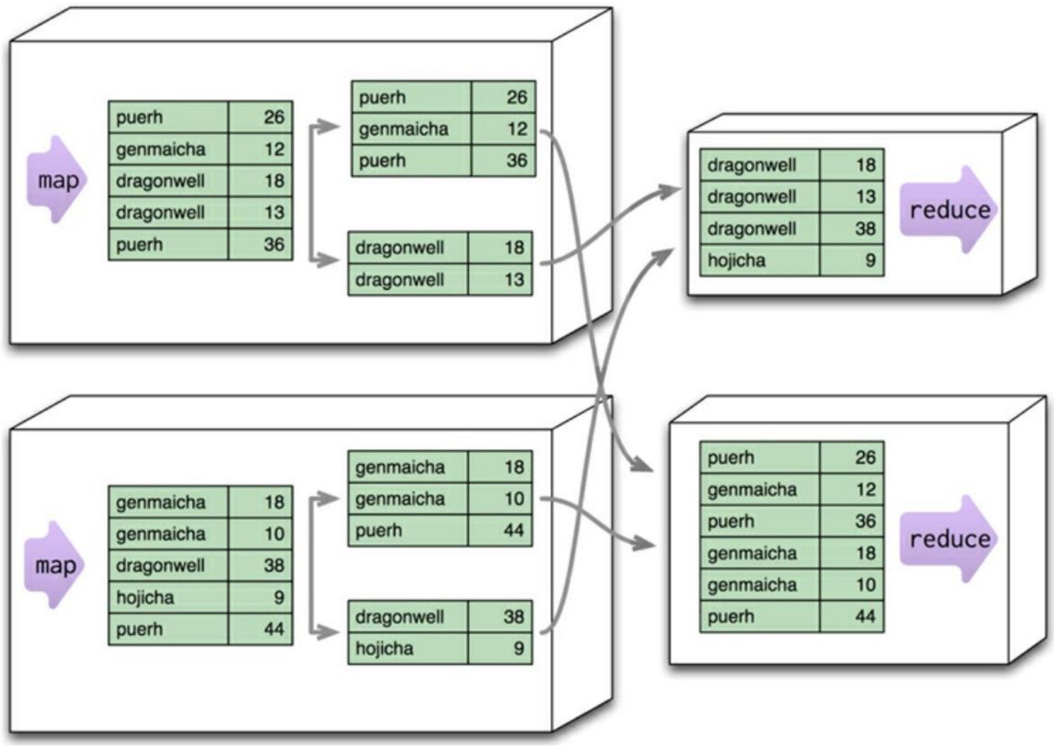


A reduce function takes several key-value pairs with the same key and aggregates them into one.

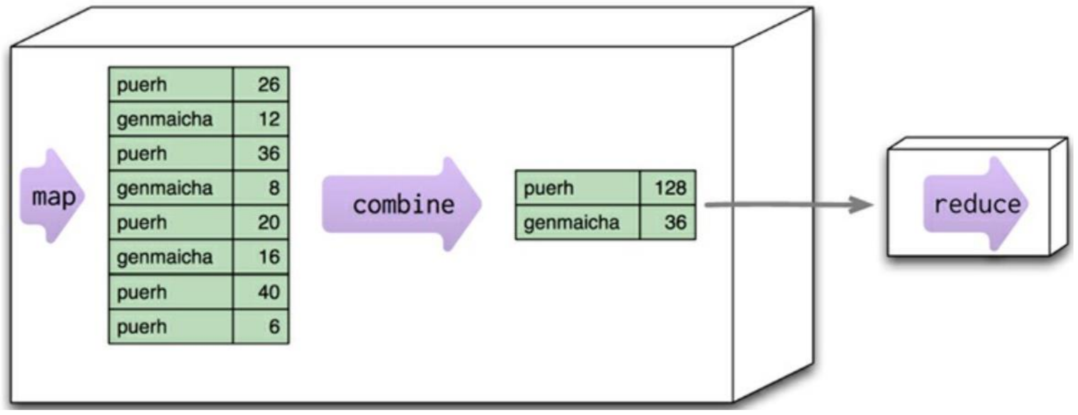
Partitioning and Combining

In the MapReduce framework, partitioning and combining play crucial roles in optimizing the processing of data. Partitioning involves distributing the intermediate key/value pairs generated by the map phase to reducers in an efficient manner. This is typically handled by a partitioner function, which determines the reducer that will handle each intermediate key. A common approach is to use a hash function on the key to ensure an even distribution across reducers, thus balancing the load and avoiding bottlenecks. Combining, on the other hand, is an optimization technique where a combiner function, which is essentially a mini-reducer, is applied to the intermediate data before it is sent to the reducers. This helps to reduce the amount of data shuffled across the network by combining the intermediate key/value pairs locally on each mapper. By performing local aggregation, combiners significantly reduce the volume of data transfer, leading to faster and more efficient processing. Together, partitioning and combining help to enhance the performance and scalability of the MapReduce framework by optimizing data distribution and minimizing network congestion.

Example:



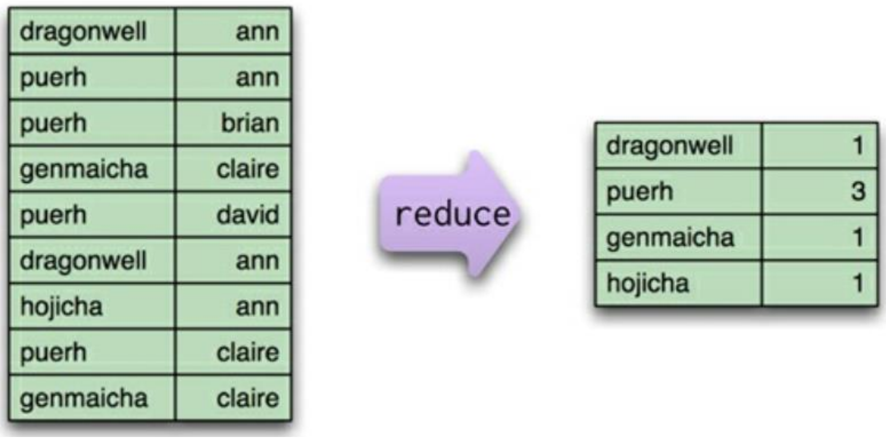
Partitioning allows reduce functions to run in parallel on different keys



Combining reduces data before sending it across the network

A combiner function is, in essence, a reducer function—indeed, in many cases the same function can be used for combining as the final reduction. The reduce function needs a special shape for this to work: Its output must match its input. We call such a function a combinable reducer.

Not all reduce functions are combinable. Eg:

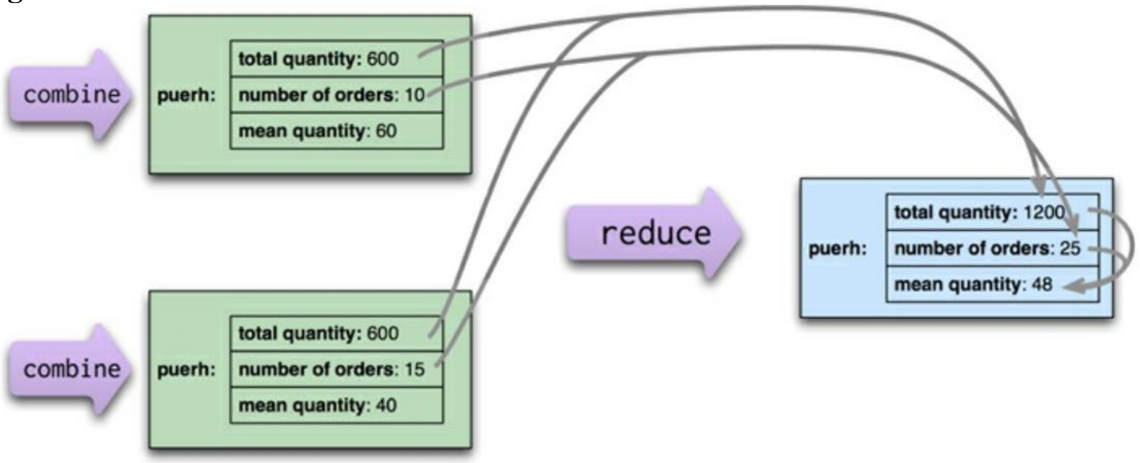


This reduce function, which counts how many unique customers order a particular tea, is not combinable

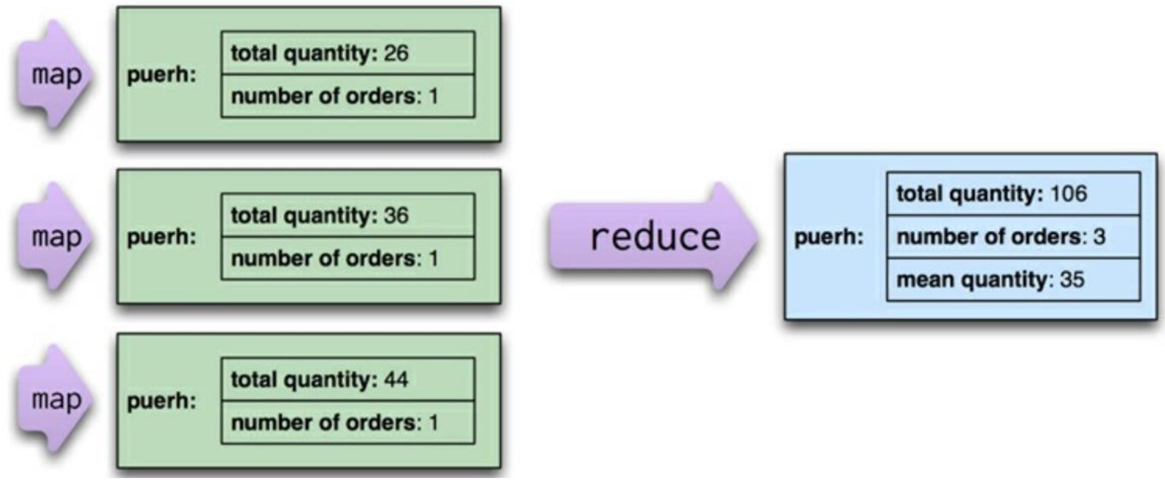
Composing Map-Reduce Calculations

Composing MapReduce calculations involves chaining multiple MapReduce jobs together to perform complex data processing tasks that cannot be accomplished with a single MapReduce job. Each job in the chain takes the output of the previous job as its input, allowing for a series of transformations and aggregations on the data. This composition is essential for tasks like multi-step data processing pipelines, where intermediate results from one phase need further processing in subsequent phases. For example, in a web log analysis scenario, the first MapReduce job might filter and clean the log data, the second job could aggregate the data by user sessions, and the third job might compute statistics on the session data. Effective composition requires careful planning of the intermediate data format and efficient handling of data transfer between jobs. By breaking down complex calculations into a sequence of simpler MapReduce jobs, developers can leverage the power of distributed computing to handle large-scale data processing in a modular and manageable way. This approach also improves fault tolerance, as each stage can be independently retried and debugged.

Eg:



When calculating averages, the sum and count can be combined in the reduce calculation, but the average must be calculated from the combined sum and count.

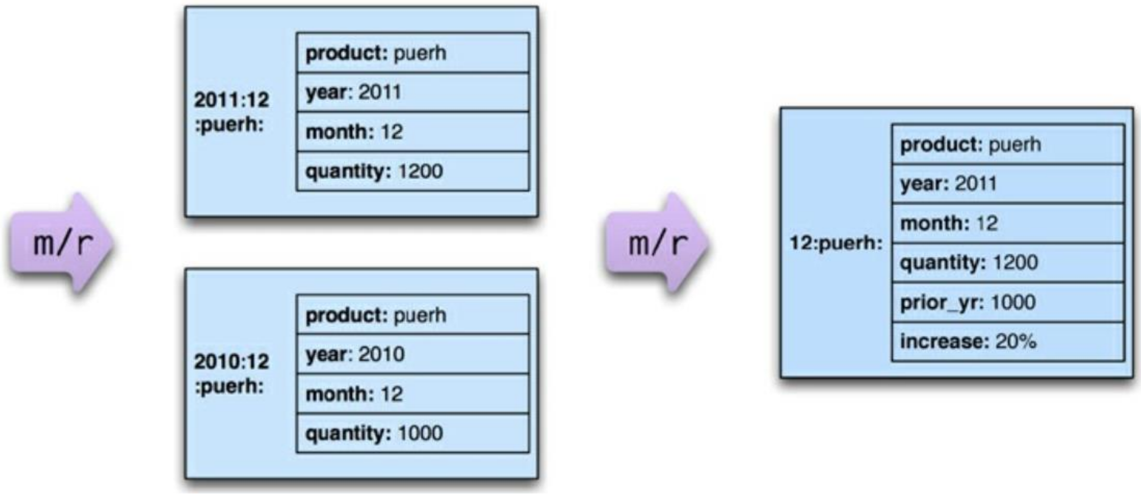


When making a count, each map emits 1, which can be summed to get a total

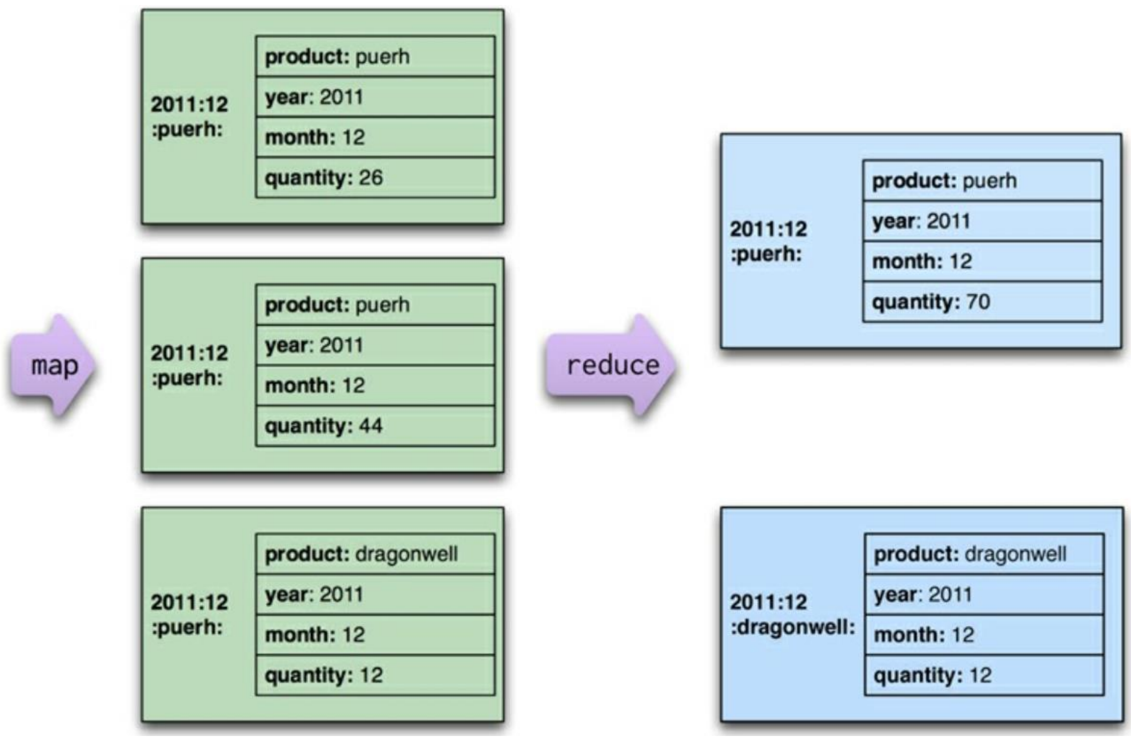
A Two Stage Map-Reduce Example

A two-stage MapReduce process involves executing two consecutive MapReduce jobs to handle complex data processing tasks. In the first stage, the map function processes the raw input data to generate intermediate key/value pairs, and the reduce function aggregates these pairs. For instance, in a word count scenario, the first stage's map function emits each word with a count of one, while the reduce function sums the counts for each word. The output of this stage, consisting of the total counts for each word, serves as the input for the second stage. In the second stage, another map function can further process these aggregated results, such as grouping words by their initial letter or filtering out infrequent words. The reduce function in this stage then performs the final aggregation or transformation, producing the final output. This two-stage approach allows for breaking down complex calculations into manageable steps, enhancing scalability and fault tolerance in distributed computing environments.

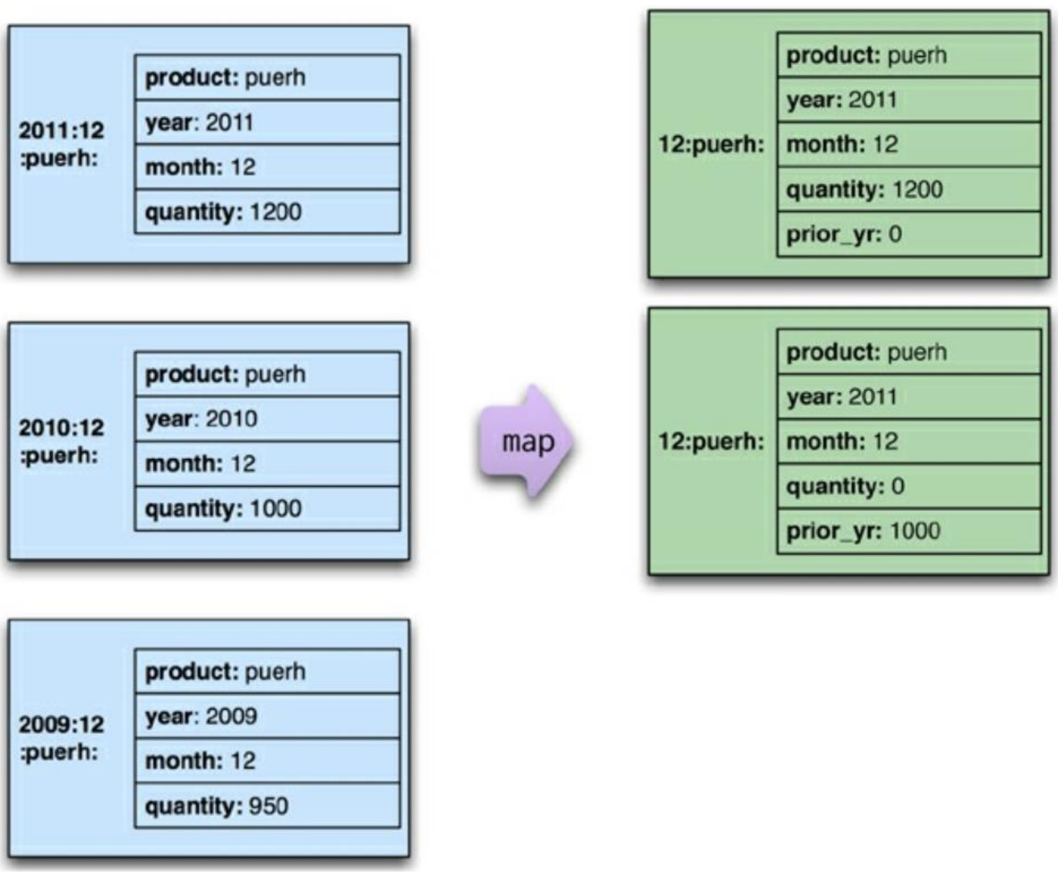
Eg:



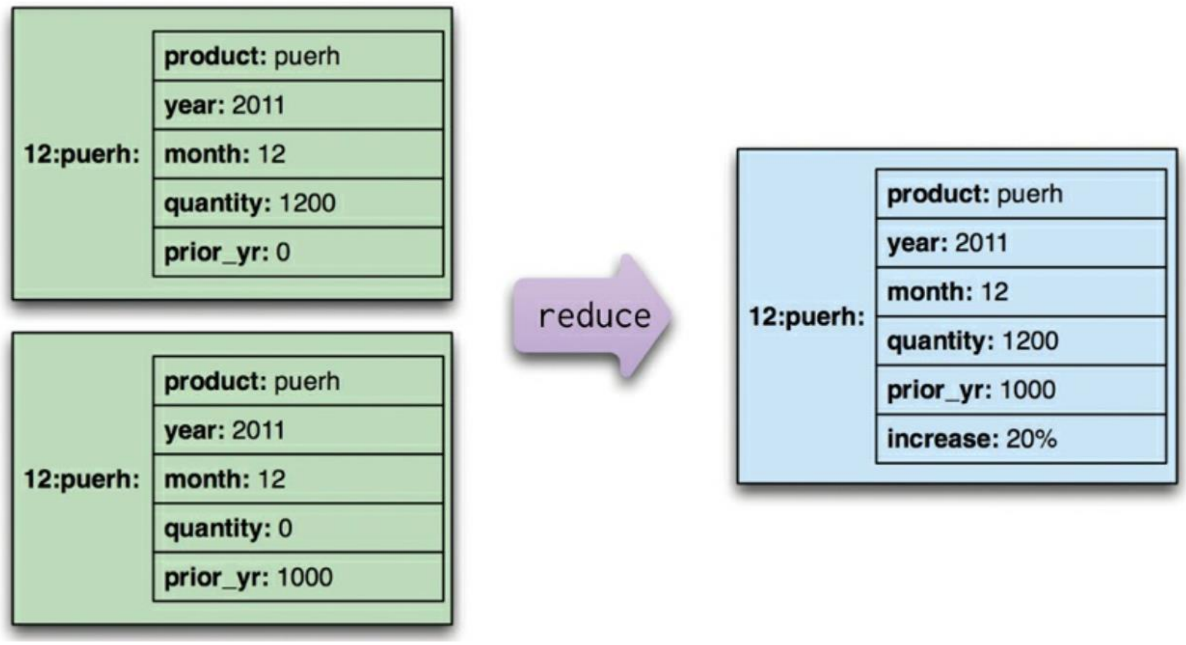
A calculation broken down into two map-reduce steps, which will be expanded in the next three figures



Creating records for monthly sales of a product



The second stage mapper creates base records for year-on-year comparisons.



The reduction step is a merge of incomplete records

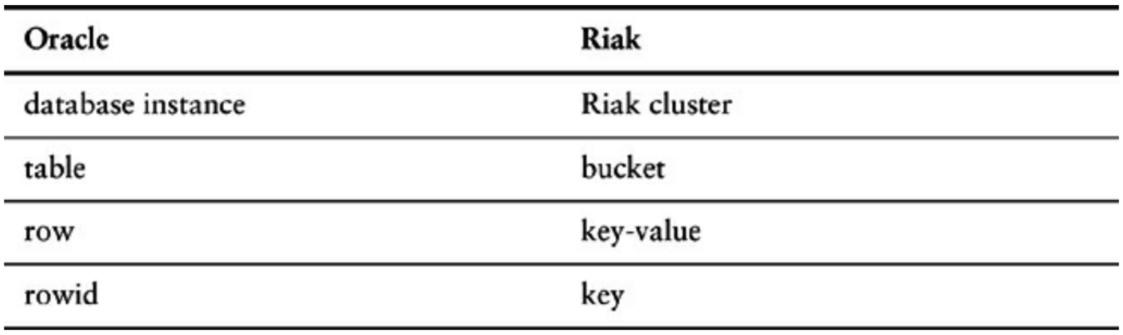
Incremental Map-Reduce

- Incremental MapReduce is an extension of the traditional MapReduce framework designed to handle dynamic data by processing only the changes in the dataset rather than reprocessing the entire dataset.
- This approach is particularly useful for applications where data is continuously updated, such as in real-time analytics or iterative machine learning algorithms.
- In incremental MapReduce, the framework identifies and processes only the newly added or modified data since the last computation.
- The map function processes these changes to generate updated intermediate key/value pairs, while the reduce function merges these updates with the previously computed results.
- By focusing on the incremental changes, this method significantly reduces computational overhead and improves efficiency, allowing for faster and more responsive data processing.
- This approach also enhances resource utilization and minimizes latency, making it ideal for applications requiring frequent updates and real-time insights.
- The map stages of a map-reduce are easy to handle incrementally—only if the input data changes does the mapper need to be rerun. Since maps are isolated from each other, incremental updates are straightforward.

Key-Value Databases

Key-value databases are a type of NoSQL database that store data as a collection of key-value pairs, where each key is unique and directly associated with a specific value. This simple and flexible data model allows for high performance and scalability, making key-value databases ideal for applications that require fast read and write operations, such as caching, session management, and real-time analytics. The value in a key-value database can be any type of data, ranging from simple strings to complex objects like JSON or binary data. Operations are performed using keys, which enables quick retrieval of values without the need for complex query language or table scans. Examples of key-value databases

include Redis, DynamoDB, and Riak. These databases are particularly well-suited for distributed systems, as they can easily scale horizontally by partitioning data across multiple nodes, ensuring high availability and fault tolerance.
Eg:



Oracle	Riak
database instance	Riak cluster
table	bucket
row	key-value
rowid	key

Key-Value Store terminologies

Key-value stores have several important terminologies that define their structure, operations, and performance characteristics. Here are some key terms:

- **Key**: A unique identifier used to access a specific value in the key-value store. Keys are typically strings but can also be other data types.
- **Value**: The data associated with a key in the key-value store. Values can be simple data types (like strings or numbers) or more complex structures (like JSON objects or binary blobs).
- **Bucket/Bucket**: A logical grouping or namespace for key-value pairs within the database. It helps organize and isolate data within a larger key-value store. Some systems use terms like "bucket" or "table."
- **Get**: An operation to retrieve the value associated with a given key. This is one of the fundamental operations in a key-value store.
- **Put**: An operation to insert or update the value associated with a given key. This operation writes data to the key-value store.
- **Delete**: An operation to remove the key-value pair from the store based on the given key.
- **TTL (Time to Live)**: A setting that defines the lifespan of a key-value pair. After the TTL expires, the key-value pair is automatically deleted from the store.
- **Node**: An individual server or instance in a distributed key-value store. Multiple nodes work together to store and manage data.
- **Cluster**: A collection of nodes that work together as a single key-value store. Clustering improves scalability and fault tolerance.
- **Distributed Hash Table (DHT)**: A decentralized data structure used to distribute and locate data across nodes in a key-value store.

Key-Value Store Features

Key-value stores are a type of NoSQL database that offer several distinct features, making them suitable for various applications that require fast, flexible, and scalable data storage. Here are some key features of key-value stores:

- **Simplicity**: Key-value stores have a straightforward data model where data is stored as a collection of key-value pairs. This simplicity allows for easy implementation and quick access to data.

- **High Performance**: Due to their simple structure, key-value stores can achieve high read and write performance. Operations like get, put, and delete are optimized for speed, making these databases ideal for applications that require rapid data access.
- **Scalability**: Key-value stores are designed to scale horizontally by adding more nodes to the system. Data can be partitioned and distributed across multiple nodes, allowing the database to handle large volumes of data and high traffic loads efficiently.
- **Flexibility**: Values in key-value stores can be of various types, including strings, numbers, JSON objects, or binary data. This flexibility allows developers to store a wide range of data types without the constraints of a rigid schema.
- **Distributed Architecture**: Many key-value stores are built to operate in distributed environments. They provide mechanisms for data replication and partitioning, ensuring high availability and fault tolerance.
- **In-Memory Capabilities**: Some key-value stores, like Redis, offer in-memory data storage, which significantly boosts read and write speeds by keeping data in RAM. This is particularly useful for caching and real-time applications.
- **Eventual Consistency**: Key-value stores often adopt an eventual consistency model, where updates to the data are propagated to all nodes over time. This model provides a good balance between consistency and availability in distributed systems.
- **Replication and High Availability**: Key-value stores often support data replication across multiple nodes to ensure data durability and high availability. This replication can be synchronous or asynchronous, depending on the desired consistency level.
- **Support for Large Data Volumes**: Key-value stores can handle large amounts of data by distributing it across multiple nodes. This capability makes them suitable for big data applications.
- **Schema-Free**: Key-value stores do not require a fixed schema, allowing for dynamic and flexible data models. Developers can easily add or modify data without schema migrations.

CRUD operations in Riak

Riak is a distributed NoSQL database designed for high availability, fault tolerance, and scalability. CRUD operations (Create, Read, Update, Delete) in Riak are straightforward, leveraging its key-value store nature. Here's an overview of how to perform these operations:
Connecting to Riak:

```
myClient = riak.RiakClient(
pb_port=10017, protocol='pbc')
```

Creating buckets and objects:

```
myBucket = myClient.bucket('test')
val1 = 1
key1 = myBucket.new('one', data=val1)
key1.store()
```

```
val2 = "two"
key2 = myBucket.new('two', data=val2)
key2.store()
```

```
val3 = {"myValue": 3}
key3 = myBucket.new('three', data=val3)
key3.store()
```

Reading objects from Riak:

```
fetched1 = myBucket.get('one')
fetched2 = myBucket.get('two')
fetched3 = myBucket.get('three')
```

Updating objects in Riak:

```
fetched3.data["myValue"] = 42
fetched3.store()
```

Deleting objects:

```
fetched1.delete()
fetched2.delete()
fetched3.delete()
```

Working with complex objects:

```
book = {
  'isbn': "1111979723",
  'title': "Moby Dick",
  'author': "Herman Melville",
  'body': "Call me Ishmael. Some years ago...",
  'copies_owned': 3
}
```

```
booksBucket = myClient.bucket('books')
newBook = booksBucket.new(book['isbn'],
data=book)
newBook.store()
fetchedBook = booksBucket.get(book['isbn'])
print(fetchedBook.encoded_data)
```

Scaling in Riak

- Riak is designed to scale horizontally by adding more nodes to a cluster, allowing it to handle large volumes of data and increasing traffic efficiently. Data is automatically distributed across the cluster using consistent hashing and virtual nodes (vnodes), which ensures even distribution and balancing of data. Each node in the cluster manages multiple vnodes, and when new nodes are added, Riak automatically redistributes data to maintain balance. This horizontal scalability allows Riak to grow seamlessly without affecting the performance or availability of the system. The cluster's data is replicated across multiple nodes for redundancy, and the replication factor can be adjusted based on the desired level of fault tolerance. Riak's eventual consistency model ensures that updates to data are eventually propagated to all replicas, while vector clocks help resolve conflicts that might occur due to network partitions or concurrent writes.

- Riak also provides fault tolerance and automatic recovery mechanisms. If a node fails, the system continues to serve data from other replicas, ensuring high availability. Riak can re-replicate data to restore the desired replication factor after a failure. Additionally, the system is optimized for both read and write scalability. Write-heavy workloads benefit from Riak's ability to handle concurrent writes across multiple nodes, while read-heavy applications can take advantage of Riak’s ability to read from any replica, reducing latency. Through its quorum-based reads and writes, Riak offers flexibility in balancing consistency and availability, depending on the application's needs. This makes Riak a robust and scalable solution for large-scale distributed applications.

Suitable use cases:

- **Storing Session Information -** Storing session information in a key-value store is a common and efficient method for managing user sessions in web applications. In this approach, each user session is represented as a unique key (often a session ID), with the associated session data (such as user preferences, authentication tokens, or temporary data) stored as the value. The key-value store provides fast access to session data, as retrieving a session is a simple lookup by key. This setup is particularly well-suited for distributed systems because key-value stores like Redis or Riak can scale horizontally, ensuring that session data is available across multiple servers. By storing session information in-memory (for example, in Redis), applications can achieve low-latency access, making it ideal for real-time applications. Additionally, many key-value stores support features like TTL (Time to Live), which automatically expires session data after a predefined period, enhancing security by cleaning up stale sessions. The simplicity and performance of key-value stores make them an excellent choice for managing session information in high-traffic applications.

- **User Profiles, Preferences-** Storing user profiles and preferences in a key-value store is an effective way to manage and quickly retrieve personalized data for users in web or mobile applications. In this model, each user profile can be stored with a unique key, typically a user ID, and the corresponding value contains the user’s preferences, settings, and other personalized data in the form of a structured format like JSON or serialized objects. This approach allows for fast access to user-specific information, as the key-value store is optimized for quick lookups by key. Key-value stores also scale easily to handle millions of users and can store complex, nested data such as theme preferences, language settings, or recently viewed items. Additionally, the flexibility of key-value stores means that as user preferences evolve, new attributes can be added without a rigid schema, providing adaptability in dynamic applications. The ability to quickly retrieve and update these profiles makes key-value stores an excellent choice for personalized experiences, particularly when performance and scalability are essential.

- **Shopping Cart Data-**Using a key-value store to manage shopping cart data is a popular approach due to its simplicity and high performance. In this model, each shopping cart is associated with a unique key, typically a user session ID or user ID, and the value contains the items added to the cart, along with relevant data such as quantities, product IDs, prices, and any discounts or promotions. The key-value store allows for fast retrieval and updates to the shopping cart, making it ideal for real-time applications where users may add, remove, or modify items frequently. This design also scales efficiently, as it can handle thousands or even millions of carts across distributed systems. Many key-value stores support features like TTL (Time to Live), which can automatically expire carts that remain inactive for a certain period, helping to manage resources and prevent stale data. The flexibility of key-value stores also enables seamless handling of cart data with varying structures, providing adaptability as the shopping cart evolves with different user interactions and business requirements.

When not to use Key-Value stores

- **Relationships among Data-**Key-value stores are not ideal for scenarios that require complex relationships among data, such as those involving many-to-many, one-to-many, or hierarchical relationships. These types of relationships often require JOIN operations or the ability to link data across multiple tables or entities, which is inherently challenging in a key-value store. Since key-value databases are designed for simplicity, with each key mapped to a single value, they do not support querying across multiple keys or efficiently managing interconnected data. For example, if you need to represent relationships like a user and their orders, or products and their categories, key-value stores lack the inherent mechanisms to easily model and query these relationships. This makes them less suitable for use cases that require complex queries, relational integrity, or the need to enforce foreign key constraints. In such cases, relational databases or graph databases are better suited as they provide powerful querying capabilities and built-in support for relationships between data.

- **Multioperation Transactions-**Key-value stores are not well-suited for multi-operation transactions that require strong consistency across multiple keys or data entities. In scenarios where multiple operations need to be executed atomically (e.g., transferring funds between accounts, updating multiple related records, or enforcing complex business rules across various entities), key-value stores typically fall short. These databases are designed to perform single key-value operations efficiently but do not natively support ACID (Atomicity, Consistency, Isolation, Durability) properties across multiple operations. While some key-value stores offer basic support for transactions, they generally lack the sophistication and reliability required for complex, multi-step transactions involving several keys or data models. This makes them less suitable for applications that rely on strict transaction management, such as banking systems or applications with complex data integrity requirements. In such cases, relational databases or more specialized systems like distributed databases with transaction support (e.g., Google Spanner) are better alternatives.

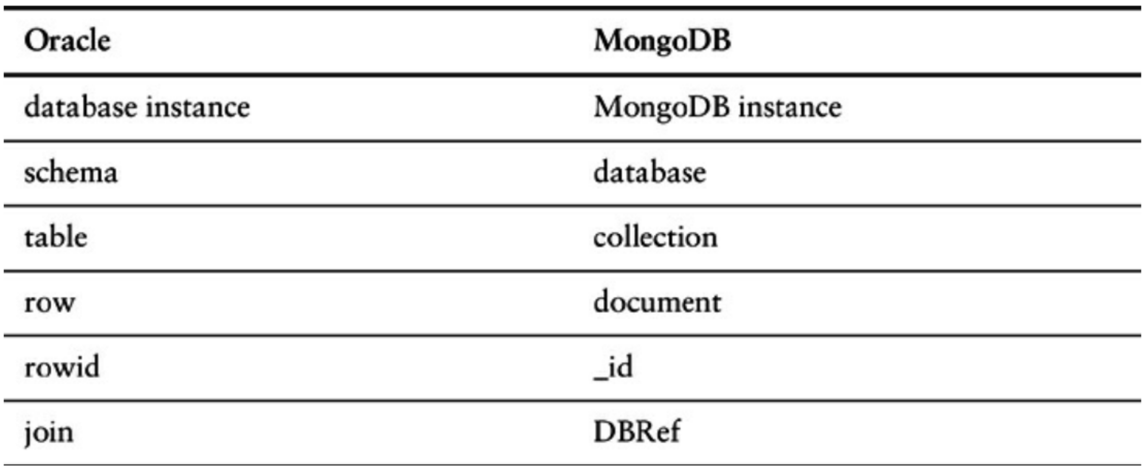
- **Query by Data-**Key-value stores are not suitable for querying data based on non-key attributes, as they are designed to retrieve values using a specific key. They do not natively support complex queries or secondary indexes that allow for filtering, sorting, or searching by data fields other than the key. In scenarios where you need to perform queries like "find all users with a specific age" or "retrieve products within a certain price range," key-value stores are limited because they lack built-in query mechanisms or advanced indexing features. These types of queries typically require scanning through all data or using additional processing, which can be inefficient and timeconsuming, especially as the data set grows. For use cases requiring flexible querying capabilities, such as filtering, range queries, or joining data across different fields, relational databases or document-based NoSQL systems with richer querying support (e.g., MongoDB or SQL databases) are more appropriate.

- **Operations by Sets -** Key-value stores are not well-suited for operations that involve sets or collections of data, such as union, intersection, or difference, especially when the data is spread across multiple keys. Since key-value databases store data as simple key-value pairs, they lack native support for set operations or the ability to treat values as collections that can be manipulated in bulk. For example, performing set-based operations like finding common elements between two sets of data, or combining multiple sets, would require manually retrieving and processing each key-value pair individually, which can be inefficient and cumbersome. Furthermore, key-value stores typically do not support advanced data types like lists or sets in the way that document-oriented databases or graph databases do. For applications requiring frequent and efficient set operations, such as collaborative filtering, tagging systems, or social network analysis, databases that provide native support for sets or collections, such as Redis (with its set data type) or graph databases, are more suitable.

Module-4 Notes

Document Databases

Document databases are a type of NoSQL database that store, retrieve, and manage data in the form of documents, typically using formats like JSON, BSON, or XML. Unlike relational databases that store data in tables with predefined schemas, document databases are schema-less or have flexible schemas, allowing for the storage of varied data structures in the same collection. Each document in a document database is a self-contained unit of data, and documents can contain nested structures, such as arrays or sub-documents, making them highly flexible and suitable for hierarchical or complex data. Popular document databases like MongoDB and CouchDB enable fast querying, indexing, and scalability, and they support features like full-text search, aggregation, and secondary indexing. Because of their flexible schema and ability to scale horizontally across distributed systems, document databases are ideal for applications with rapidly changing data structures, such as content management systems, e-commerce platforms, and real-time analytics. Eg:



Oracle	MongoDB
database instance	MongoDB instance
schema	database
table	collection
row	document
rowid	_id
join	DBRef

Features of Document oriented databases

Document-oriented databases offer several key features that make them suitable for flexible and scalable data management in modern applications. Here are some of the important features:

- Schema Flexibility: Document databases do not require a predefined schema, allowing for dynamic and flexible data storage. Each document can have its own structure, with varying fields and data types, making it easier to handle changing data models.
- JSON/BSON/XML Storage: Documents are typically stored in formats like JSON, BSON, or XML, which are human-readable and allow for complex, nested data structures. This makes it easy to represent hierarchical data and reduces the need for normalization, as opposed to relational databases.
- Indexing and Querying: Document databases support powerful indexing mechanisms, enabling efficient searches on various fields, including those within nested structures. Complex queries, including range queries and full-text search, can be executed quickly.
- Scalability: Document databases are designed to scale horizontally by distributing data across multiple servers or nodes. This makes them suitable for handling large volumes of data and high traffic, especially in distributed systems.
- Data Aggregation: Many document databases, such as MongoDB, provide built-in aggregation frameworks that allow for complex data analysis and transformation. This is useful for analytics and reporting without needing to export data to an external processing system.

- High Availability and Fault Tolerance: Document databases typically support replication and automatic failover, ensuring high availability and data durability. Data is replicated across multiple nodes to provide fault tolerance, so the system continues to function even in the event of hardware failures.
- Atomic Operations: Some document databases support atomic operations, allowing modifications to be made to individual documents in a way that ensures consistency. For example, MongoDB supports atomic updates on individual documents, providing reliable data integrity.
- Ease of Integration: Due to their flexible schema and use of JSON-like data formats, document databases are well-suited for integration with web applications, APIs, and microservices, making them a popular choice for modern software architectures.

MongoDB

MongoDB is a popular, open-source, document-oriented NoSQL database designed for scalability, flexibility, and high performance. It stores data in a JSON-like format called BSON (Binary JSON), which allows for complex, nested structures that can evolve over time without requiring a fixed schema. MongoDB supports horizontal scaling through sharding, where data is distributed across multiple servers, ensuring high availability and the ability to handle large datasets and high-traffic applications. Its powerful query language allows for efficient data retrieval, including support for full-text search, aggregation, and geospatial queries. MongoDB also provides features like automatic failover, replica sets for data redundancy, and ACID-compliant transactions for reliable data consistency. Because of its flexibility and scalability, MongoDB is widely used in applications ranging from real-time analytics and content management systems to e-commerce platforms and mobile applications.

CAP Theorem and MongoDB

The CAP Theorem (Consistency, Availability, Partition Tolerance) states that a distributed system can only guarantee two of the three properties at the same time: Consistency (every read operation returns the most recent write), Availability (every request will receive a response, even if it’s not the most up-to-date data), and Partition Tolerance (the system continues to operate despite network or hardware failures that split the system into partitions). MongoDB, being a distributed database, primarily adheres to the principles of the CA (Consistency and Availability) model under normal conditions, ensuring that all nodes in a replica set are consistent with each other and that the system remains available for read/write operations. However, MongoDB can be configured to prioritize Partition Tolerance over consistency in situations where network partitions occur. By default, MongoDB uses eventual consistency, which can lead to temporary inconsistencies across replica sets until data is synchronized, making it better suited for situations where availability and partition tolerance are more critical than strict consistency, especially in distributed systems with high traffic.

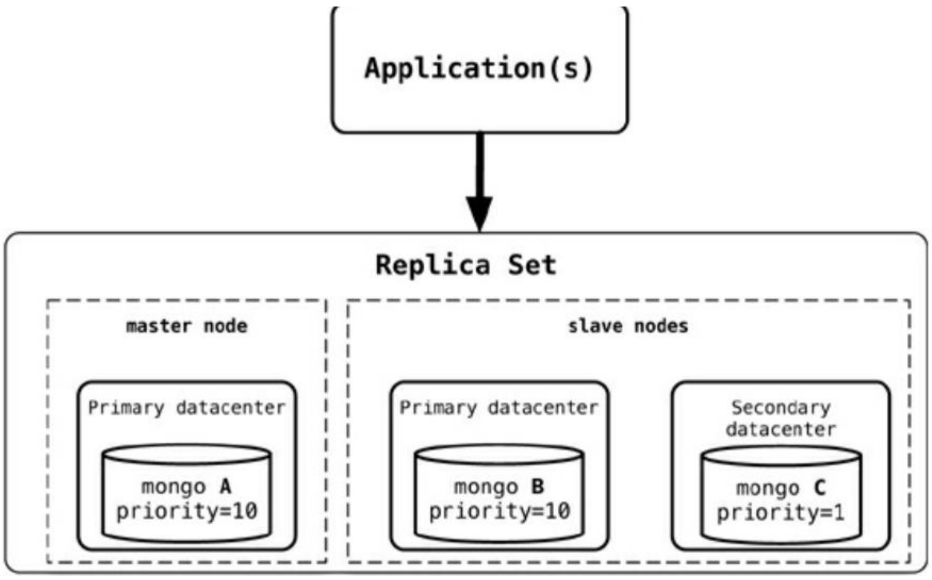
Availability in MongoDB

In MongoDB, replica sets are a group of MongoDB servers that maintain the same data set, providing redundancy and high availability. A replica set consists of a primary node and one or more secondary nodes. The primary node handles all write operations, while the secondary nodes replicate the data from the primary. If the primary node fails, one of the secondary nodes can be automatically promoted to become the new primary, ensuring that the database remains available and minimizing downtime.

Key Features of Replica Sets:

- Automatic Failover: If the primary node becomes unavailable, the replica set will automatically elect a new primary from the secondary nodes to ensure continuous availability.
- Data Replication: Data is replicated from the primary to the secondaries asynchronously, meaning the secondaries may lag slightly behind the primary. This provides fault tolerance and ensures that there are multiple copies of the data across different nodes.
- Read and Write Distribution: By default, writes are directed to the primary node, while reads can be directed to either the primary or secondary nodes. However, this can be configured to prioritize reading from the primary or only from secondaries.
- Consistency and Durability: MongoDB supports strong consistency for write operations on the primary, while reads can be configured for consistency or availability depending on the application's needs. Replica sets help ensure data durability and protection against data loss in case of server failures.
- Configurable Write Concern and Read Concern: MongoDB allows fine-grained control over how many nodes must acknowledge a write or read operation, offering a balance between consistency, availability, and performance.

Eg:



Replica set configuration with higher priority assigned to nodes in the same datacenter

The application writes or reads from the primary (master) node. When connection is established, the application only needs to connect to one node (primary or not, does not matter) in the replica set, and the rest of the nodes are discovered automatically. When the primary node goes down, the driver talks to the new primary elected by the replica set

MongoDB CRUD operations

//To list all databases

```
show dbs;
```

//To create or use databases

```
use blog;
```

//Creating a collection (table) inside databases

```
db.createCollection("posts");

//Inserting documents (Create)
db.posts.insertOne({
 title: "Post Title 1",
 body: "Body of post.",
 category: "News",
 likes: 1,
 tags: ["news", "events"],
 date: Date()
})

db.posts.insertMany([
 {
  title: "Post Title 2",
  body: "Body of post.",
  category: "Event",
  likes: 2,
  tags: ["news", "events"],
  date: Date()
 },
 {
  title: "Post Title 3",
  body: "Body of post.",
  category: "Technology",
  likes: 3,
  tags: ["news", "events"],
  date: Date()
 },
 {
  title: "Post Title 4",
  body: "Body of post.",
  category: "Event",
  likes: 4,
  tags: ["news", "events"],
  date: Date()
 }
])
```

//Retrieval operations

```
db.posts.find()
db.posts.findOne()
db.posts.find( {category: "News"} )
db.posts.find({}, {title: 1, date: 1})
db.posts.find({}, {_id: 0, title: 1, date: 1})
db.posts.find({}, {category: 0})
```

//Update operations

```
db.posts.updateOne( { title: "Post Title 1" }, { $set: { likes: 2 } } )
db.posts.updateOne(
 { title: "Post Title 5" },
 {
  $set:
   {
    title: "Post Title 5",
    body: "Body of post.",
    category: "Event",
    likes: 5,
    tags: ["news", "events"],
    date: Date()
   }
 },
 { upsert: true }
)

db.posts.updateMany({}, { $inc: { likes: 1 } })
```

//Delete operations

```
db.posts.deleteOne({ title: "Post Title 5" })
db.posts.deleteMany({ category: "Technology" })
```

Module-4 (Contd..)

Availability

MongoDB offers several options to ensure high availability:

1. Replica Sets:
   - Primary and Secondary Nodes: In a replica set, data is replicated across multiple nodes. One node is the primary that receives all write operations, while secondary nodes replicate the data from the primary and can serve read operations.
   - Automatic Failover: If the primary node goes down, an eligible secondary node is automatically promoted to primary, ensuring that the database remains available.
2. Sharding:

   - Data Distribution: Sharding distributes data across multiple servers or clusters, which can help in scaling out horizontally. Each shard is a replica set, ensuring data redundancy and high availability.
   - Query Routing: MongoDB uses a query router (mongos) to direct operations to the correct shard, which helps maintain performance and availability.
3. Cloud Deployment:
   - MongoDB Atlas: A fully managed cloud database service that offers automated backups, monitoring, and alerts. Atlas ensures high availability through built-in replication and failover mechanisms across multiple regions and availability zones.
4. Backup and Recovery:
   - Regular Backups: Regularly scheduled backups and point-in-time recovery options help in maintaining data integrity and availability in case of failures.
   - Snapshot and Continuous Backup: Cloud providers like MongoDB Atlas offer continuous backups and snapshot capabilities, enabling quick recovery from data loss incidents.
5. Geographical Distribution:
   - Multi-Region Deployment: Deploying replica sets across different geographic regions can protect against regional outages, ensuring that your database remains available globally.
6. Maintenance and Upgrades:
   - Rolling Upgrades: MongoDB supports rolling upgrades, allowing you to update the database software without significant downtime.

Replica-sets

A MongoDB replica set is a group of MongoDB instances that maintain the same data set, providing redundancy and high availability. Here's a detailed overview of how replica sets work:

Components of a Replica Set

1. Primary Node:
   - Handles all write operations.
   - Only one primary per replica set.
   - Clients can read from the primary.
2. Secondary Nodes:
   - Replicate the data from the primary node.
   - Can serve read operations if configured.
   - If the primary fails, an eligible secondary is elected as the new primary.
3. Arbiter:
   - Does not store data.
   - Participates in elections to break ties but does not become a primary.

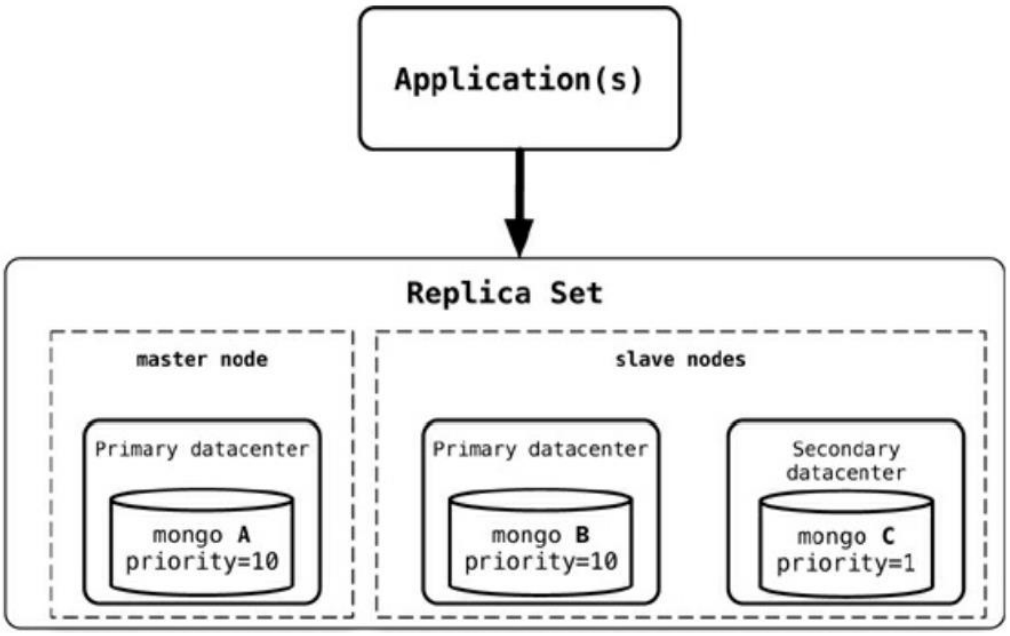
Key Features of Replica Sets

1. Data Replication:
   - The primary node records all changes in the oplog (operations log).
   - Secondary nodes replicate these changes asynchronously from the oplog to maintain the same dataset.
2. Automatic Failover:
   - If the primary node becomes unavailable, the replica set members initiate an election process to elect a new primary from the secondary nodes.

- The new primary is chosen based on factors like node priority and freshness of data.

3. Read Preference:
    - Clients can specify read preferences to read from the primary or secondary nodes based on their requirements.
    - Read preferences include: primary, primaryPreferred, secondary, secondaryPreferred, and nearest.
4. Data Consistency:
    - By default, MongoDB ensures strong consistency for writes by acknowledging write operations only when the data is written to the primary.
    - Read operations can be configured for eventual consistency by reading from secondary nodes.

Working



The application writes or reads from the primary (master) node. When connection is established, the application only needs to connect to one node (primary or not, does not matter) in the replica set, and the rest of the nodes are discovered automatically. When the primary node goes down, the driver talks to the new primary elected by the replica set. The application does not have to manage any of the communication failures or node selection criteria. Using replica sets gives you the ability to have a highly available document data store.

Scaling

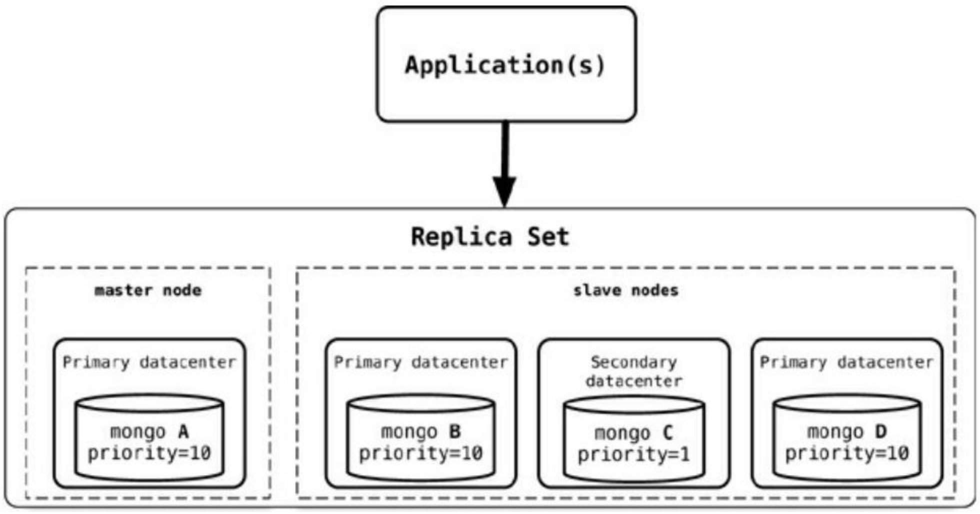
Scaling in **MongoDB** is a critical process that ensures a database can handle increasing data volumes, user traffic and processing demands. As applications grow, maintaining optimal performance and resource utilization becomes essential.

The need for scaling in MongoDB arises due to below factors:

- **Increased Data Volume:** As the amount of data stored in a MongoDB database grows the performance of queries and operations can be affected. Scaling allows to distribution of this data across various servers or nodes and prevents poor performance.

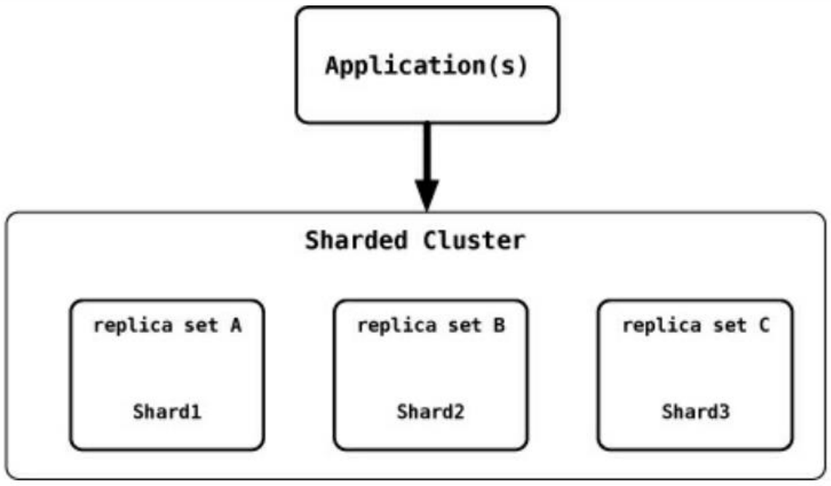
- **Rising User Traffic:** Growing applications often experience a lot of users and increased concurrent requests. Scaling ensures the database can handle a large number of read and write operations and also maintains responsiveness and user experience.
- **Diverse Workloads**: Applications may implement various methods to support new features or functionalities which lead to diverse workloads. Scaling allows organizations to adapt to changing demands by optimizing the database system for different types of queries and operations.
- **Improved Fault Tolerance:** Horizontal scaling in MongoDB is achieved through sharding and enhances **fault tolerance**. By distributing data across multiple servers (**shards**), the system can continue to operate even if one server fails. It ensures **high availability** and **reliability**.
- **Cost-Efficiency: Horizontal scaling,** which involves adding more hardware as needed and offers a **cost-effective** approach to handling increased workloads. Organizations can scale incrementally based on **demand**, optimizing infrastructure costs.

Working
Scaling for heavy read-applications:
Scaling for heavy-read loads can be achieved by adding more read slaves, be directed to the slaves. Given a heavy-read application, with our 3-node replica-set cluster, we can add more read capacity to the cluster as the read load increases just by adding more slave nodes to the replica set to execute reads.



Scaling for write-only applications:

When a new node is added, it will sync up with the existing nodes, join the replica set as secondary node, and start serving read requests. An advantage of this setup is that we do not have to restart any other nodes, and there is no downtime for the application either.



The shard key plays an important role. You may want to place your MongoDB database shards closer to their users, so sharding based on user location may be a good idea. When sharding by customer location, all user data for the East Coast of the USA is in the shards that are served from the East Coast, and all user data for the West Coast is in the shards that are on the West Coast.

Suitable use cases

- **Event logging-** MongoDB can be effectively used for event logging by leveraging its flexible schema, scalability, and real-time capabilities. Events can be logged directly into MongoDB collections, taking advantage of features like capped collections to manage fixed-size log data efficiently. The database's ability to handle high write loads and its powerful querying capabilities make it suitable for storing and retrieving log events quickly. Additionally, MongoDB can be integrated with log management and analysis tools like the ELK stack or Graylog to enhance log aggregation, search, and visualization, providing a robust solution for real-time event monitoring, debugging, and analysis

- **Content Management Systems, Blogging Platforms-** MongoDB is an excellent choice for building content management systems (CMS) and blogging platforms due to its flexible schema, scalability, and ease of handling diverse data types. Its document-oriented structure allows for dynamic content storage, enabling the easy management of various content types like blog posts, comments, and user profiles. MongoDB's scalability ensures that the CMS or blogging platform can handle growing traffic and content volume seamlessly. Additionally, its powerful querying and indexing capabilities facilitate fast retrieval of content, making it suitable for real-time content updates and interactive user experiences. Integration with modern web frameworks and APIs further enhances MongoDB's utility in developing robust and efficient CMS and blogging platforms.

- **Web Analytics or Real-Time Analytics** - MongoDB is well-suited for web analytics and real-time analytics applications due to its high write throughput, flexible schema design, and powerful aggregation framework. Its ability to ingest and store large volumes of unstructured and semi-structured data quickly makes it ideal for capturing web activity logs, user interactions, and other analytical data points. MongoDB's real-time capabilities allow for immediate processing and analysis of incoming data, providing timely insights and enabling dynamic dashboards and reports. The database's scalability ensures that it can handle the growing data needs of analytics platforms, while its integration with tools like Apache Kafka and Spark enhances its ability to support complex data processing and analysis pipelines, making it a robust choice for real-time and web analytics solutions.

- **E-Commerce Applications** - MongoDB is highly effective for e-commerce applications due to its flexible schema, which allows for dynamic and diverse product catalogs that can easily evolve with business needs. Its ability to handle large volumes of transactions with high availability and horizontal scalability ensures that the application can support growing user bases and increased traffic during peak times, such as sales events. MongoDB's powerful indexing and aggregation capabilities enable fast and efficient querying, crucial for providing real-time inventory updates, personalized recommendations, and advanced search features. Additionally, its document-oriented data model aligns well with the complex data structures inherent in e-commerce, such as customer profiles, order histories, and shopping carts, making MongoDB a robust and versatile choice for modern e-commerce platforms.

When not to use (Limitations)

- **Complex Transactions Spanning Different Operations** -MongoDB is not well-suited for complex transactions that span multiple operations across different collections or documents due to its limited support for multi-document ACID transactions. While MongoDB does offer support for multi-document transactions, it is not as robust as traditional relational databases when dealing with complex transactional workflows that require strict consistency and rollbacks across various entities. This limitation can pose challenges in scenarios requiring intricate financial transactions, inventory management, or other critical operations that demand guaranteed atomicity, consistency, isolation, and durability (ACID) across numerous operations. For such use cases, a relational database might be a more appropriate choice.

- **Queries against Varying Aggregate Structure-** MongoDB may not be the best choice for scenarios requiring complex queries against varying aggregate structures due to its document-oriented nature. While MongoDB supports aggregation through its powerful aggregation framework, the flexibility of its schema can lead to inconsistent data structures across documents, making it challenging to perform efficient queries or joins on heterogeneous data. In cases where the data structure varies significantly between records or requires frequent schema changes, querying and aggregating data can become inefficient and complex. Relational databases or specialized analytics platforms may be more suitable for handling complex queries against data with varying aggregate structures, as they provide a more rigid and standardized schema for optimized querying and aggregation.

Module-5

Graph Databases

Overview

Graph databases are specialized NoSQL databases that store data in the form of graphs, consisting of nodes (entities) and edges (relationships). They are designed to efficiently handle highly interconnected data, making them ideal for use cases where relationships between entities are central, such as social networks, recommendation systems, and fraud detection. Unlike traditional relational databases, graph databases explicitly define relationships, allowing for fast traversal and querying of complex, dynamic networks. Their flexible schema and ability to perform real-time queries on interconnected data make them a powerful tool for applications that require deep relationship analysis and pattern recognition. Popular graph databases include Neo4j, Amazon Neptune, and ArangoDB.

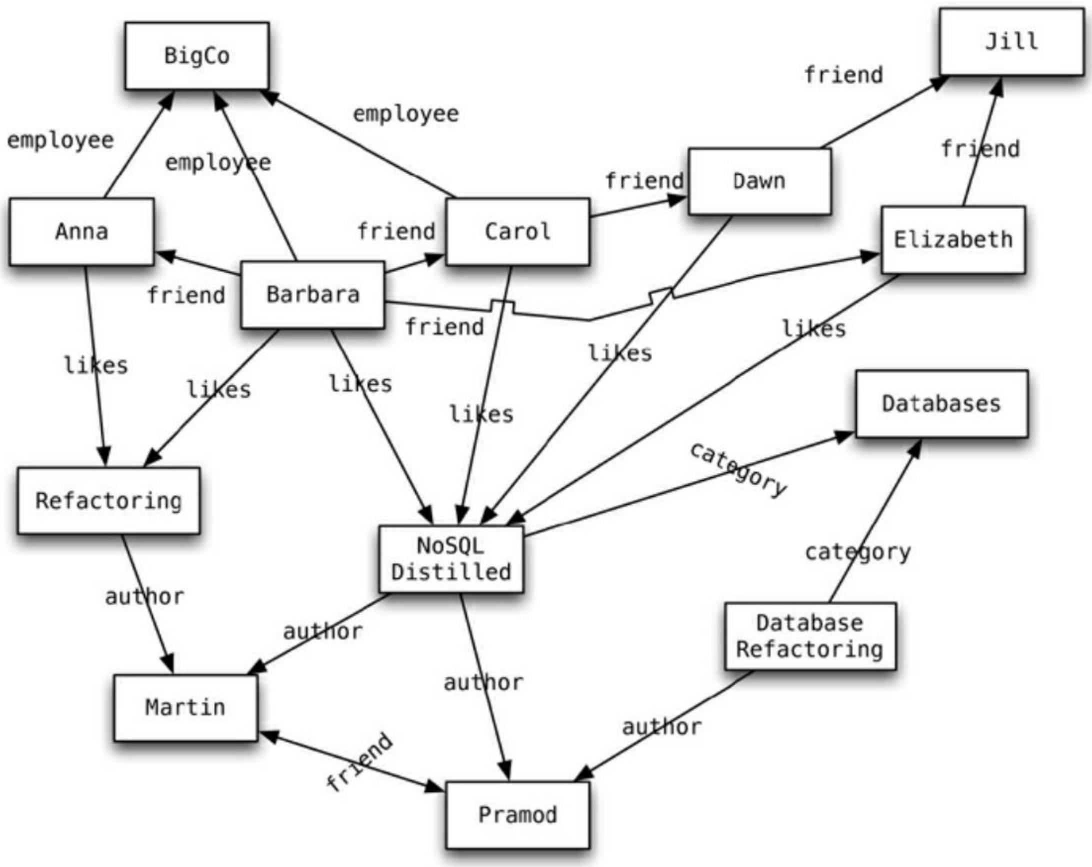
Terminologies of Graph databases

Graph databases use specific terminologies to represent and manage data. Here's an overview of key terms with examples:

1. **Node**: Represents an entity or object in the graph. Each node can have properties that store information about the entity.
   - **Example**: In a social network, a node might represent a **Person**, with properties like name, age, and location.
2. **Edge (Relationship)**: Represents a connection or relationship between two nodes. Each edge has a direction and can also have properties.
   - **Example**: In the social network, an edge could represent a **Friendship** between two **Person** nodes, with properties like since (the date the friendship started).
3. **Property**: Data attached to nodes or edges that provide additional details about them.
   - **Example**: A **Person** node could have properties like name = "Alice", age = 30, while an edge between Alice and Bob could have a property since = "2015".
4. **Graph**: The entire collection of nodes, edges, and their relationships that form a connected structure.
   - **Example**: The entire social network graph includes all users (nodes) and their relationships (edges), such as friendships, followers, or likes.
5. **Degree**: The number of edges connected to a node. It can be categorized as **in-degree** (edges pointing to the node) and **out-degree** (edges pointing away from the node).
   - **Example**: If Alice is friends with Bob, Carol, and Dave, the degree of Alice is 3 (three friendships).
6. **Traversal**: The process of exploring the graph by following edges from one node to another. Traversals are used to find paths, patterns, or connections in the graph.
   - **Example**: A query to find all friends of Alice and their friends (i.e., second-degree connections) would be a traversal of Alice's friends, followed by traversing their friends.
7. **Path**: A sequence of nodes connected by edges. It represents a route from one node to another.
   - **Example**: A path could be Alice → Bob → Carol, where each arrow represents an edge or relationship.
8. **Subgraph**: A subset of a graph that includes a subset of nodes and edges. It represents a portion of the overall graph, often used for analysis or exploration.

- **Example**: A subgraph could represent a group of friends (nodes) and their mutual relationships (edges) within a social network.

Example structure



Once we have a graph of these nodes and edges created, we can query the graph in many ways, such as “get all nodes employed by Big Co that like NoSQL Distilled.” A query on the graph is also known as **traversing** the graph. An advantage of the graph databases is that we can change the traversing requirements without having to change the nodes or edges. If we want to “get all nodes that like NoSQL Distilled,” we can do so without having to change the existing data or the model of the database, because we can traverse the graph any way we like.

Features

Graph databases offer several key features that make them highly effective for handling complex, interconnected data. Some of the main features include:

1. **Efficient Relationship Handling**: Graph databases are designed to handle relationships between data points as first-class citizens, allowing for fast and efficient querying of complex relationships, even when they involve multiple hops between nodes.
2. **Flexible Schema**: Unlike relational databases, graph databases have a flexible schema that allows nodes and edges to have varying structures, making them ideal for handling dynamic and evolving data with changing relationships.
3. **Real-Time Querying**: Graph databases support real-time traversal and querying of data, enabling quick insights into interconnected data. This is particularly beneficial for use cases like social networks, fraud detection, and recommendation systems where real-time performance is crucial.

4. **ACID Transactions**: Many graph databases support ACID (Atomicity, Consistency, Isolation, Durability) transactions, ensuring that operations on the graph are safe, reliable, and consistent.
5. **Graph Algorithms**: Graph databases come with built-in support for graph algorithms like shortest path, centrality, community detection, and PageRank, which are useful for analyzing relationships and patterns in the data.
6. **Intuitive Representation**: Graphs provide an intuitive way to model data that is naturally interconnected, such as social networks, supply chains, or recommendation engines, as it closely mirrors real-world relationships.
7. **Scalability**: Graph databases are designed to scale horizontally, meaning they can efficiently handle large volumes of interconnected data by distributing the data across multiple servers or nodes.
8. **Optimized for Traversal**: Graph databases are optimized for traversing nodes and edges, making them ideal for applications that need to explore or analyze complex relationships between data points.
9. **Deep and Complex Queries**: Graph databases allow for deep and complex queries that would be difficult to perform efficiently in relational databases, such as querying for multiple levels of connections, discovering hidden patterns, and exploring hierarchical structures.
10. **Built-in Indexing**: Graph databases automatically index nodes and edges based on their relationships, which improves the performance of querying and traversing large graphs.

These features make graph databases particularly suitable for applications in areas like social networks, fraud detection, recommendation systems, knowledge graphs, and network analysis, where relationships between entities are essential.

CRUD operations on NEO4J

Neo4j is a popular graph database that uses Cypher, a declarative query language, to perform CRUD (Create, Read, Update, Delete) operations on graph data. Here's an example of how to perform each of these operations in Neo4j using Cypher:

1. Create (C) - Adding nodes and relationships
To create nodes and relationships in Neo4j, you use the CREATE keyword.
Example: Create a Person node and a Movie node with a LIKES relationship between them.

```
CREATE (a:Person {name: 'Alice', age: 30}),
    (m:Movie {title: 'The Matrix', release_year: 1999}),
    (a)-[:LIKES]->(m);
```

This query creates:

- A Person node for Alice with properties name and age.
- A Movie node for "The Matrix" with properties title and release_year.
- A relationship LIKES from Alice to "The Matrix".

2. Read (R) - Retrieving nodes and relationships
To retrieve data, you use the MATCH keyword to specify the pattern you are looking for.
Example: Find all movies liked by Alice.

```
MATCH (a:Person {name: 'Alice'})-[:LIKES]->(m:Movie)
RETURN m.title, m.release_year;
```

This query finds the Movie nodes that are connected to the Person node representing Alice via a LIKES relationship and returns the titles and release years of those movies.

3. Update (U) - Modifying existing nodes and relationships
To update properties, use the SET keyword.
Example: Update Alice's age and add a new property to the LIKES relationship.

```
MATCH (a:Person {name: 'Alice'}), (m:Movie {title: 'The Matrix'})
SET a.age = 31,
   m.genre = 'Sci-Fi',
   (a)-[:LIKES]->(m).since = 2024;
```

This query updates:

- Alice's age to 31.
- Adds a genre property to the Movie node for "The Matrix".
- Adds a since property to the LIKES relationship indicating the year Alice liked the movie.

4. Delete (D) - Removing nodes and relationships
To delete nodes or relationships, use the DELETE keyword.
Example: Delete the relationship between Alice and "The Matrix".

```
MATCH (a:Person {name: 'Alice'})-[r:LIKES]->(m:Movie {title: 'The Matrix'})
DELETE r;
```

This query deletes the LIKES relationship between Alice and "The Matrix" while leaving the nodes intact.
Example: Delete a node (person) and ensure that related relationships are also deleted.

```
MATCH (a:Person {name: 'Alice'})
DETACH DELETE a;
```

This query deletes the Person node for Alice and also deletes any relationships connected to her.

Scaling in Graph databases

Application-level sharding in graph databases involves partitioning the data across multiple databases or nodes to achieve horizontal scalability. This is especially important for large-scale graph applications, where a single server may not be sufficient to handle the growing data or query demands. Sharding in graph databases differs from traditional relational databases because the relationships between data points (nodes and edges) play a critical role in determining how data is distributed. Here's how application-level sharding can be achieved for graph databases:

1. Sharding Strategy
The first step in application-level sharding is deciding on a strategy to partition the graph data. There are a few common approaches to this:

- Entity-based Sharding: The graph is divided by types of entities (e.g., Person, Movie, Product). Each entity type or group of related entities is stored on different nodes or databases.
    - Example: All Person nodes might be stored on one database or server, and all Movie nodes on another. Relationships like LIKES could then be stored across these databases, requiring queries to traverse between them.
- Range-based Sharding: Data is partitioned based on specific ranges of values for node properties (e.g., partitioning users by age, geographic location, or any other suitable property). This is often more suitable for applications that can divide the graph based on identifiable segments.
    - Example: You might shard users in a social network by the first letter of their name or by their geographical region.
- Graph Component Sharding: The graph is split based on connected components, where each subgraph (which could be a community or cluster) is stored on a different shard. This is particularly useful when the graph has distinct groups that don't share many relationships across groups.
    - Example: In a social network, one shard could contain all the users in a particular community or group of users that interact frequently, while another shard holds different groups.

2. Handling Relationships Between Shards

One of the challenges with graph sharding is how to manage relationships (edges) that span multiple shards. To maintain performance, strategies need to be implemented to minimize cross-shard traversal:

- Cross-shard Traversal: When relationships span multiple shards, the system must be able to efficiently locate the related data. This can be done using techniques like storing references or mappings between shards and routing queries to the appropriate shard.
    - Example: If a query needs to traverse from a Person node on one shard to a Movie node on another, the application can use metadata about shard locations to fetch the relevant data.
- Query Routing and Aggregation: The application layer needs to be responsible for query routing, ensuring that queries are sent to the appropriate shards based on the data being requested. Additionally, results from multiple shards may need to be aggregated to provide a unified response.
    - Example: A recommendation system may need to gather data from multiple shards to suggest friends based on mutual connections across different graph partitions.

3. Replication for Availability and Fault Tolerance

To ensure high availability, graph databases can use replication. In a sharded setup, each shard can have multiple replicas distributed across different servers, ensuring that if one server fails, the data can still be accessed from another replica.

- Replica Sets: Similar to MongoDB’s replica sets, graph databases like Neo4j can have replication mechanisms to provide fault tolerance and ensure that data is available even if one server or shard fails.

4. Consistency Models

In a distributed graph database setup, maintaining consistency across shards can be tricky, especially for graph-based operations that require ACID properties. Depending on the application’s needs, different consistency models can be used:

- Eventual Consistency: In scenarios where strict consistency isn't required for all operations, eventual consistency can be used, allowing for faster writes and lower overhead in managing consistency across distributed nodes.
- Strong Consistency: For applications that require strong consistency (e.g., financial transactions or other critical operations), distributed transactions across shards must be managed, ensuring data integrity and consistency.

5. Sharding in Graph Database Solutions
    - Neo4j: Neo4j supports a form of sharding through its clustering features, allowing for horizontal scaling by replicating graph data across different nodes. However, its sharding capabilities are limited in terms of true application-level sharding and require manual configuration to distribute data across nodes.
    - Amazon Neptune: Amazon Neptune, a managed graph database service, allows scaling through replication and partitioning. It automatically distributes data across multiple nodes and shards, enabling high availability and performance in large-scale graph applications.

Use cases of Graph databases

- **Connected data** - A use case of graph databases with connected data is in social network analysis, where users (nodes) are connected by relationships (edges) like friendships, followers, or group memberships. Graph databases are ideal for efficiently querying these connections, enabling features such as friend recommendations, community detection, and real-time suggestions based on mutual relationships. By leveraging graph traversal capabilities, social platforms can quickly find connections like "friends of friends" or suggest new content based on similar interests. The flexible schema of graph databases also allows for dynamic additions of new relationship types, making them well-suited for modeling and querying complex, interconnected data in social networks.

- **Routing, Dispatch, and Location-Based Services** - A use case of graph databases in routing, dispatch, and location-based services is in logistics and transportation management, where real-time route optimization and dispatching rely heavily on efficiently mapping and traversing networks of locations (nodes) and roads or routes (edges). In such systems, graph databases model roads, intersections, and routes as a network of nodes and edges, making it easy to calculate the shortest paths, optimal routes, or dispatch orders based on real-time conditions. For instance, a delivery service can use a graph database to dynamically calculate the best route for drivers, taking into account traffic, road closures, or time constraints. Similarly, location-based services like ride-sharing apps use graph databases to find the fastest paths for users and drivers, considering factors such as proximity, traffic, and available vehicles. The ability to perform complex, real-time traversal queries on this interconnected data allows for efficient routing, dispatching, and location-based recommendations, which is crucial for optimizing operations in logistics, transportation, and ride-sharing industries.

- **Recommendation Engines** - A use case of graph databases in recommendation engines is in e-commerce platforms where personalized product recommendations are crucial. Graph databases excel in this scenario by modeling the relationships between users, products, and their interactions (e.g., views, purchases, ratings) as a network of nodes and edges. For instance, users are represented as nodes, products as other nodes, and the interactions (such as "purchased," "viewed," or "liked") are edges connecting them.

By traversing this graph, the system can identify patterns like "users who bought this product also bought" or "products similar to those the user has liked." Graph databases allow for real-time, highly personalized recommendations based on the user's past behavior and the behaviors of similar users or products. This is particularly effective in uncovering hidden relationships and offering more relevant, context-aware suggestions, making them ideal for industries such as e-commerce, media streaming, and online content platforms.

When not to use (Limitations)

In some situations, graph databases may not appropriate. When you want to update all or a subset of entities—for example, in an analytics solution where all entities may need to be updated with a changed property—graph databases may not be optimal since changing a property on all the nodes is not a straightforward operation. Even if the data model works for the problem domain, some databases may be unable to handle lots of data, especially in global graph operations (those involving the whole graph)