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:

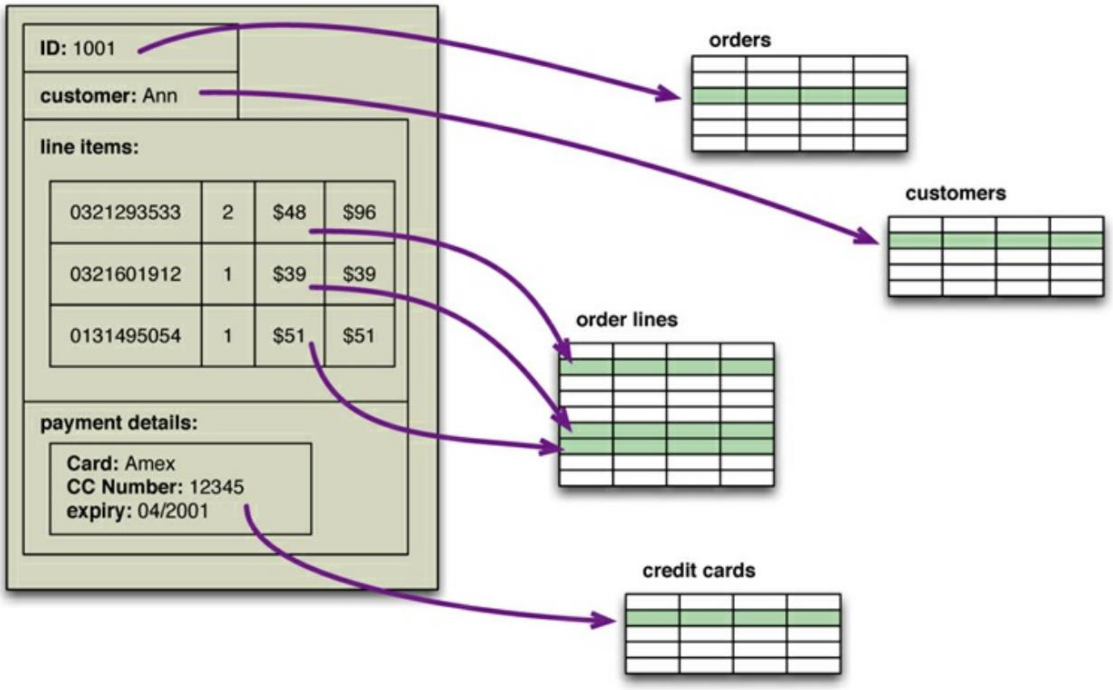


Figure 1.1. An order, which looks like a single aggregate structure in the UI, is split into many rows from many tables in a relational database

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

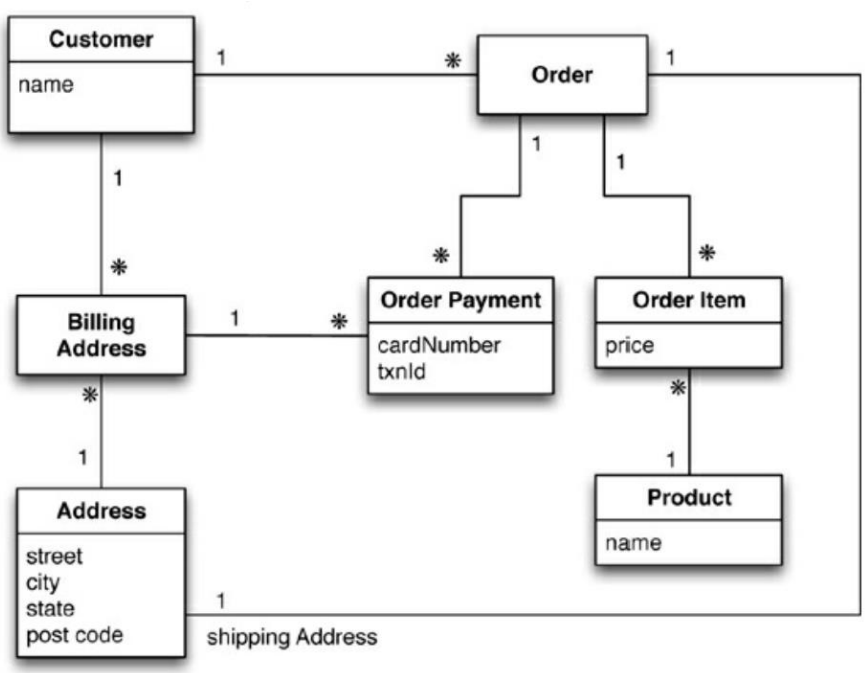
Aggregate data models are a foundational concept in NoSQL databases, particularly in document-oriented and key-value stores, where they enable the grouping of related data into cohesive units known as aggregates. An aggregate typically encompasses all the data related to a specific entity or concept, 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

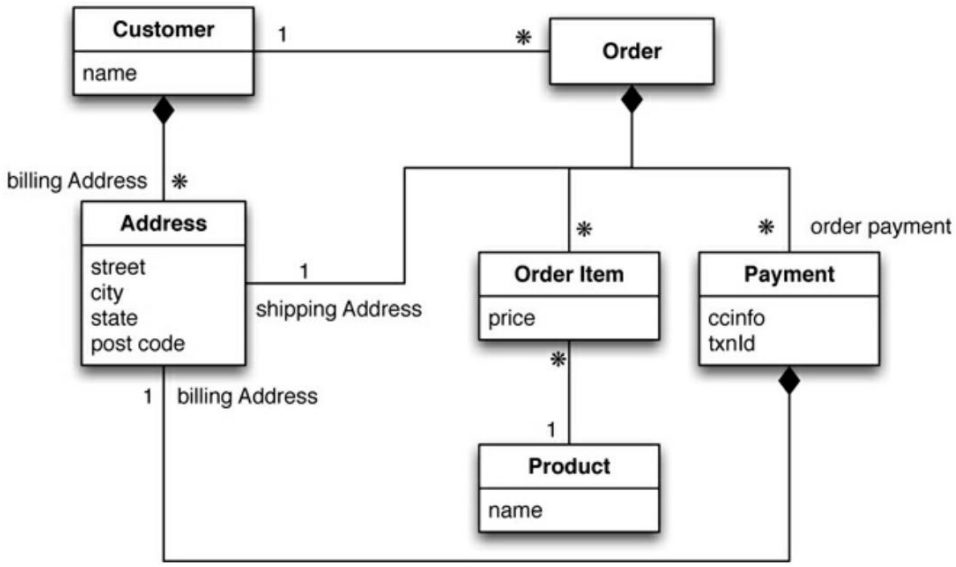
Id	City
77	Chicago

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:



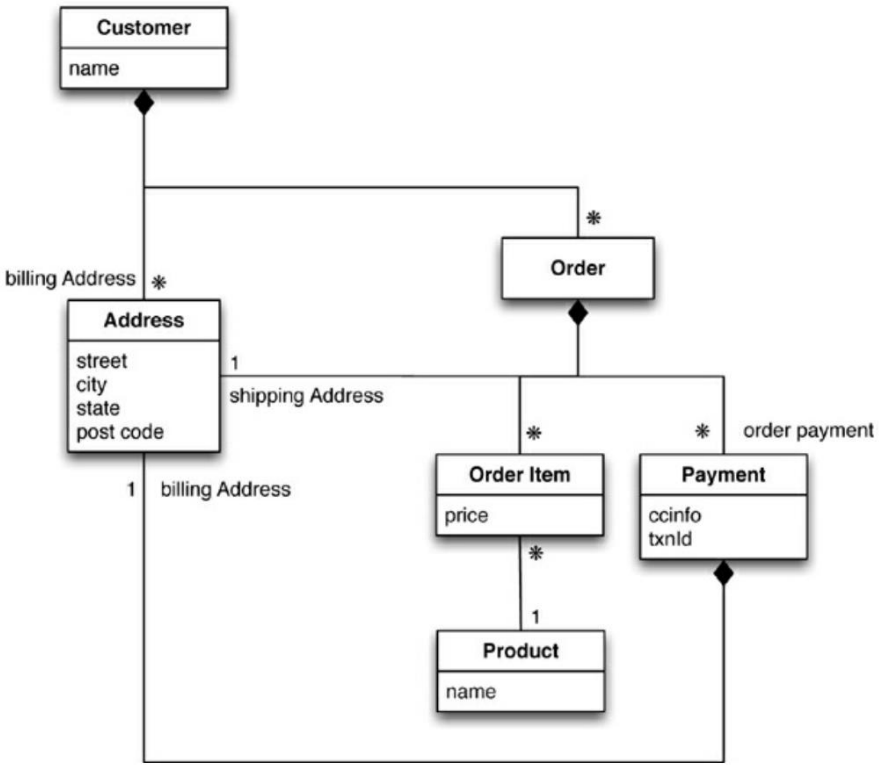
A single logical address record appears three times in the example data, but instead of using IDs it's treated as a value and copied each time. This fits the domain where we would not want the shipping address, nor the payment's billing address, to change. In a relational database, we would ensure that 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.

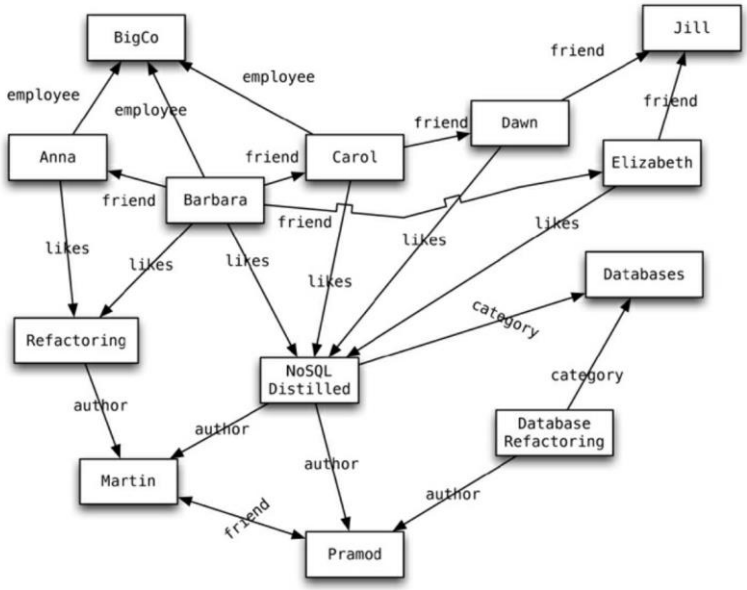
Aggregate Relationships

- 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)
  }}
```

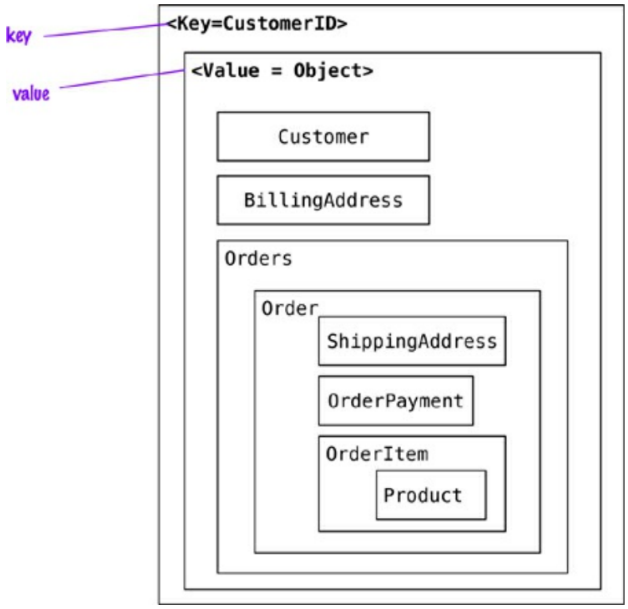
Materialized views

- 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.

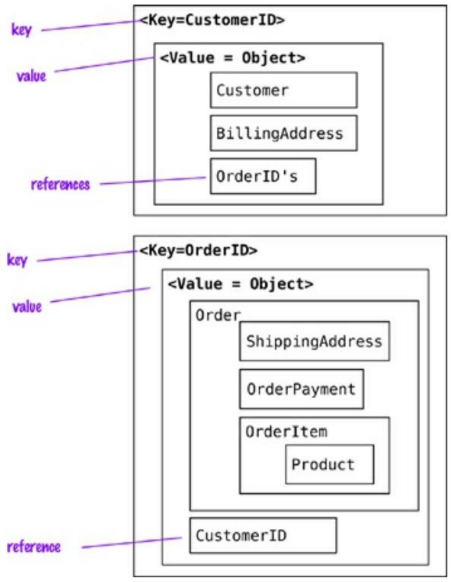
- By refreshing these views at specified intervals, either on demand or automatically, RDBMS can optimize read operations, significantly reducing the load on the underlying tables during heavy read queries.
- Materialized views are particularly useful in scenarios where data is frequently queried but 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

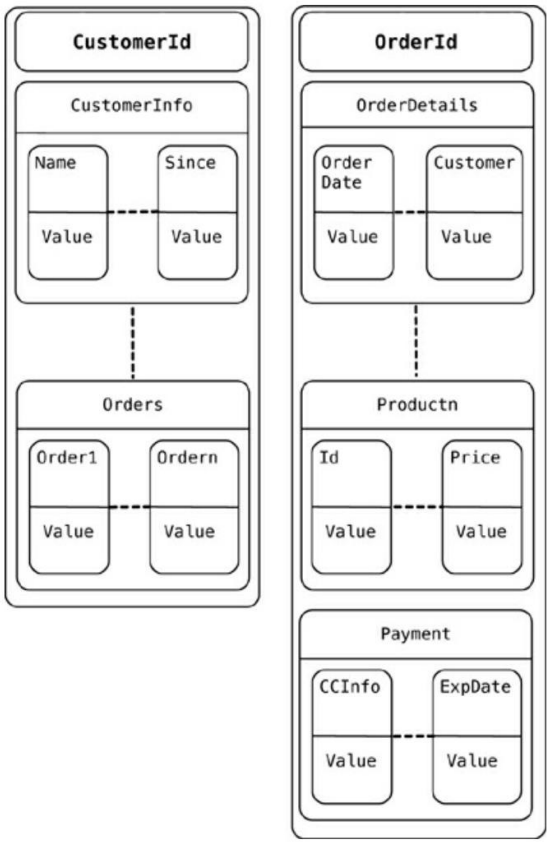
Embedding of customers and their orders using key-value stores



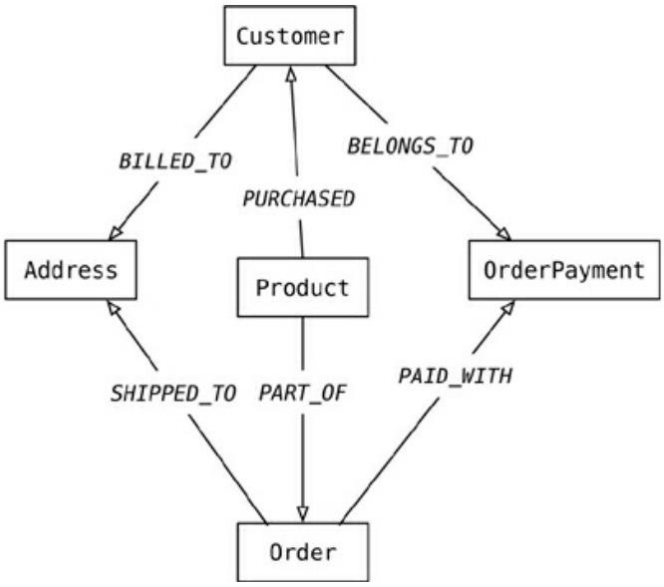
Storing of customers and orders separately



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

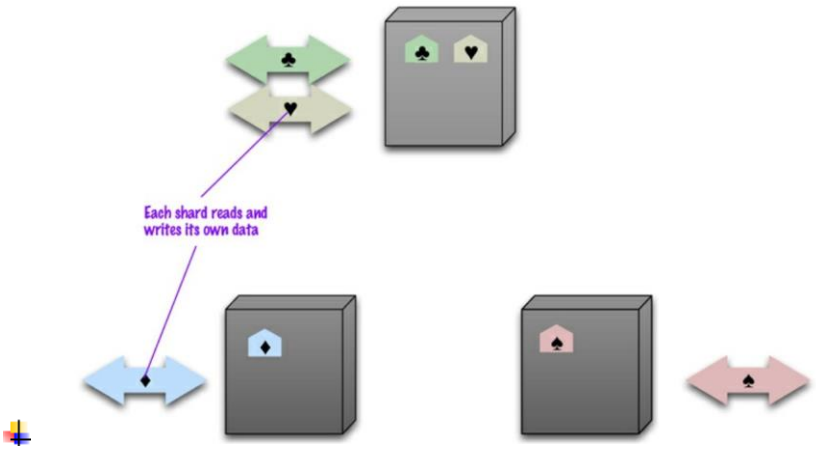
NoSQL databases employ various distribution models to manage data across multiple nodes or servers effectively. These models are designed to enhance scalability, availability, and performance, allowing NoSQL systems to handle large volumes of data and high levels of concurrent requests. Broadly, there 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

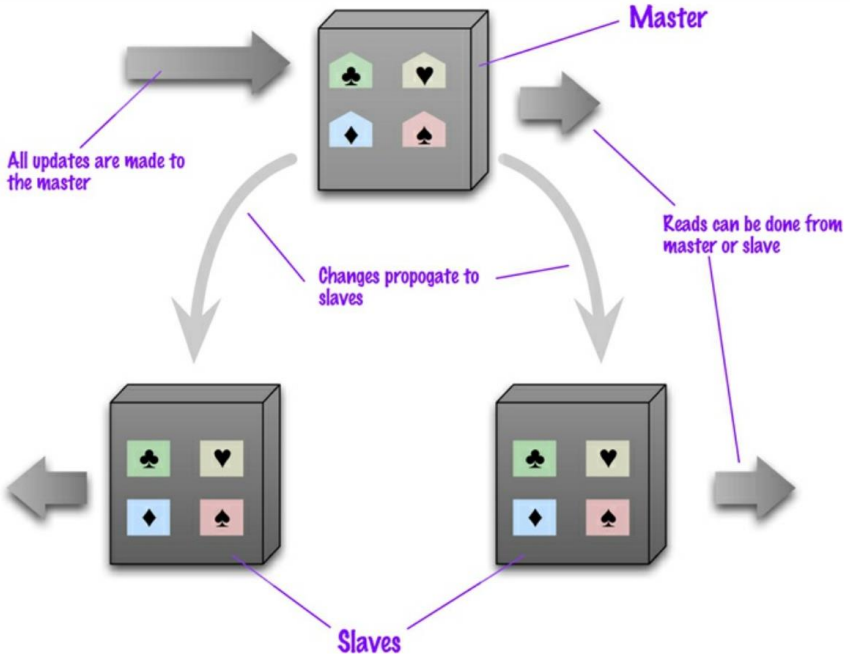
- Sharding is a database architecture pattern that involves partitioning data across multiple servers or nodes, enabling horizontal scaling and improving performance for large datasets. In a sharded database, data is divided into smaller, more manageable pieces called "shards," which are distributed 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:

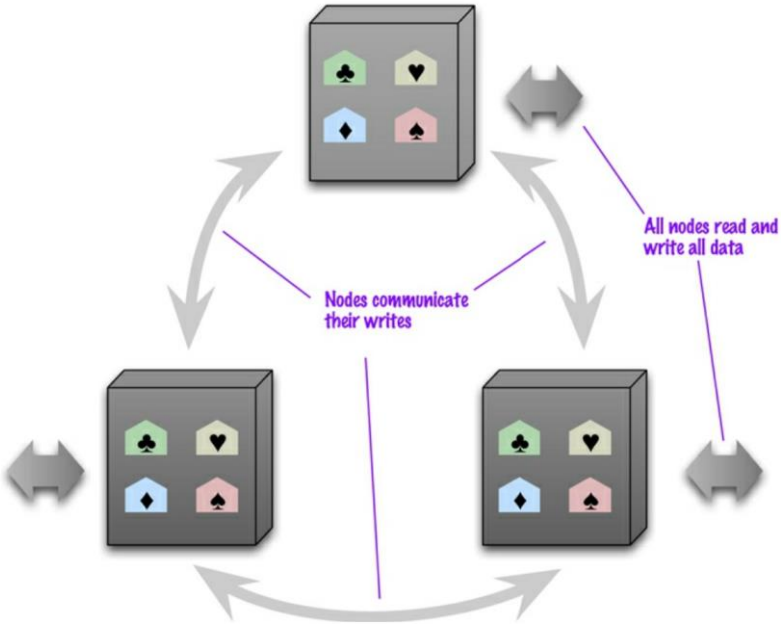


- Master-slave replication is most helpful for scaling when you have a read-intensive dataset. You can scale horizontally to handle more read requests by adding more slave nodes and ensuring that all read requests are routed to the slaves
- Read resilience in a master-slave database architecture refers 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:



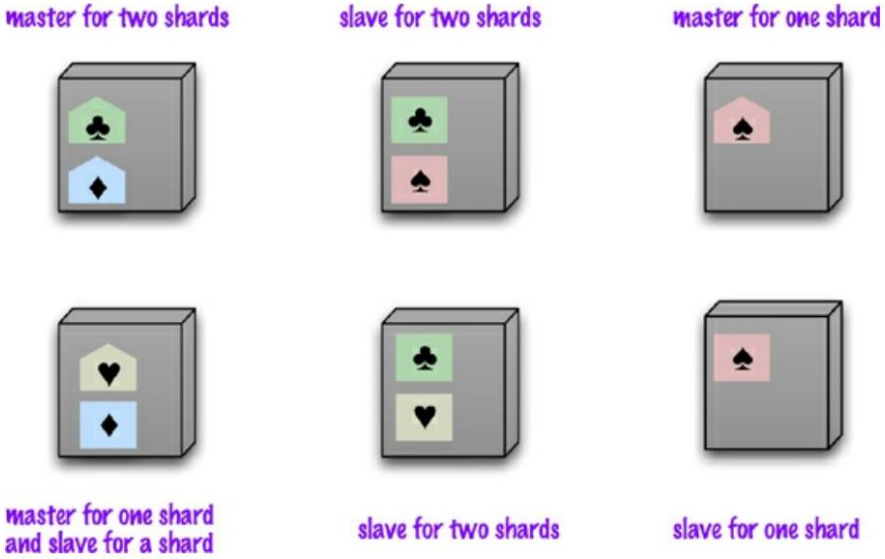
The biggest complication even with P2P replication is consistency. When you can write to two different places, you run the risk that two people will attempt to update the same record at the same time—a write-write conflict. Inconsistencies on read lead to problems but at least they are relatively transient.

Combining Sharding and Replication

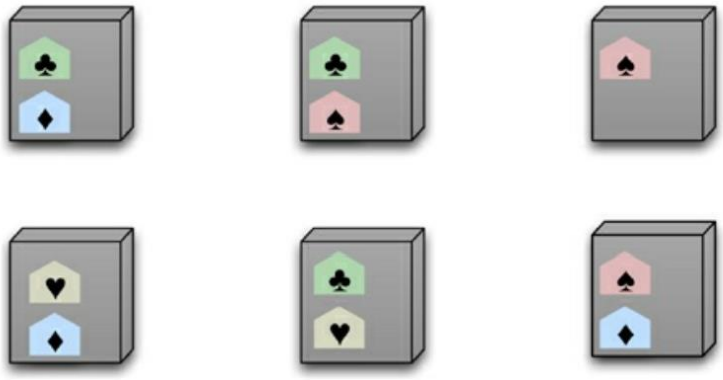
- Combining sharding and replication in a database architecture creates a robust solution that enhances both scalability and availability.
- In this hybrid model, data is partitioned into smaller, manageable shards that are distributed across multiple servers, enabling horizontal scaling to accommodate 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

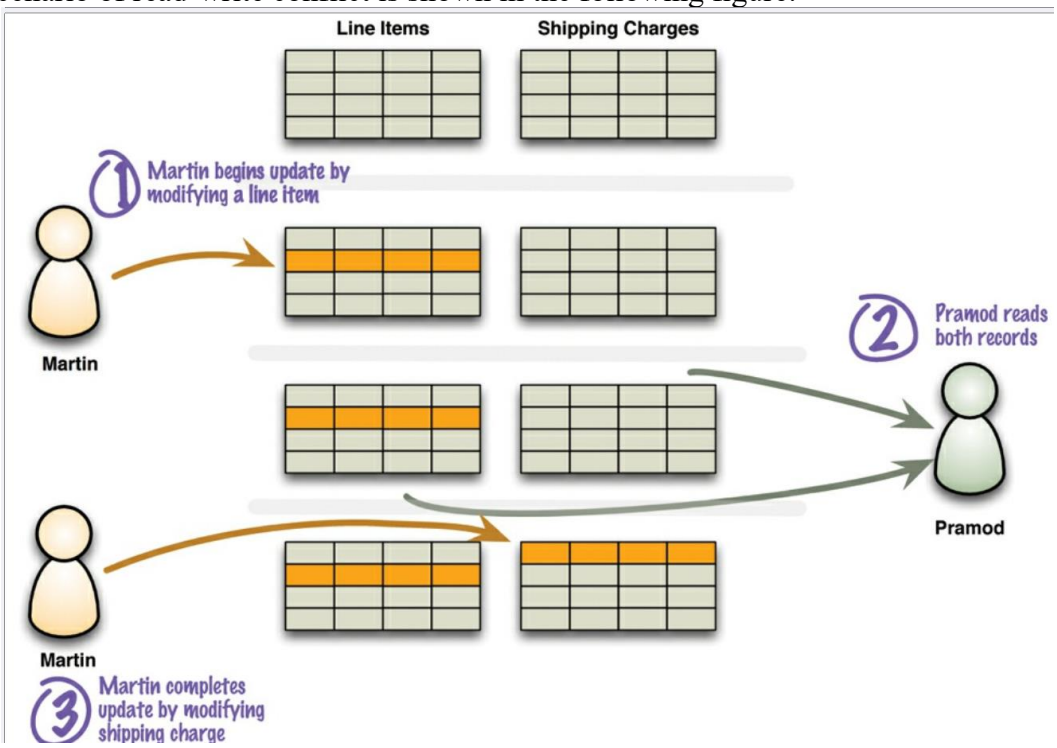
- Coincidentally, Martin and Pramod are looking at the company website and notice that the phone number is out of date. Implausibly, they both have update access, so they both go in at the same time to update the number. This issue is called a write-write conflict: two people updating the same data item 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

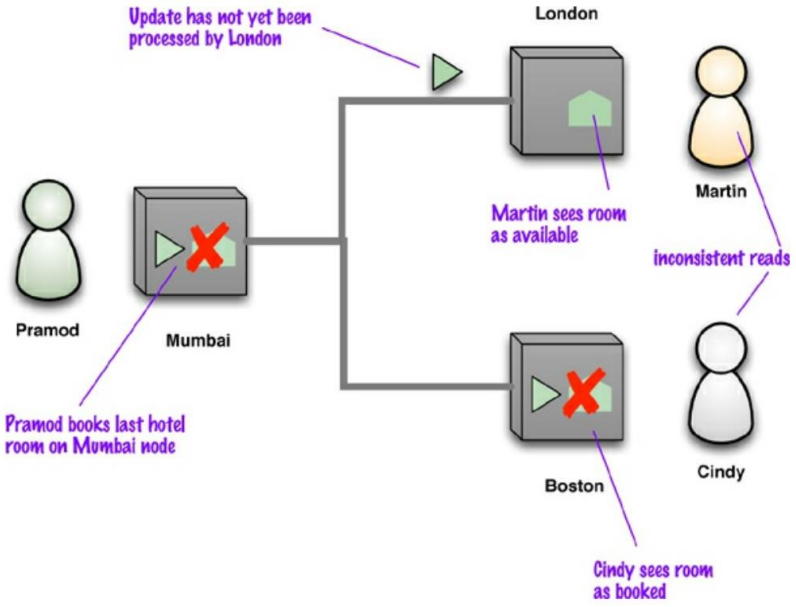
Let's imagine we have an order with line items and a shipping charge. The shipping charge is calculated based on the line items in the order. If we add a line item, we thus also need to recalculate and update the shipping charge. In a relational database, the shipping charge and line items will be in 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:



- 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:



- **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

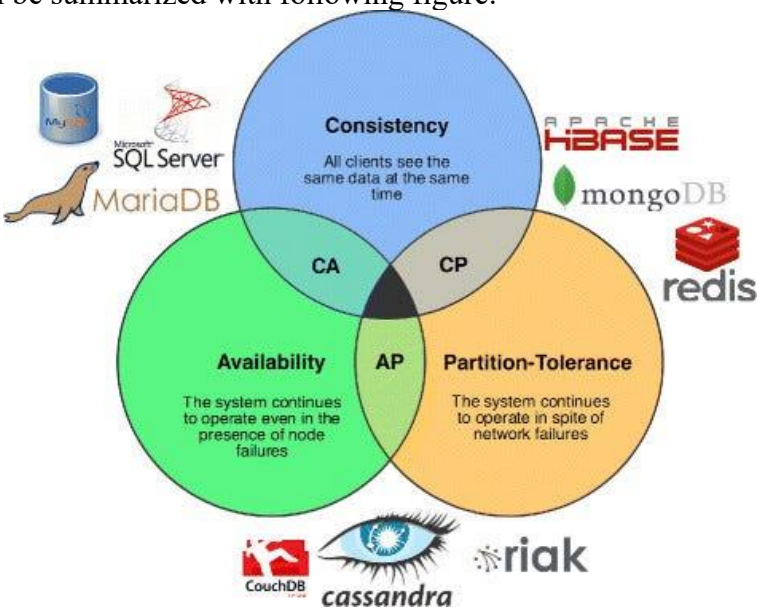
3. Partition Tolerance (P)

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:



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.