

Introduction to Databases

Understand what a database is and its use cases in the system design.

We'll cover the following



- Problem statement
 - Limitations of file storage
 - Solution
- Advantages
- How will we explain databases?

Problem statement

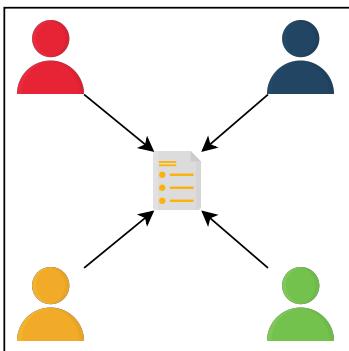
Let's start with a simple question. Can we make a software application without using databases? Let's suppose we have an application like WhatsApp. People use our application to communicate with their friends. Now, where and how can we store information (a list of people's names and their respective messages) permanently and retrieve it?

We can use a simple file to store all the records on separate lines and retrieve them from the same file. But using a file for storage has some limitations.

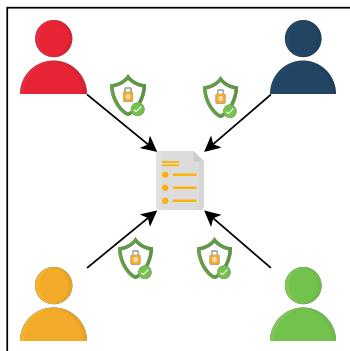
Limitations of file storage

- We can't offer concurrent management to separate users accessing the storage files from different locations.
- We can't grant different access rights to different users.
- How will the system scale and be available when adding thousands of entries?
- How will we search content for different users in a short time?

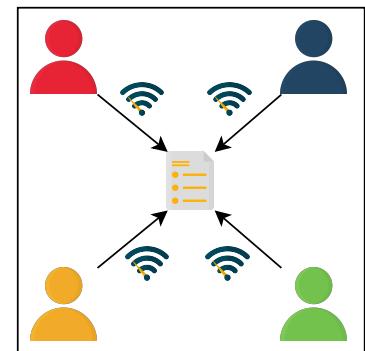




Concurrent access to data by multiple users



Different access rights to different users



Speed affected by millions of users

The limitations of file storage

Solution

The above limitations can be addressed using databases.

A **database** is an organized collection of data that can be managed and accessed easily. Databases are created to make it easier to store, retrieve, modify, and delete data in connection with different data-processing procedures.



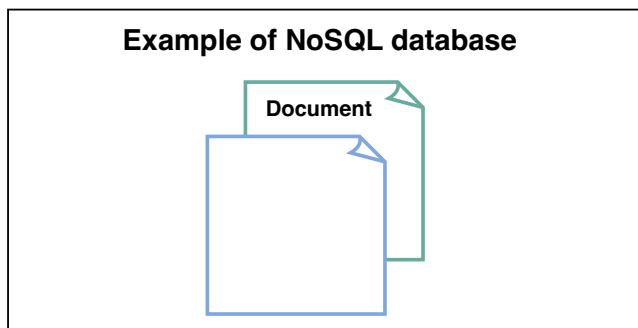
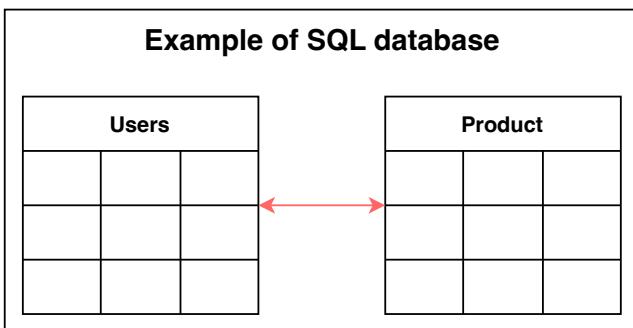
Some of the applications where we use database management are the banking systems, online shopping stores, and so on. Different organizations have different sizes of databases according to their needs.

Note: According to a [source](#), the World Data Center for Climate (WDCC) is the largest database in the world. It contains around 220 terabytes of web data and 6 petabytes of additional data.

There are two basic types of databases:

- SQL (relational databases)
- NoSQL (non-relational databases)

They differ in terms of their intended use case, the type of information they hold, and the storage method they employ.



Relational database has a well defined structure such as attributes (columns of the table). NoSQL databases such as document databases often have application-defined structure of data.

Relational databases, like phone books that record contact numbers and addresses, are organized and have predetermined schemas. Non-relational databases, like file directories that store anything from a person's constant information to shopping preferences, are unstructured, scattered, and feature a dynamic schema. We'll discuss their differences and their types in detail in the next lesson.

Advantages

A proper database is essential for every business or organization. This is because the database stores all essential information about the organization, such as personnel records, transactions, salary information, and so on. Following are some of the reasons why the database is important:

- **Managing large data:** A large amount of data can be easily handled with a database, which wouldn't be possible using other tools.
- **Retrieving accurate data (data consistency):** Due to different constraints in databases, we can retrieve accurate data whenever we want.
- **Easy updation:** It is quite easy to update data in databases using data manipulation language (DML).
- **Security:** Databases ensure the security of the data. A database only allows authorized users to access data.
- **Data integrity:** Databases ensure data integrity by using different constraints for data.
- **Availability:** Databases can be replicated (using [data replication](#)) on different servers, which can be concurrently updated. These replicas

ensure availability.

- **Scalability:** Databases are divided (using [data partitioning](#)) to manage the load on a single node. This increases scalability.

How will we explain databases?

We have divided the database chapter into four lessons:

1. [Types of Databases](#): We'll discuss the different types of databases, their advantages, and their disadvantages.
2. [Data Replication](#): We'll discuss what data replication is and its different models with their pros and cons.
3. [Data Partitioning](#): We'll discuss what data partitioning is and its different models with their pros and cons.
4. [Cost-benefit analysis](#): We'll discuss which database sharding approach is best for different kinds of databases.

Let's get started by understanding different types of databases and their preferred use cases.

[!\[\]\(c694a3ff3b077d76910920a6a1593ab4_img.jpg\) Back](#)

Advanced Details of Load Balancers

[Mark As Completed](#)

[Next !\[\]\(aa53ad6fea213b8b2226d3077e30533a_img.jpg\)](#)

Types of Databases

Types of Databases

Understand various types of databases and their use cases in system design.

We'll cover the following



- Relational databases
 - Why relational databases?
 - Flexibility
 - Reduced redundancy
 - Concurrency
 - Integration
 - Backup and disaster recovery
 - Drawback
 - Impedance mismatch
- Why non-relational (NoSQL) databases?
 - Types of NoSQL databases
 - Key-value database
 - Document database
 - Graph database
 - Columnar database
 - Drawbacks of NoSQL databases
 - Lack of standardization
 - Consistency
- Choose the right database
- Quiz

As we discussed earlier, databases are divided into two types: relational and non-relational. Let's discuss these types in detail.

Relational databases

Relational databases adhere to particular schemas before storing the data. The data stored in relational databases has prior structure. Mostly, this model organizes data into one or more relations (also called tables), with a unique key for each tuple (instance). Each entity of the data consists of instances and attributes, where instances are stored in rows, and the attributes of each instance are stored in columns. Since each tuple has a unique key, a tuple in one table can be linked to a tuple in other tables by storing the primary keys in other tables, generally known as foreign keys.

A Structure Query Language (SQL) is used for manipulating the database. This includes insertion, deletion, and retrieval of data.

There are various reasons for the popularity and dominance of relational databases, which include simplicity, robustness, flexibility, performance, scalability, and compatibility in managing generic data.

Relational databases provide the atomicity, consistency, isolation, and durability (**ACID**) properties to maintain the integrity of the database. ACID is a powerful abstraction that simplifies complex interactions with the data and hides many anomalies (like dirty reads, dirty writes, read skew, lost updates, write skew, and phantom reads) behind a simple transaction abort.

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

Let's discuss ACID in detail:

- **Atomicity:** A transaction is considered an atomic unit. Therefore, either all the statements within a transaction will successfully execute, or none of them will execute. If a statement fails within a transaction, it should be aborted and rolled back.
- **Consistency:** At any given time, the database should be in a consistent state, and it should remain in a consistent state after every transaction. For example, if multiple users want to view a record from the database, it should return a similar result each time.

- **Isolation:** In the case of multiple transactions running concurrently, they shouldn't be affected by each other. The final state of the database should be the same as the transactions were executed sequentially.
- **Durability:** The system should guarantee that completed transactions will survive permanently in the database even in system failure events.

Various database management systems (DBMS) are used to define relational database schema along with other operations, such as to store, retrieve, and run SQL queries on data. Some of the popular DBMS are as follows:

- MySQL
- Oracle Database
- Microsoft SQL Server
- IBM DB2
- Postgres
- SQLite

Why relational databases?

Relational databases are the default choices of software professionals for structured data storage. There are a number of advantages to these databases. One of the greatest powers of the relational database is its abstractions of ACID transactions and related programming semantics. This make it very convenient for the end-programmer to use a relational database. Let's revisit some important features of relational databases:

Flexibility

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

Reduced redundancy

One of the biggest advantages of the relational database is that it eliminates data redundancy. The information related to a specific entity appears in one table while the relevant data to that specific entity appears in the other tables linked

through foreign keys. This process is called normalization and has the additional benefit of removing an inconsistent dependency.

Concurrency

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

Integration

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

Backup and disaster recovery

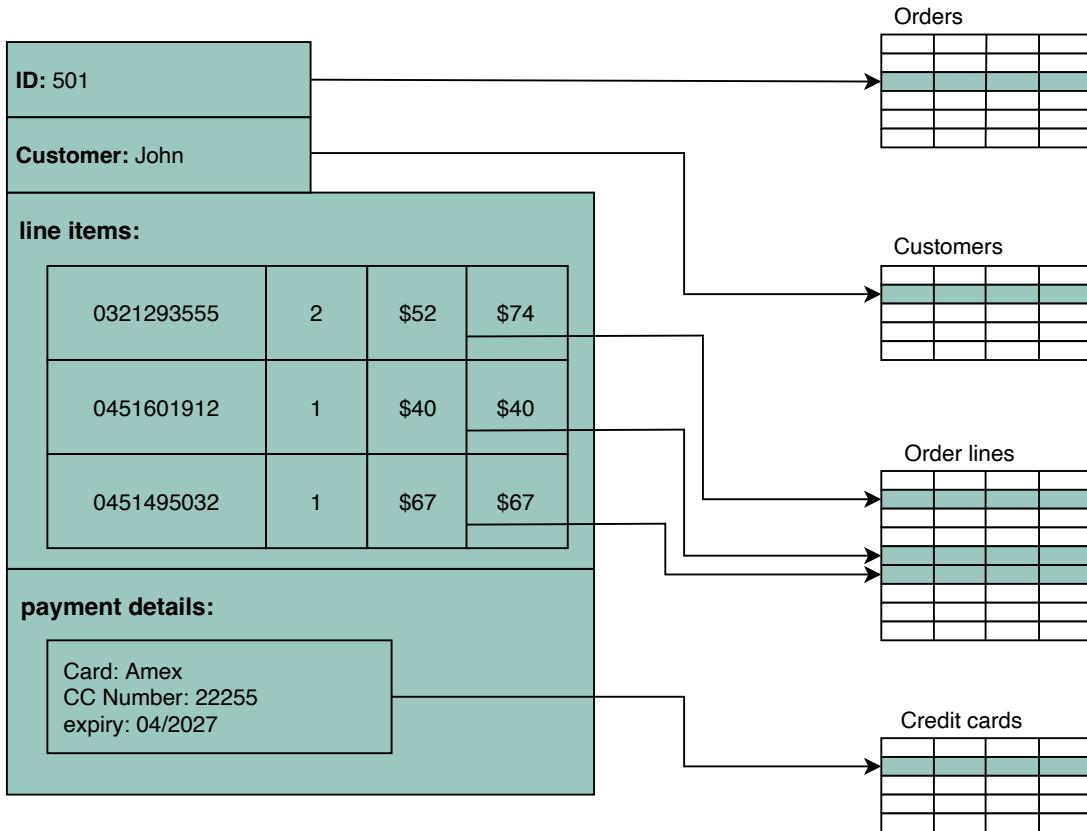
Relational databases guarantee the state of data is consistent at any time. The export and import operations make backup and restoration easier. Most cloud-based relational databases perform continuous mirroring to avoid loss of data and make the restoration process easier and quicker.

Drawback

Impedance mismatch

Impedance mismatch is the difference between the relational model and the in-memory data structures. The relational model organizes data into a tabular structure with relations and tuples. SQL operation on this structured data yields relations aligned with relational algebra. However, it has some limitations. In particular, the values in a table take simple values that can't be a structure or a list. The case is different for in-memory, where a complex data structure can be

stored. To make the complex structures compatible with the relations, we would need a translation of the data in light of relational algebra. So, the impedance mismatch requires translation between two representations, as denoted in the following figure:



A single aggregated value in the view is composed of several rows and tables in the relational database

Why non-relational (NoSQL) databases?

A NoSQL database is designed for a variety of data models to access and manage data. There are various types of NoSQL databases, which we'll explain in the next section. These databases are used in applications that require a large volume of semi-structured and unstructured data, low latency, and flexible data models. This can be achieved by relaxing some of the data consistency restrictions of other databases. Following are some characteristics of the NoSQL database:

- **Simple design:** Unlike relational databases, NoSQL doesn't require dealing with the impedance mismatch—for example, storing all the employees' data in one document instead of multiple tables that require join

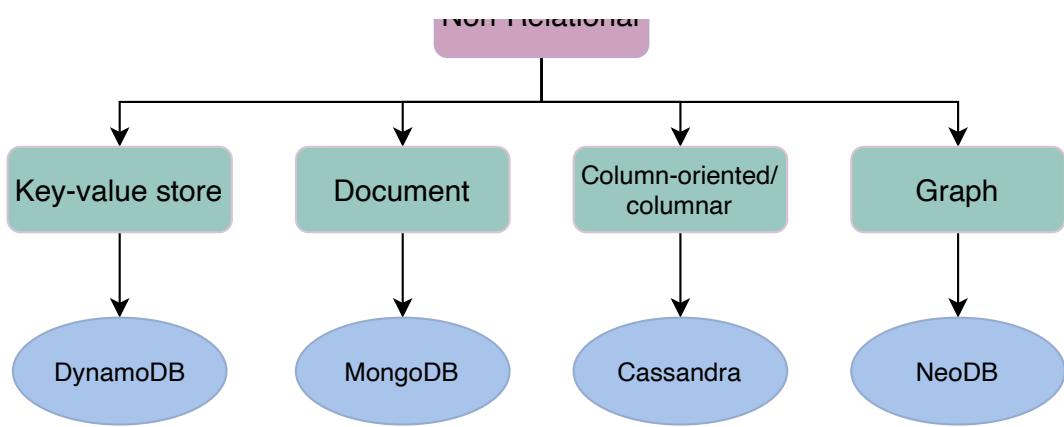
operations. This strategy makes it simple and easier to write less code, debug, and maintain.

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

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

Types of NoSQL databases

Various types of NoSQL databases are described below:



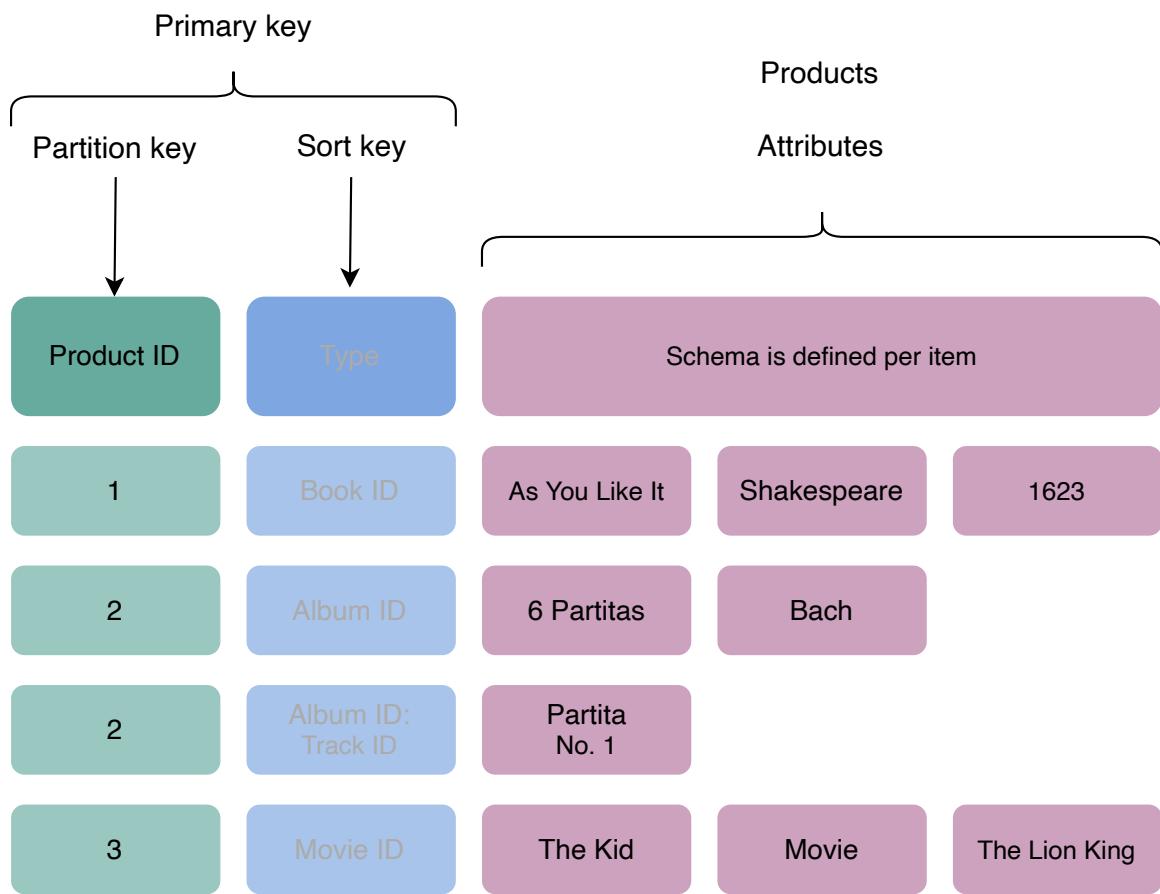
The types of NoSQL databases

Key-value database

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

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

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



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

Document database

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

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

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

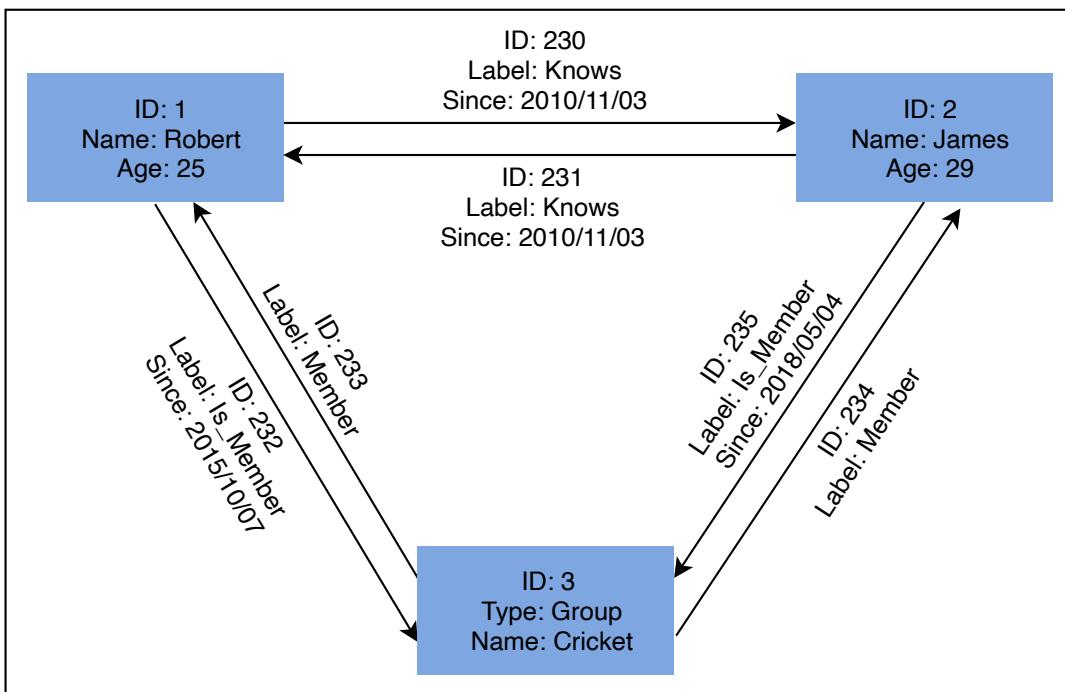
```
1 { "id": 1001,
2   "name": "Brown",
3   "title": "Mr.",
4   "email": "brown@anyEmail.com",
5   "cell": "123-465-9999",
6   "likes": [
7     "designing",
8     "cycling",
9     "skiing"],
10  "businesses": [
11    { "name": "ABC co.",
12      "partner": "Vike",
13      "status": "Bankrupt",
14      "date_founded": {
15        "$date": "2021-12-10" } }]
```

A JSON file containing data of a businessman

Graph database

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

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



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

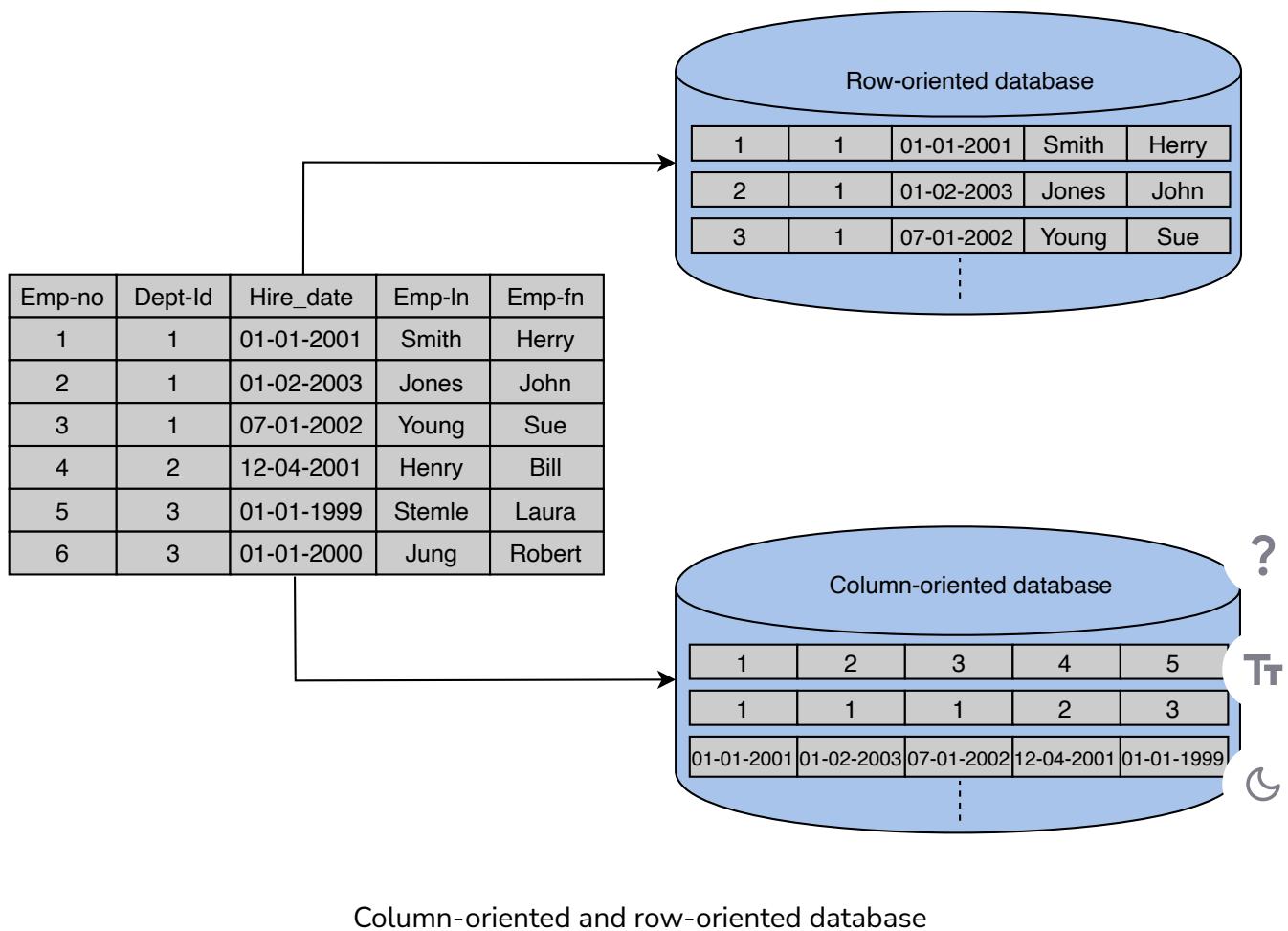
Use case: Graph databases can be used in social applications and provide interesting facts and figures among different kinds of users and their activities. The focus of graph databases is to store data and pave the way to drive analyses and decisions based on relationships between entities. The nature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

Columnar database

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

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

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



Drawbacks of NoSQL databases

Lack of standardization

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

Consistency

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

Choose the right database

Various factors affect the choice of database to be used in an application. A comparison between the relational and non-relational databases is shown in the following table to help us choose:

Relational and Non-relational Databases

Relational Database	Non-relational Database
If the data to be stored is structured	If the data to be stored is unstructured
If ACID properties are required	If there's a need to serialize and deserialize data
If the size of the data is relatively small and can fit on a node)	If the size of the data to be stored is large

Note: When NoSQL databases first came into being, they were drastically different to program and use as compared to traditional databases. Though, due to extensive research in academia and industry over the last many years, the programmer-facing differences between NoSQL and traditional stores are blurring. We might be using the same SQL constructs to talk to a NoSQL store and get a similar level of performance and consistency as a traditional store. [Google's Cloud Spanner](#) is one such database that's geo-replicated with automatic horizontal sharding ability and high-speed global snapshots of data.

Quiz

Test your knowledge of the different types of databases via a quiz.

1

Which database should we use when we have unstructured data and there's a need for high performance?

[Reset Quiz](#)



Question 1 of 6
0 attempted



[Submit Answer](#)

[← Back](#)

[Mark As Completed](#)

[Next →](#)

Introduction to Databases

Data Replication

Data Replication

Understand the models through which data is replicated across several nodes.

We'll cover the following



- Replication
- Synchronous versus asynchronous replication
- Data replication models
 - Single leader/primary-secondary replication
 - Primary-secondary replication methods
 - Statement-based replication
 - Write-ahead log (WAL) shipping
 - Logical (row-based) log replication
 - Multi-leader replication
 - Conflict
 - Handle conflicts
 - Conflict avoidance
 - Last-write-wins
 - Custom logic
 - Multi-leader replication topologies
 - Peer-to-peer/leaderless replication
 - Quorums

Data is an asset for an organization because it drives the whole business. Data provides critical business insights into what's important and what needs to be changed. Organizations also need to securely save and serve their clients' data on demand. Timely access to the required data under varying conditions (increasing reads and writes, disks and node failures, network and power outages, and so on) is required to successfully run an online business.

We need the following characteristics from our data store:

- Availability under faults (failure of some disk, nodes, and network and power outages).
- Scalability (with increasing reads, writes, and other operations).
- Performance (low latency and high throughput for the clients).

It's challenging, or even impossible, to achieve the above characteristics on a single node.

Replication

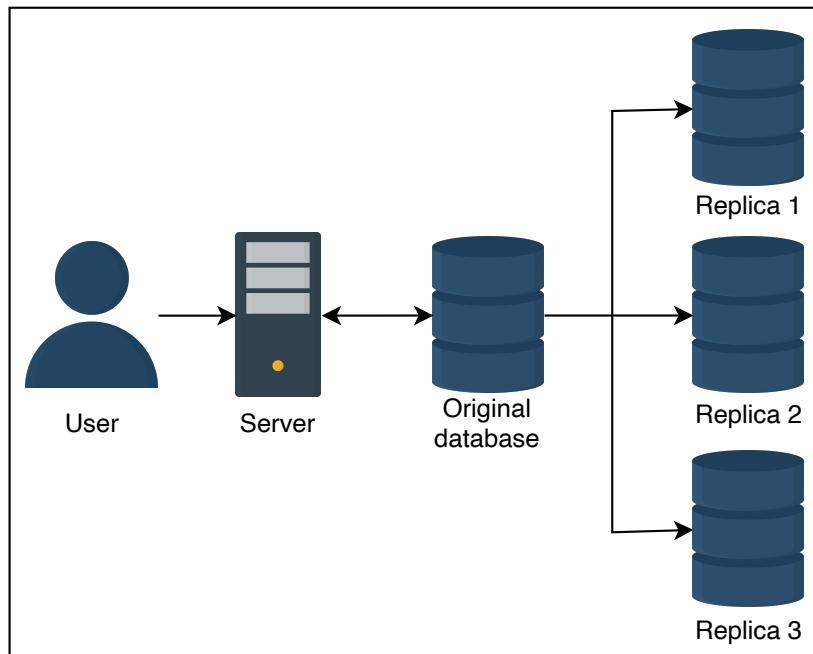
Replication refers to keeping multiple copies of the data at various nodes (preferably geographically distributed) to achieve availability, scalability, and performance. In this lesson, we assume that a single node is enough to hold our entire data. We won't use this assumption while discussing the partitioning of data in multiple nodes. Often, the concepts of replication and partitioning go together.

However, with many benefits, like availability, replication comes with its complexities. Replication is relatively simple if the replicated data doesn't require frequent changes. The main problem in replication arises when we have to maintain changes in the replicated data over time.

Additional complexities that could arise due to replication are as follows:

- How do we keep multiple copies of data consistent with each other?
- How do we deal with failed replica nodes?
- Should we replicate synchronously or asynchronously?
 - How do we deal with replication lag in case of asynchronous replication?
- How do we handle concurrent writes?
- What consistency model needs to be exposed to the end programmers?

We'll explore the answer to these questions in this lesson.



Replication in action

Before we explain the different types of replication, let's understand the synchronous and asynchronous approaches of replication.

Synchronous versus asynchronous replication

There are two ways to disseminate changes to the replica nodes:

- Synchronous replication
- Asynchronous replication

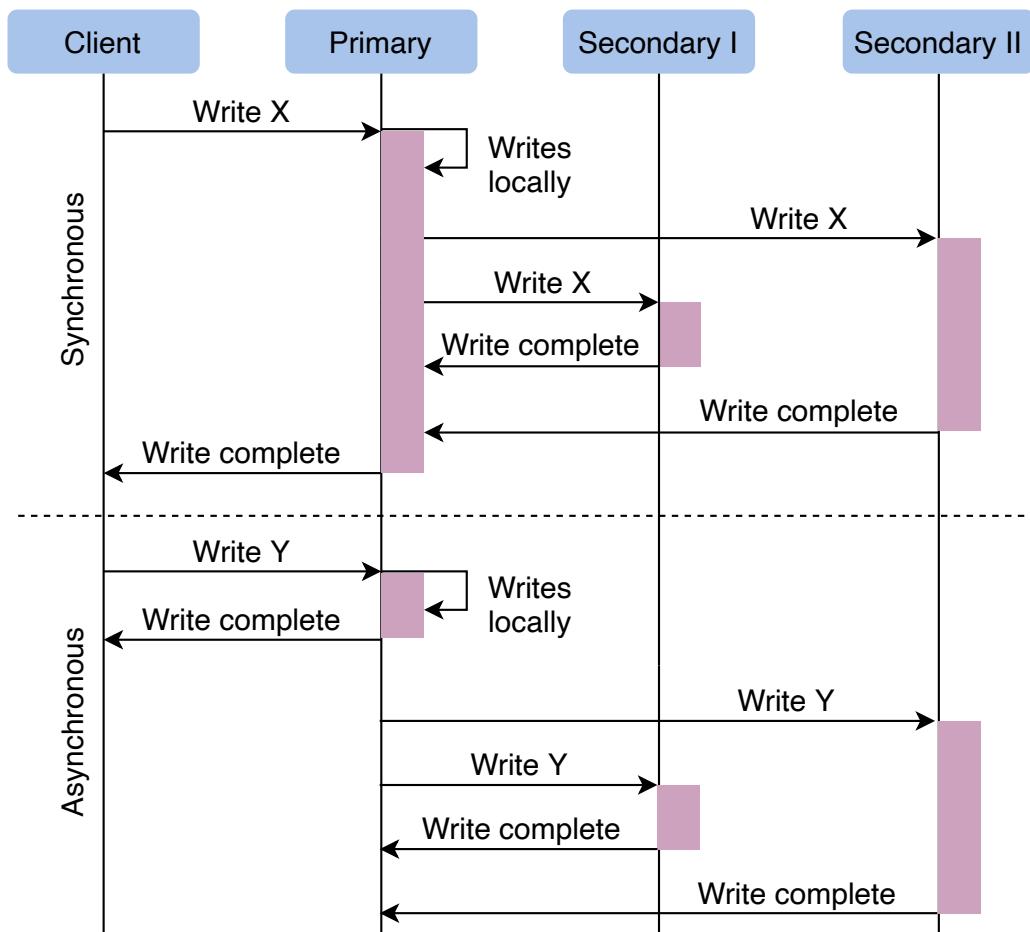
In **synchronous replication**, the primary node waits for acknowledgments from secondary nodes about updating the data. After receiving acknowledgment from all secondary nodes, the primary node reports success to the client. Whereas in **asynchronous replication**, the primary node doesn't wait for the acknowledgment from the secondary nodes and reports success to the client after updating itself.

The advantage of synchronous replication is that all the secondary nodes are completely up to date with the primary node. However, there's a disadvantage to this approach. If one of the secondary nodes doesn't acknowledge due to failure or fault in the network, the primary node would be unable to acknowledge the client until it receives the successful acknowledgment from the crashed node.

This causes high latency in the response from the primary node to the client.

On the other hand, the advantage of asynchronous replication is that the primary node can continue its work even if all the secondary nodes are down. However, if the primary node fails, the writes that weren't copied to the secondary nodes will be lost.

The above paragraph explains a trade-off between data consistency and availability when different components of the system can fail.



Synchronous versus asynchronous replication

Data replication models

Now, let's discuss various mechanisms of data replication. In this section, we'll discuss the following models along with their strengths and weaknesses:

- Single leader or primary-secondary replication
- Multi-leader replication
- Peer-to-peer or leaderless replication

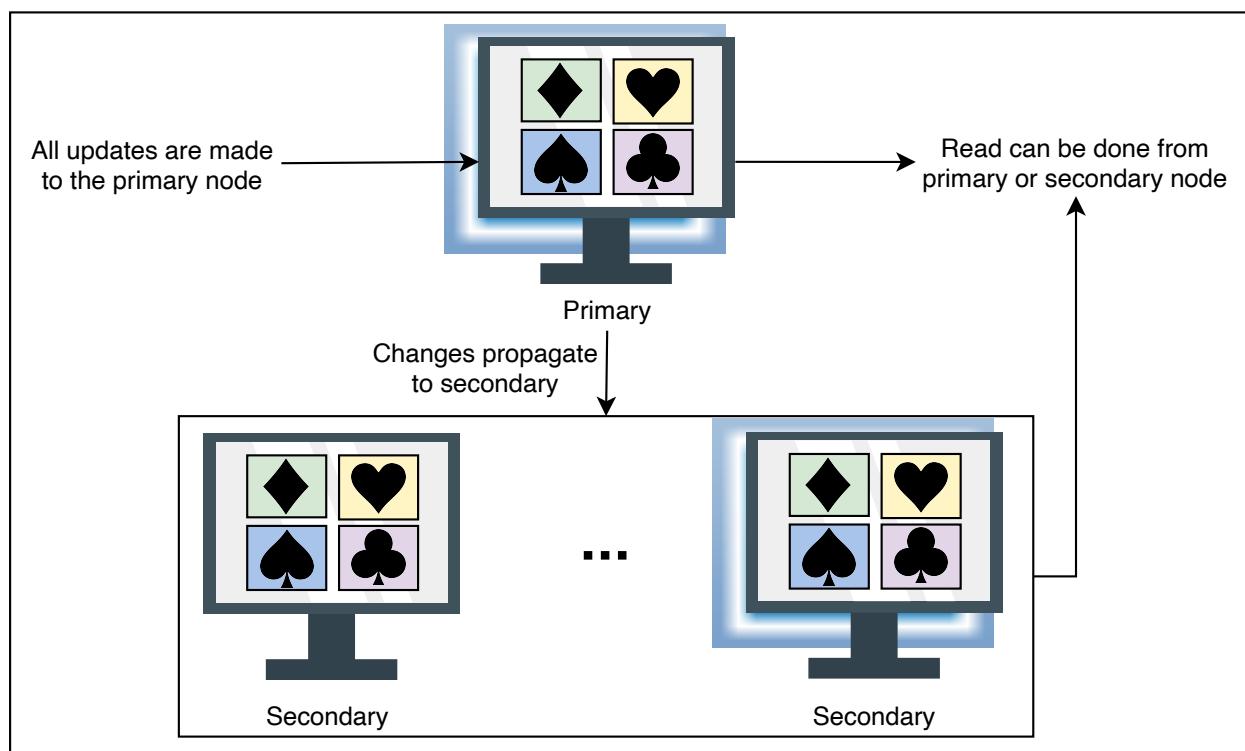
Single leader/primary-secondary replication

In **primary-secondary replication**, data is replicated across multiple nodes. One node is designated as the primary. It's responsible for processing any writes to data stored on the cluster. It also sends all the writes to the secondary nodes and keeps them in sync.

Primary-secondary replication is appropriate when our workload is read-heavy. To better scale with increasing readers, we can add more followers and distribute the read load across the available followers. However, replicating data to many followers can make a primary bottleneck. Additionally, primary-secondary replication is inappropriate if our workload is write-heavy.

Another advantage of primary-secondary replication is that it's read resilient. Secondary nodes can still handle read requests in case of primary node failure. Therefore, it's a helpful approach for read-intensive applications.

Replication via this approach comes with inconsistency if we use asynchronous replication. Clients reading from different replicas may see inconsistent data in the case of failure of the primary node that couldn't propagate updated data to the secondary nodes. So, if the primary node fails, any missed updates not passed on to the secondary nodes can be lost.



Primary-secondary data replication model where data is replicated from primary to secondary

Point to Ponder

Question

What happens when the primary node fails?

[Hide Answer](#) ^

In case of failure of the primary node, a secondary node can be appointed as a primary node, which speeds up the process of recovering the initial primary node. There are two approaches to select the new primary node: manual and automatic.

In a **manual approach**, an operator decides which node should be the primary node and notifies all secondary nodes.

In an **automatic approach**, when secondary nodes find out that the primary node has failed, they appoint the new primary node by conducting an election known as a leader election.

Primary-secondary replication methods

There are many different replication methods in primary-secondary replication:

- Statement-based replication
- Write-ahead log (WAL) shipping
- Logical (row-based) log replication

Let's discuss each of them in detail.

Statement-based replication

In the **statement-based replication** approach, the primary node saves all statements that it executes, like insert, delete, update, and so on, and sends them to the secondary nodes to perform. This type of replication was used in MySQL before version 5.1.

This type of approach seems good, but it has its disadvantages. For example, any nondeterministic function (such as `NOW()`) might result in distinct writes on the follower and leader. Furthermore, if a write statement is dependent on a prior write, and both of them reach the follower in the wrong order, the outcome on the follower node will be uncertain.

Write-ahead log (WAL) shipping

In the **write-ahead log (WAL) shipping** approach, the primary node saves the query before executing it in a log file known as a write-ahead log file. It then uses these logs to copy the data onto the secondary nodes. This is used in PostgreSQL and Oracle. The problem with WAL is that it only defines data at a very low level. It's tightly coupled with the inner structure of the database engine, which makes upgrading software on the leader and followers complicated.

Logical (row-based) log replication

In the **logical (row-based) log replication** approach, all secondary nodes replicate the actual data changes. For example, if a row is inserted or deleted in a table, the secondary nodes will replicate that change in that specific table. The binary log records change to database tables on the primary node at the record level. To create a replica of the primary node, the secondary node reads this data and changes its records accordingly. Row-based replication doesn't have the same difficulties as WAL because it doesn't require information about data layout inside the database engine.

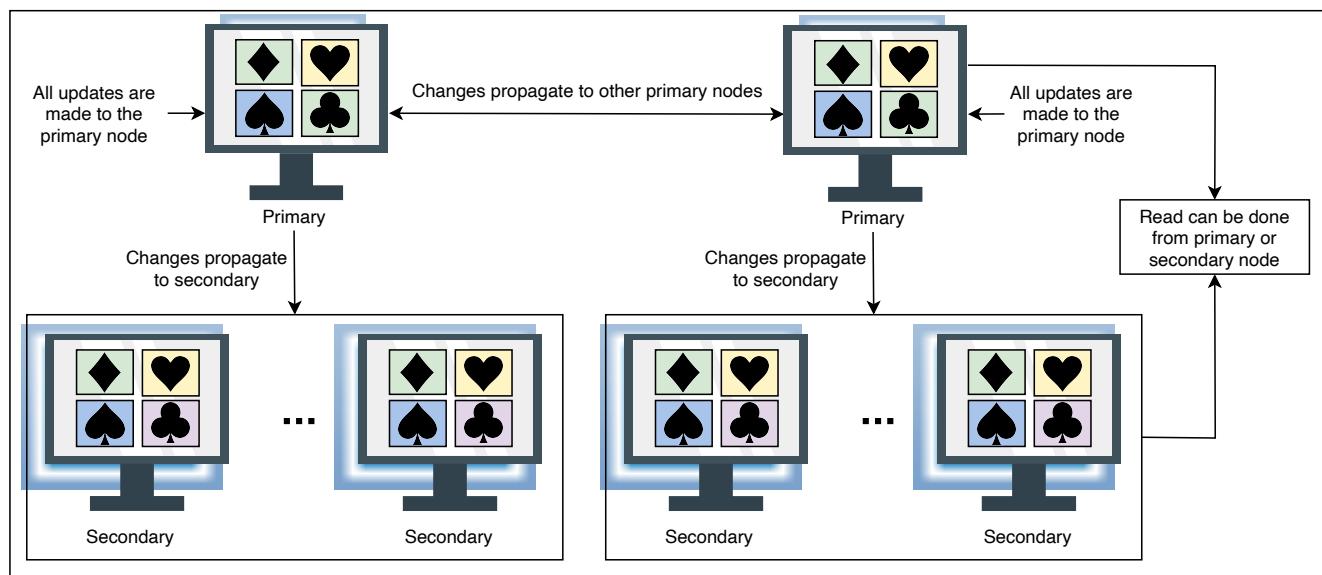
Multi-leader replication

As discussed above, single leader replication using asynchronous replication has a drawback. There's only one primary node, and all the writes have to go through it, which limits the performance. In case of failure of the primary node,

the secondary nodes may not have the updated database.

Multi-leader replication is an alternative to single leader replication. There are multiple primary nodes that process the writes and send them to all other primary and secondary nodes to replicate. This type of replication is used in databases along with external tools like the Tungsten Replicator for MySQL.

This kind of replication is quite useful in applications in which we can continue work even if we're offline—for example, a calendar application in which we can set our meetings even if we don't have access to the internet. Once we're online, it replicates its changes from our local database (our mobile phone or laptop acts as a primary node) to other nodes.



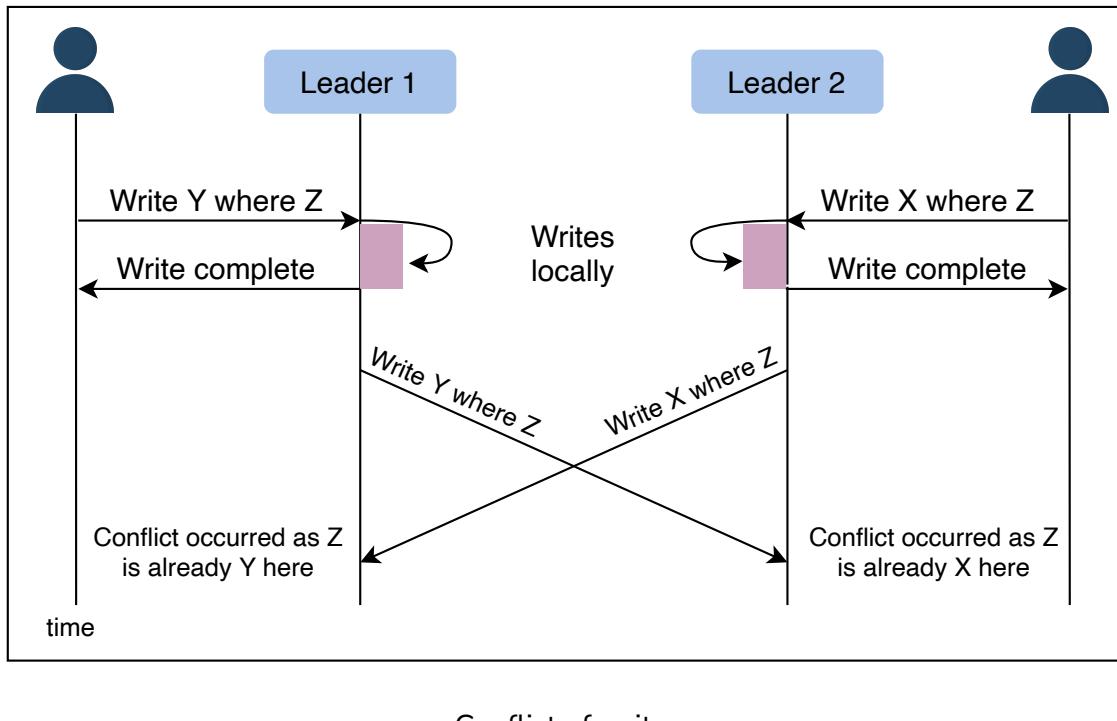
Multi-leader data replication model (all-to-all topology)

Conflict

Multi-leader replication gives better performance and scalability than single leader replication, but it also has a significant disadvantage. Since all the primary nodes concurrently deal with the write requests, they may modify the same data, which can create a conflict between them. For example, suppose the same data is edited by two clients simultaneously. In that case, their writes will be successful in their associated primary nodes, but when they reach the other primary nodes asynchronously, it creates a conflict.

Handle conflicts

Conflicts can result in different data at different nodes. These should be handled efficiently without losing any data. Let's discuss some of the approaches to handle conflicts:



Conflict of writes

Conflict avoidance

A simple strategy to deal with conflicts is to prevent them from happening in the first place. Conflicts can be avoided if the application can verify that all writes for a given record go via the same leader.

However, the conflict may still occur if a user moves to a different location and is now near a different data center. If that happens, we need to reroute the traffic. In such scenarios, the conflict avoidance approach fails and results in concurrent writes.

Last-write-wins

Using their local clock, all nodes assign a timestamp to each update. When a conflict occurs, the update with the latest timestamp is selected.

This approach can also create difficulty because the clock synchronization across nodes is challenging in distributed systems. There's clock skew that can result in data loss.

Custom logic

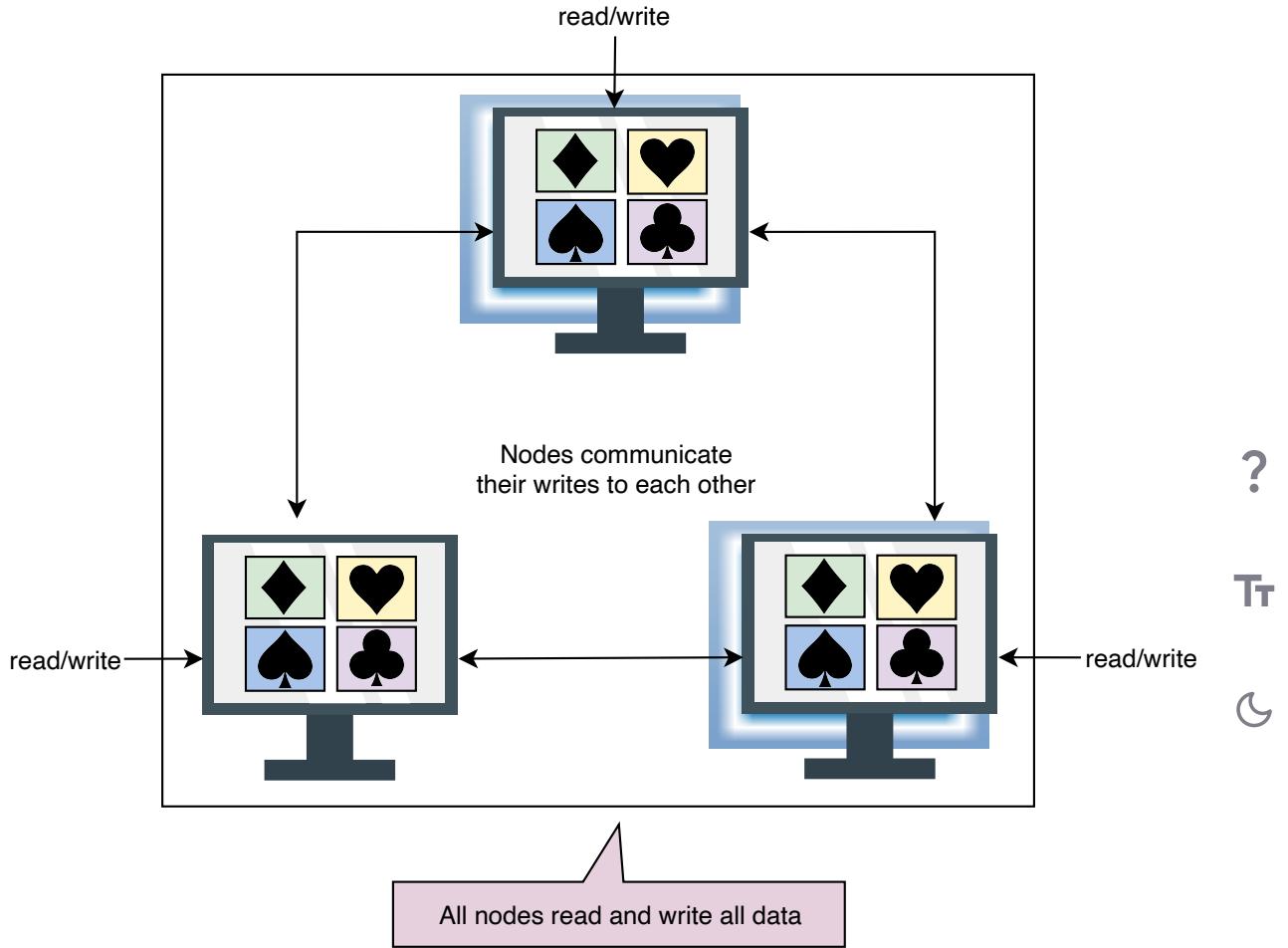
In this approach, we can write our own logic to handle conflicts according to the needs of our application. This custom logic can be executed on both reads and writes. When the system detects a conflict, it calls our custom conflict handler.

Multi-leader replication topologies

There are many topologies through which multi-leader replication is implemented, such as circular topology, star topology, and all-to-all topology. The most common is the all-to-all topology. In star and circular topology, there's again a similar drawback that if one of the nodes fails, it can affect the whole system. That's why all-to-all is the most used topology.

Peer-to-peer/leaderless replication

In primary-secondary replication, the primary node is a bottleneck and a single point of failure. Moreover, it helps to achieve read scalability but fails in providing write scalability. The **peer-to-peer replication** model resolves these problems by not having a single primary node. All the nodes have equal weightage and can accept reads and writes requests. Amazon popularized such a scheme in their DynamoDB data store.



Peer-to-peer data replication model where all nodes apply reads and writes to all the data

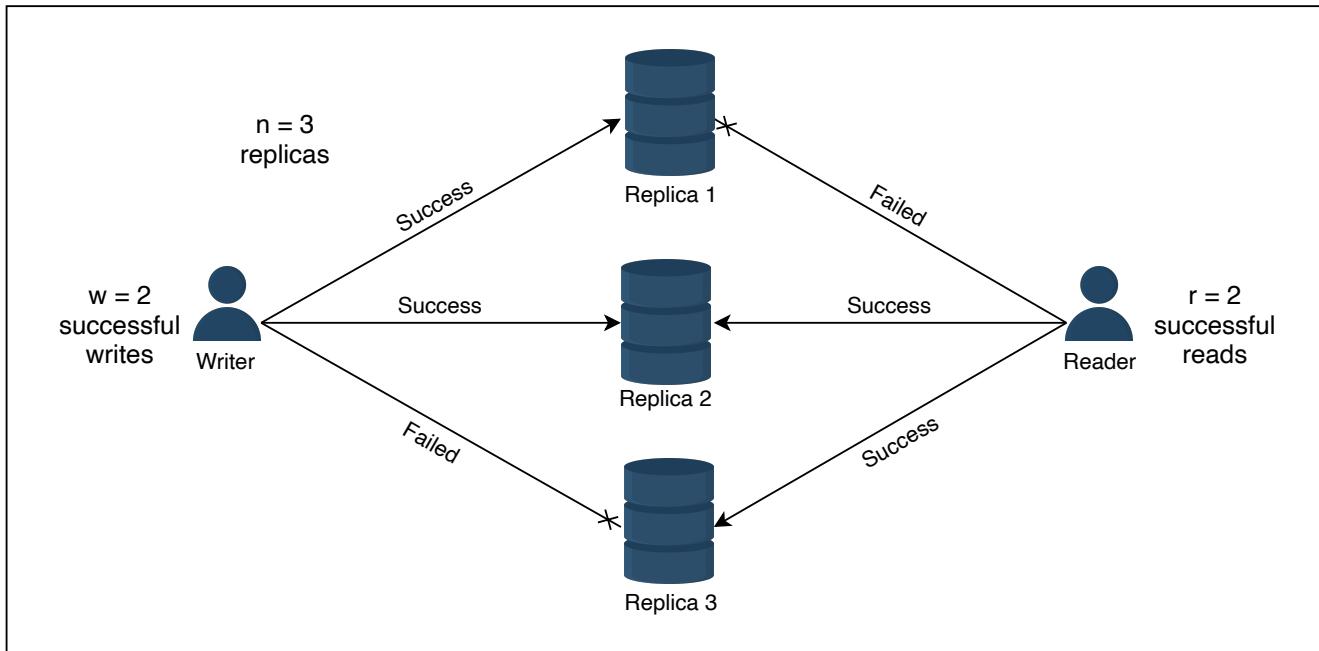
Like primary-secondary replication, this replication can also yield inconsistency. This is because when several nodes accept write requests, it may lead to concurrent writes. A helpful approach used for solving write-write inconsistency is called **quorums**.

Quorums

Let's suppose we have three nodes. If at least two out of three nodes are guaranteed to return successful updates, it means only one node has failed. This means that if we read from two nodes, at least one of them will have the updated version, and our system can continue working.

If we have n nodes, then every write must be updated in at least w nodes to be considered a success, and we must read from r nodes. We'll get an updated value from reading as long as $w + r > n$ because at least one of the nodes must have an updated write from which we can read. Quorum reads and writes adhere to these r and w values. These n , w , and r are configurable in Dynamo-style

databases.



Reader getting an updated value from replica 2

For more details on the topic of Quorum, refer to the following links:

- [What is Quorum?](#)
- [What is Quorum in distributed systems?](#)



Which replication mechanism is the most appropriate (high throughput, low latency to client, low implementation overhead) when our workload is read-heavy?

[Reset Quiz](#) ↕

[Submit Answer](#)

← Back

Types of Databases

[Mark As Completed](#)

[Next →](#)

Data Partitioning

Data Partitioning

Learn about data partitioning models along with their pros and cons.

We'll cover the following



- Why do we partition data?
- Sharding
 - Vertical sharding
 - Horizontal sharding
 - Key-range based sharding
 - Advantages
 - Disadvantages
 - Hash-based sharding
 - Advantages
 - Disadvantages
 - Consistent hashing
 - Advantages of consistent hashing
 - Disadvantages of consistent hashing
 - Rebalance the partitions
 - Avoid hash mod n
 - Fixed number of partitions
 - Dynamic partitioning
 - Partition proportionally to nodes
 - Partitioning and secondary indexes
 - Partition secondary indexes by document
 - Partition secondary indexes by the term
 - Request routing
 - ZooKeeper
 - Conclusion

Why do we partition data?

Data is an asset for any organization. Increasing data and concurrent read/write traffic to the data puts scalability pressure on traditional databases. As a result, the latency and throughput are affected. Traditional databases are attractive due to their properties such as range queries, secondary indices, and transactions with the ACID properties.

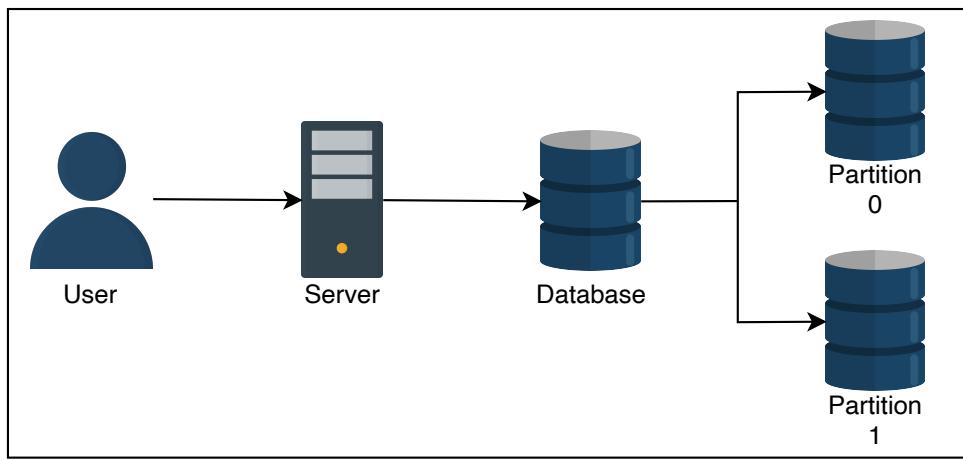
At some point, a single node-based database isn't enough to tackle the load. We might need to distribute the data over many nodes but still export all the nice properties of relational databases. In practice, it has proved challenging to provide single-node database-like properties over a distributed database.

One solution is to move data to a NoSQL-like system. However, the historical codebase and its close cohesion with traditional databases make it an expensive problem to tackle.

Organizations might scale traditional databases by using a third-party solution. But often, integrating a third-party solution has its complexities. More importantly, there are abundant opportunities to optimize for the specific problem at hand and get much better performance than a general-purpose solution.

Data partitioning (or sharding) enables us to use multiple nodes where each node manages some part of the whole data. To handle increasing query rates and data amounts, we strive for balanced partitions and balanced read/write load.

We'll discuss different ways to partition data, related challenges, and their solutions in this lesson.



A database with two partitions to distribute the data and associated read/write load

Sharding

To divide load among multiple nodes, we need to partition the data by a phenomenon known as **partitioning** or **sharding**. In this approach, we split a large dataset into smaller chunks of data stored at different nodes on our network.

The partitioning must be balanced so that each partition receives about the same amount of data. If partitioning is unbalanced, the majority of queries will fall into a few partitions. Partitions that are heavily loaded will create a system bottleneck. The efficacy of partitioning will be harmed because a significant portion of data retrieval queries will be sent to the nodes that carry the highly congested partitions. Such partitions are known as hotspots. Generally, we use the following two ways to shard the data:

- Vertical sharding
- Horizontal sharding

Vertical sharding

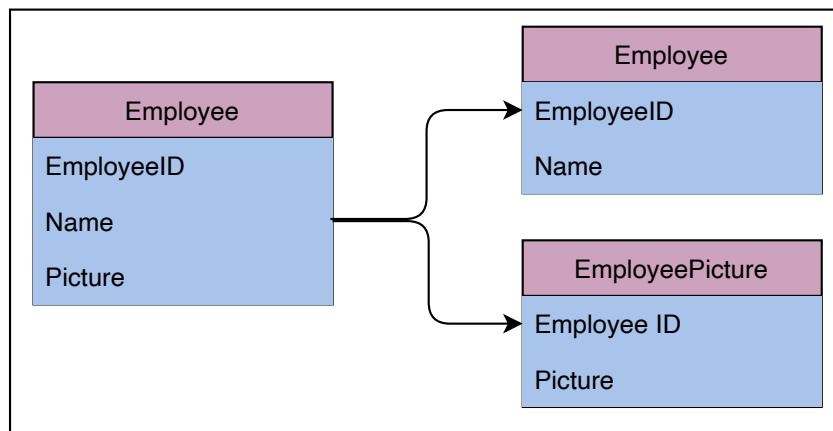
We can put different tables in various database instances, which might be running on a different physical server. We might break a table into multiple tables so that some columns are in one table while the rest are in the other. We should be careful if there are joins between multiple tables. We may like to keep such tables together on one shard.

Often, **vertical sharding** is used to increase the speed of data retrieval from a

table consisting of columns with very wide text or a binary large object (blob). In this case, the column with large text or a blob is split into a different table.

As shown in the figure a couple paragraphs below, the `Employee` table is divided into two tables: a reduced `Employee` table and an `EmployeePicture` table. The `EmployeePicture` table has just two columns, `EmployeeID` and `Picture`, separated from the original table. Moreover, the primary key `EmployeeID` of the `Employee` table is added in both partitioned tables. This makes the data read and write easier, and the reconstruction of the table is performed efficiently.

Vertical sharding has its intricacies and is more amenable to manual partitioning, where stakeholders carefully decide how to partition data. In comparison, horizontal sharding is suitable to automate even under dynamic conditions.



Vertical partitioning

Note: Creating shards by moving specific tables of a database around is also a form of vertical sharding. Usually, those tables are put in the same shard because they often appear together in queries, for example, for joins. We will see an example of such a use-case [ahead in the course](#).

Horizontal sharding

At times, some tables in the databases become too big and affect read/write latency. **Horizontal sharding** or partitioning is used to divide a table into multiple tables by splitting data row-wise, as shown in the figure in the next

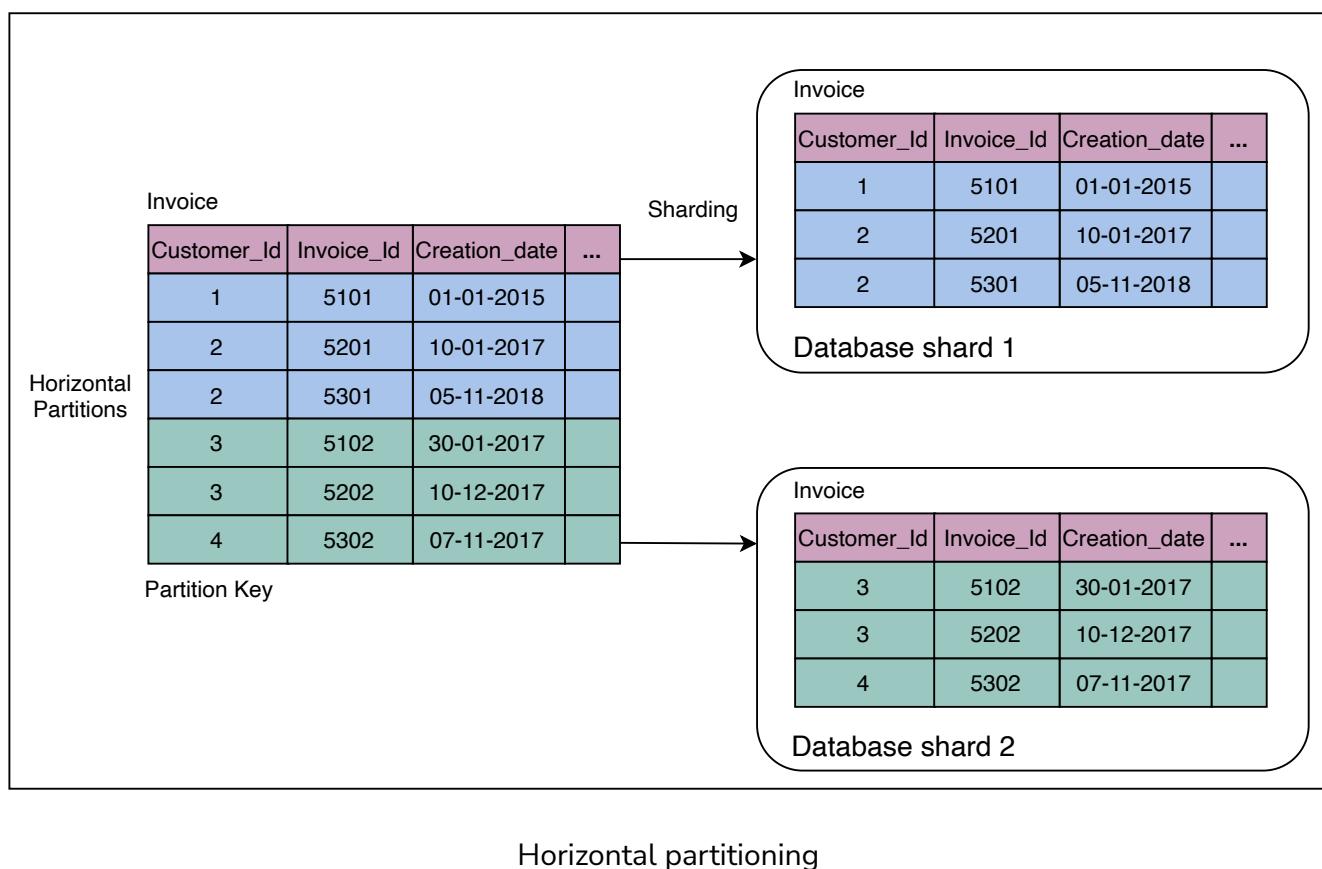
section. Each partition of the original table distributed over database servers is called a **shard**. Usually, there are two strategies available:

- Key-range based sharding
- Hash based sharding

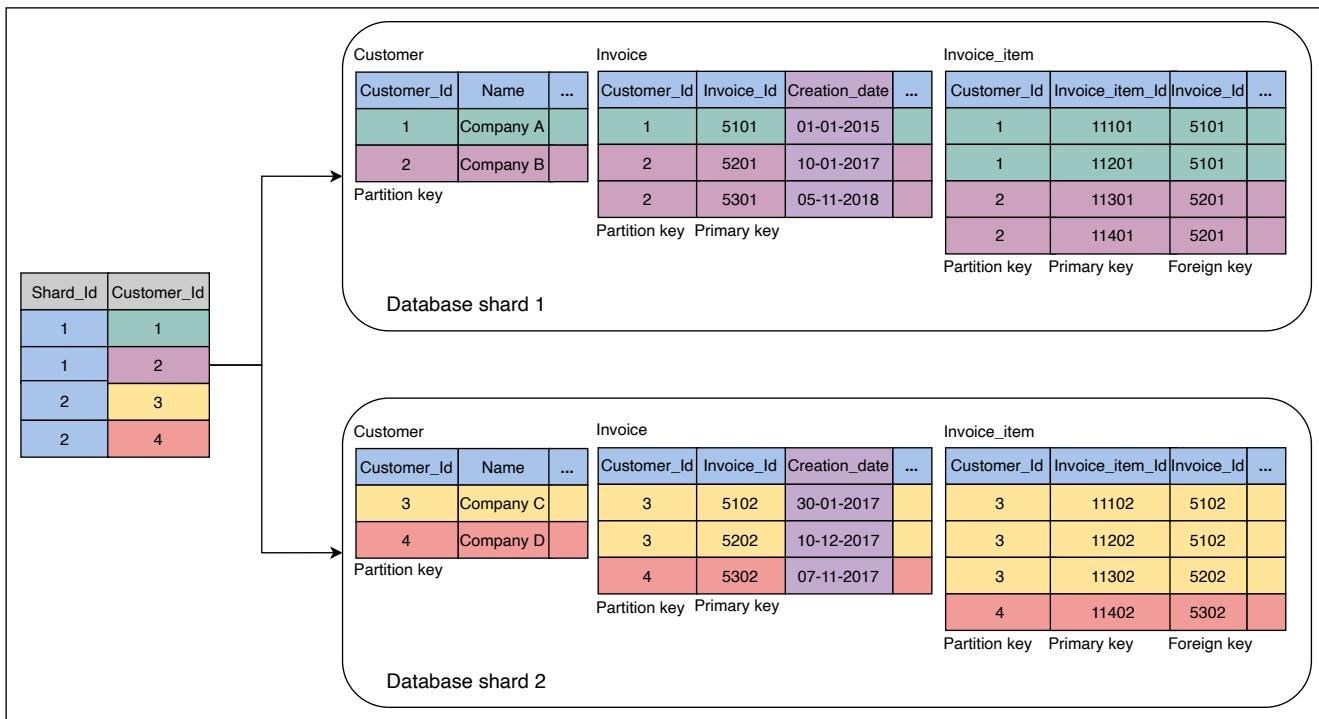
Key-range based sharding

In the **key-range based sharding**, each partition is assigned a continuous range of keys.

In the following figure, horizontal partitioning on the **Invoice** table is performed using the key-range based sharding with **Customer_Id** as the partition key. The two different colored tables represent the partitions.



Sometimes, a database consists of multiple tables bound by foreign key relationships. In such a case, the horizontal partition is performed using the same partition key on all tables in a relation. Tables (or subtables) that belong to the same partition key are distributed to one database shard. The following figure shows that several tables with the same partition key are placed in a single database shard:



Horizontal partitioning on a set of tables

The basic design techniques used in multi-table sharding are as follows:

- There's a partition key in the **Customer** mapping table. This table resides on each shard and stores the partition keys used in the shard. Applications create a mapping logic between the partition keys and database shards by reading this table from all shards to make the mapping efficient. Sometimes, applications use advanced algorithms to determine the location of a partition key belonging to a specific shard.
- The partition key column, **Customer_Id**, is replicated in all other tables as a data isolation point. It has a trade-off between an impact on increased storage and locating the desired shards efficiently. Apart from this, it's helpful for data and workload distribution to different database shards. The data routing logic uses the partition key at the application tier to map queries specified for a database shard.
- Primary keys are unique across all database shards to avoid key collision during data migration among shards and the merging of data in the online analytical processing (OLAP) environment.
- The column **Creation_date** serves as the data consistency point, with an assumption that the clocks of all nodes are synchronized. This column is used as a criterion for merging data from all database shards into the

global view when essential.

Advantages

- Using key-range-based sharding method, the range-query-based scheme is easy to implement. We precisely know where (which node, which shard) to look for a specific range of keys.
- Range queries can be performed using the partitioning keys, and those can be kept in partitions in sorted order. How exactly such a sorting happens over time as new data comes in is implementation specific.

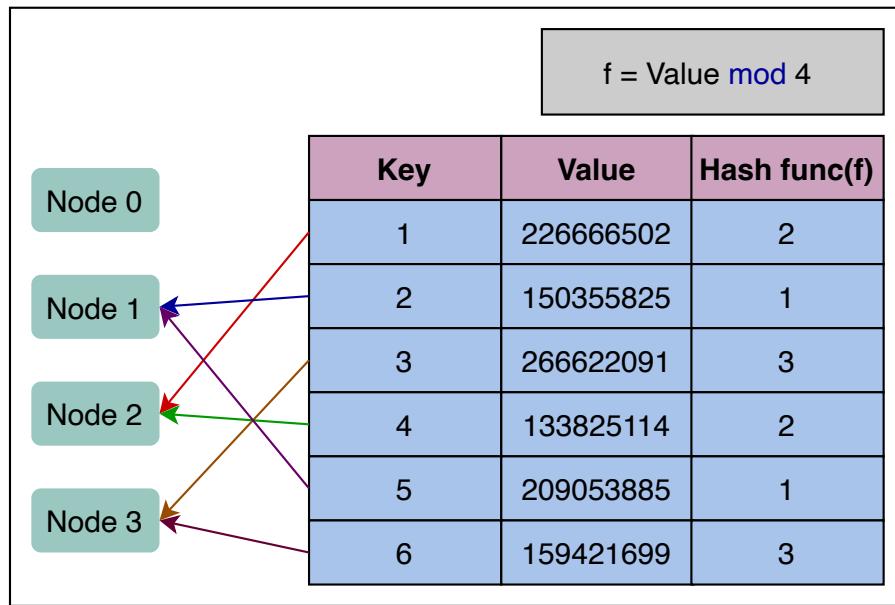
Disadvantages

- Range queries can't be performed using keys other than the partitioning key.
- If keys aren't selected properly, some nodes may have to store more data due to an uneven distribution of the traffic.

Hash-based sharding

Hash-based sharding uses a hash-like function on an attribute, and it produces different values based on which attribute the partitioning is performed. The main concept is to use a hash function on the key to get a hash value and then mod by the number of partitions. Once we've found an appropriate hash function for keys, we may give each partition a range of hashes (rather than a range of keys). Any key whose hash occurs inside that range will be kept in that partition.

In the illustration below, we use a hash function of $Value \bmod n = n$. The n is the number of nodes, which is four. We allocate keys to nodes by checking the mod for each key. Keys with a mod value of 2 are allocated to node 2. Keys with a mod value of 1 are allocated to node 1. Keys with a mod value of 3 are allocated to node 3. Because there's no key with a mod value of 0, node 0 is left vacant.



Hash-based sharding

Advantages

- Keys are uniformly distributed across the nodes.

Disadvantages

- We can't perform range queries with this technique. Keys will be spread over all partitions.

Note: How many shards per database?

Empirically, we can determine how much each node can serve with acceptable performance. It can help us find out the maximum amount of data that we would like to keep on any one node. For example, if we find out that we can put a maximum of 50 GB of data on one node, we have the following:

Database size = 10 TB

Size of a single shard = 50 GB

Number of shards the database should be distributed in = $10 \text{ TB}/50 \text{ GB}$
= 200 shards

Consistent hashing

Consistent hashing assigns each server or item in a distributed hash table a place on an abstract circle, called a ring, irrespective of the number of servers in the table. This permits servers and objects to scale without compromising the system's overall performance.

Advantages of consistent hashing

- It's easy to scale horizontally.
- It increases the throughput and improves the latency of the application.

Disadvantages of consistent hashing

- Randomly assigning nodes in the ring may cause non-uniform distribution.

Rebalance the partitions

Query load can be imbalanced across the nodes due to many reasons, including the following:

- The distribution of the data isn't equal.
- There's too much load on a single partition.
- There's an increase in the query traffic, and we need to add more nodes to keep up.

We can apply the following strategies to rebalance partitions.

Avoid hash mod n

Usually, we avoid the hash of a key for partitioning (we used such a scheme to explain the concept of hashing in simple terms earlier). The problem with the addition or removal of nodes in the case of *hashmodn* is that every node's partition number changes and a lot of data moves. For example, assume we have $\text{hash}(\text{key}) = 1235$. If we have five nodes at the start, the key will start on node 1 ($1235 \bmod 5 = 0$). Now, if a new node is added, the key would have to be moved to node 6 ($1235 \bmod 6 = 5$), and so on. This moving of keys from one node to another makes rebalancing costly.

Fixed number of partitions

In this approach, the number of partitions to be created is fixed at the time when

we set our database up. We create a higher number of partitions than the nodes and assign these partitions to nodes. So, when a new node is added to the system, it can take a few partitions from the existing nodes until the partitions are equally divided.

There's a downside to this approach. The size of each partition grows with the total amount of data in the cluster since all the partitions contain a small part of the total data. If a partition is very small, it will result in too much overhead because we may have to make a large number of small-sized partitions, each costing us some overhead. If the partition is very large, rebalancing the nodes and recovering from node failures will be expensive. It's very important to choose the right number of partitions. A fixed number of partitions is used in Elasticsearch, Riak, and many more.

Dynamic partitioning

In this approach, when the size of a partition reaches the threshold, it's split equally into two partitions. One of the two split partitions is assigned to one node and the other one to another node. In this way, the load is divided equally. The number of partitions adapts to the overall data amount, which is an advantage of dynamic partitioning.

However, there's a downside to this approach. It's difficult to apply dynamic rebalancing while serving the reads and writes. Dynamic rebalancing during reads and writes is challenging because it involves moving data between nodes, causing latency and potential conflicts. Ensuring data consistency (as data is simultaneously moved and accessed) and availability (potentially requiring pauses in reads/writes during rebalancing) introduces complexities that can impact system performance and reliability. This approach is used in HBase and MongoDB.

Partition proportionally to nodes

In this approach, the number of partitions is proportionate to the number of nodes, which means every node has fixed partitions. In earlier approaches, the number of partitions was dependent on the size of the dataset. That isn't the case here. While the number of nodes remains constant, the size of each partition

rises according to the dataset size. However, as the number of nodes increases, the partitions shrink. When a new node enters the network, it splits a certain number of current partitions at random, then takes one half of the split and leaves the other half alone. This can result in an unfair split. This approach is used by Cassandra and Ketama.

Point to Ponder

Question

Who performs the rebalancing? Is it automatic or manual?

[Hide Answer](#) ^

There are two ways to perform rebalancing: automatic and manual. In **automatic rebalancing**, there's no administrator. The system determines when to perform the partitions and when to move data from one node to another.

In **manual rebalancing**, the administrator determines when and how to perform the partitioning. Organizations perform rebalancing according to their needs. Some use automatic rebalancing, and some use manual.

Partitioning and secondary indexes

We've discussed key-value data model partitioning schemes in which the records are retrieved with primary keys. But what if we have to access the records through secondary indexes? Secondary indexes are the records that aren't identified by primary keys but are just a way of searching for some value. For example, the above [illustration of horizontal partitioning](#) contains the customer

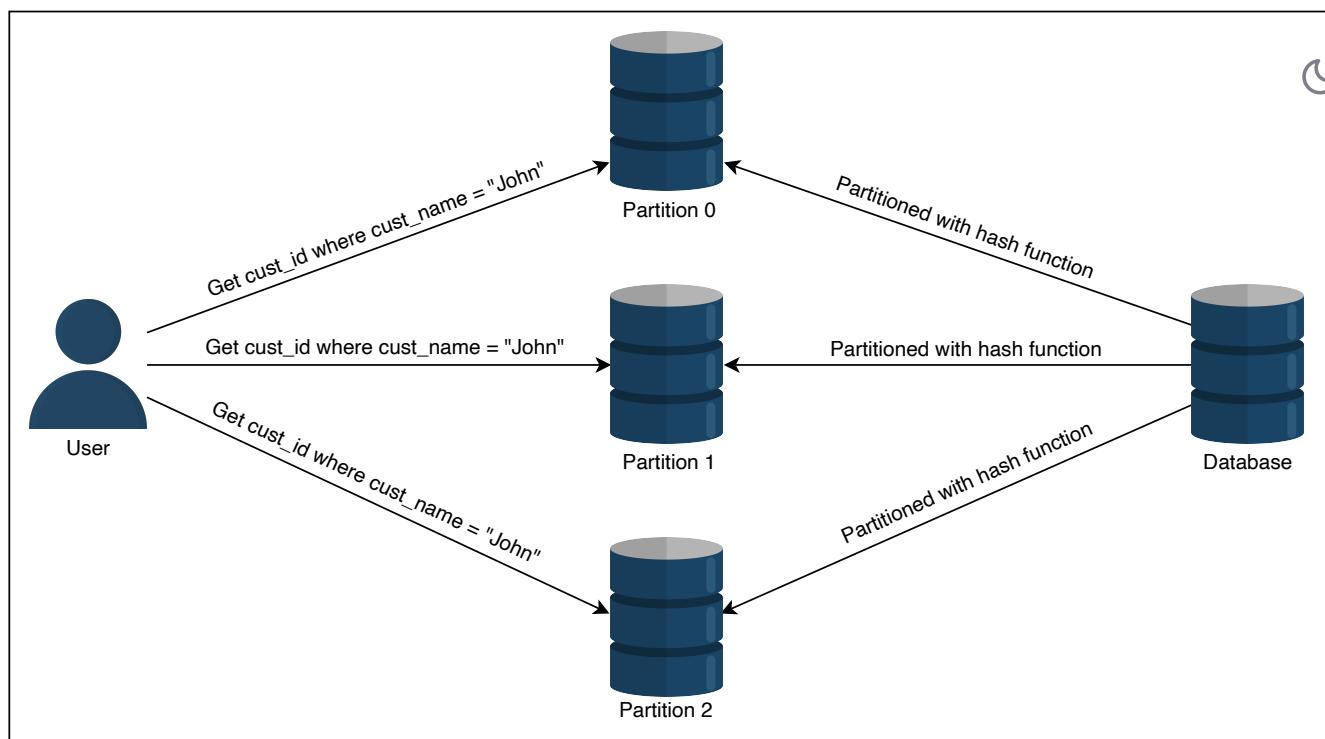
table, searching for all customers with the same creation year.

We can partition with secondary indexes in the following ways.

Partition secondary indexes by document

Each partition is fully independent in this indexing approach. Each partition has its secondary indexes covering just the documents in that partition. It's unconcerned with the data held in other partitions. If we want to write anything to our database, we need to handle that partition only containing the document ID we're writing. It's also known as the local index. In the illustration below, there are three partitions, each having its own identity and data. If we want to get all the customer IDs with the name **John**, we have to request from all partitions.

However, this type of querying on secondary indexes can be expensive. As a result of being restricted by the latency of a poor-performing partition, read query latencies may increase.



Partitioning secondary indexes by document

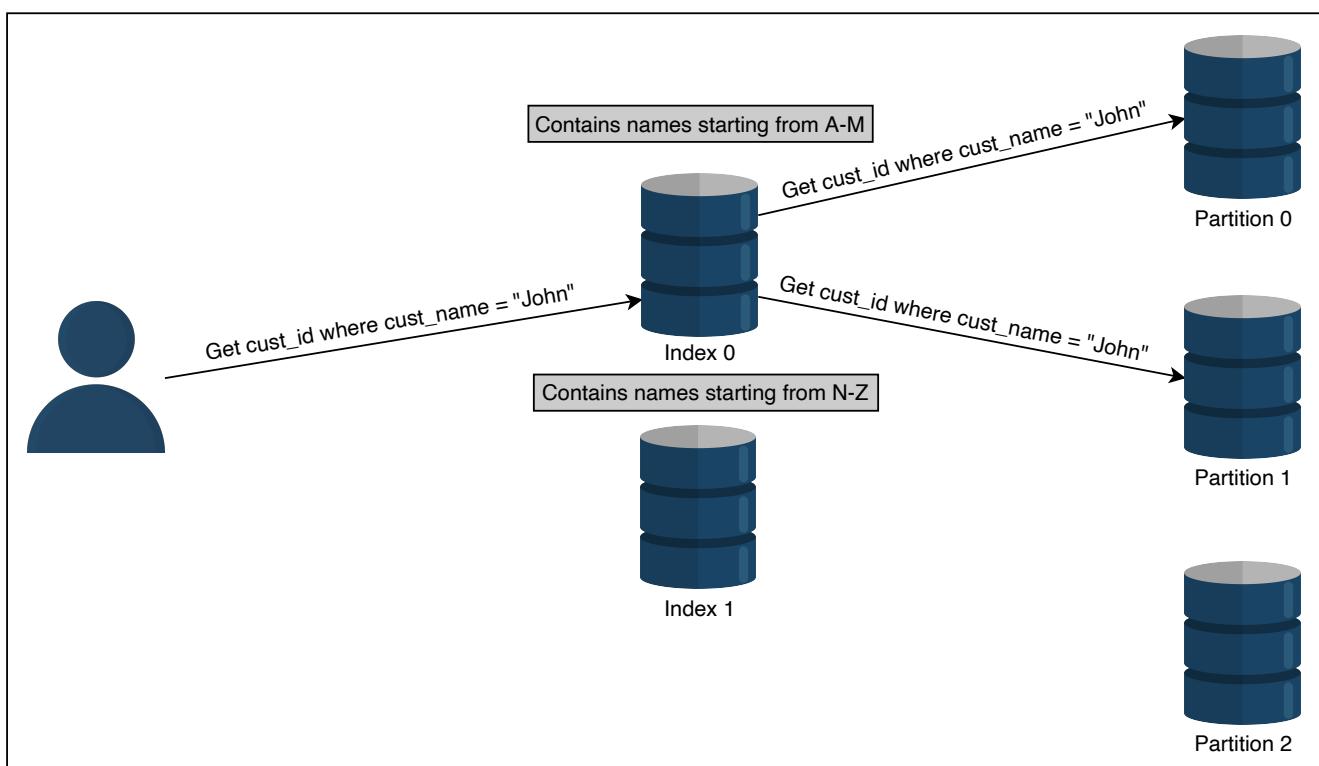
Partition secondary indexes by the term

Instead of creating a secondary index for each partition (a local index), we can

make a global index for secondary terms that encompasses data from all partitions.

In the illustration below, we create indexes on names (the term on which we're partitioning) and store all the indexes for names on separated nodes. To get the `cust_id` of all the customers named **John**, we must determine where our term index is located. The `index 0` contains all the customers with names starting with "A" to "M." The `index 1` includes all the customers with names beginning with "N" to "Z." Because **John** lies in `index 0`, we fetch a list of `cust_id` with the name **John** from `index 0`.

Partitioning secondary indexes by the term is more read-efficient than partitioning secondary indexes by the document. This is because it only accesses the partition that contains the term. However, a single write in this approach affects multiple partitions, making the method write-intensive and complex.



Partitioning secondary indexes by term

Request routing

We've learned how to partition our data. However, one question arises here: How does a client know which node to connect to while making a request? The allocation of partitions to nodes varies after rebalancing. If we want to read a

specific key, how do we know which IP address we need to connect to read?

This problem is also known as **service discovery**. Following are a few approaches to this problem:

- Allow the clients to request any node in the network. If that node doesn't contain the requested data, it forwards that request to the node that does contain the related data.
- The second approach contains a routing tier. All the requests are first forwarded to the routing tier, and it determines which node to connect to fulfill the request.
- The clients already have the information related to partitioning and which partition is connected to which node. So, they can directly contact the node that contains the data they need.

In all of these approaches, the main challenge is to determine how these components know about updates in the partitioning of the nodes.

ZooKeeper

To track changes in the cluster, many distributed data systems need a separate management server like ZooKeeper. **Zookeeper** keeps track of all the mappings in the network, and each node connects to ZooKeeper for the information. Whenever there's a change in the partitioning, or a node is added or removed, ZooKeeper gets updated and notifies the routing tier about the change. HBase, Kafka and SolrCloud use ZooKeeper.

Conclusion

For all current distributed systems, partitioning has become the standard protocol. Because systems contain increasing amounts of data, partitioning the data makes sense since it speeds up both writes and reads. It increases the system's availability, scalability, and performance.

 Back

Mark As Completed

Next 

Data Replication

Trade-offs in Databases

Trade-offs in Databases

Learn when to use horizontal sharding instead of vertical sharding and vice versa.

We'll cover the following



- Which is the best database sharding approach?
 - Advantages and disadvantages of a centralized database
 - Advantages
 - Disadvantages
 - Advantages and disadvantages of a distributed database
 - Advantages
 - Disadvantages
 - Query optimization and processing speed in a distributed database
 - Parameters assumption
 - Possible approaches
 - Conclusion

Which is the best database sharding approach?

Both horizontal and vertical sharding involve adding resources to our computing infrastructure. Our business stakeholders must decide which is suitable for our organization. We must scale our resources accordingly for our organization and business to grow, to prevent downtime, and to reduce latency. We can scale these resources through a combination of adjustments to CPU, physical memory requirements, hard disk adjustments, and network bandwidth.

The following sections explain the pros and cons of no sharding versus sharding.

Advantages and disadvantages of a centralized database

Advantages

- Data maintenance, such as updating and taking backups of a centralized database, is easy.
- Centralized databases provide stronger consistency and ACID transactions than distributed databases.
- Centralized databases provide a much simpler programming model for the end programmers as compared to distributed databases.
- It's more efficient for businesses to have a small amount of data to store that can reside on a single node.

Disadvantages

- A centralized database can slow down, causing high latency for end users, when the number of queries per second accessing the centralized database is approaching single-node limits.
- A centralized database has a single point of failure. Because of this, its probability of not being accessible is much higher.

Advantages and disadvantages of a distributed database

Advantages

- It's fast and easy to access data in a distributed database because data is retrieved from the nearest database shard or the one frequently used.
- Data with different **levels of distribution transparency** ↴ can be stored in separate places.
- Intensive transactions consisting of queries can be divided into multiple optimized subqueries, which can be processed in a parallel fashion.

Disadvantages

- Sometimes, data is required from multiple sites, which takes more time than expected.
- Relations are partitioned vertically or horizontally among different nodes. Therefore, operations such as joins need to reconstruct complete relations by carefully fetching data. These operations can become much more

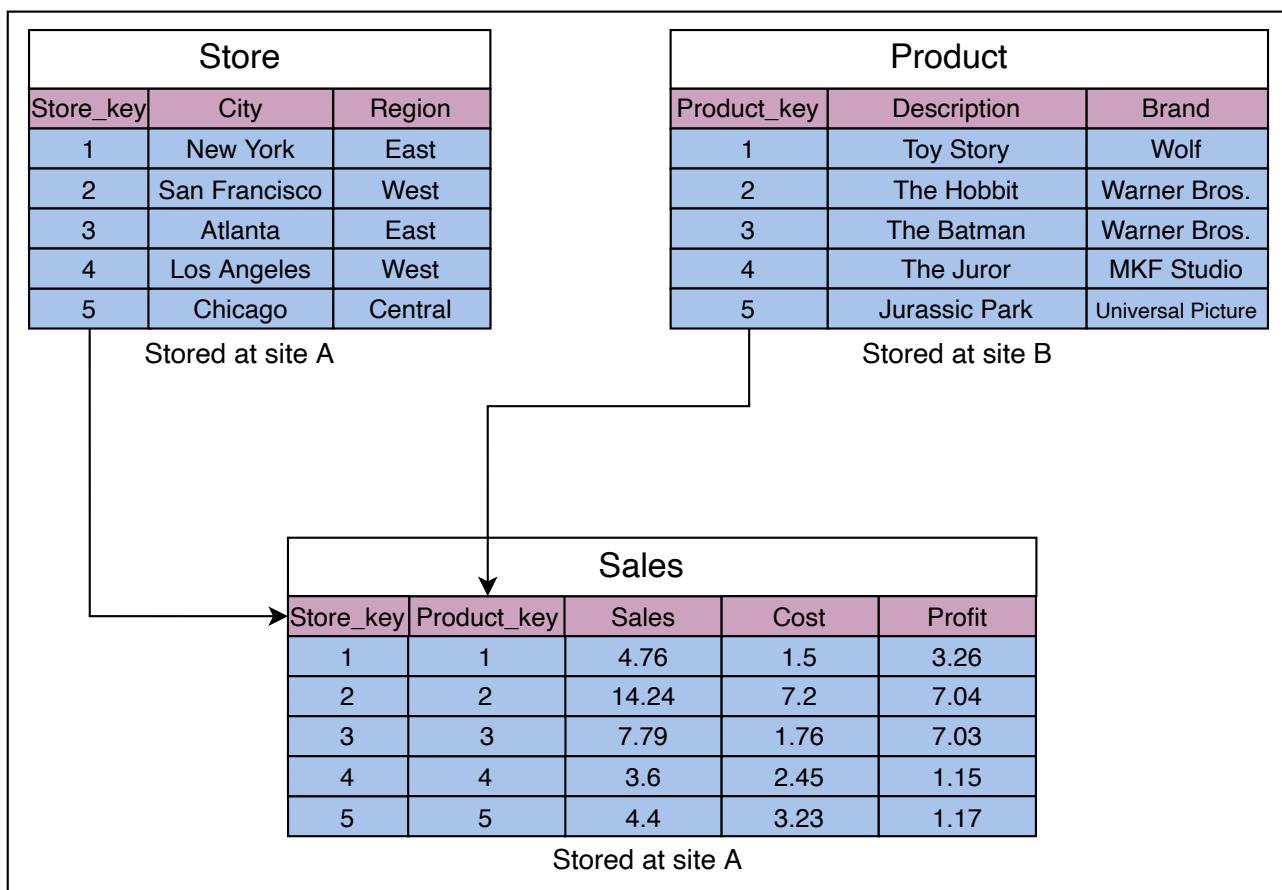
expensive and complex.

- It's difficult to maintain consistency of data across sites in the distributed database, and it requires extra measures.
- Updations and backups in distributed databases take time to synchronize data.

Query optimization and processing speed in a distributed database

A transaction in the distributed database depends on the type of query, number of sites (shards) involved, communication speed, and other factors, such as underlying hardware and the type of database used. However, as an example, let's assume a query accessing three tables, **Store**, **Product**, and **Sales**, residing on different sites.

The number of attributes in each table is given in the following figure:



Database schema consisting of three tables: **Store**, **Product**, and **Sales**

Let's assume the distribution of both tables on different sites is the following:

- The **Store** table has 10,000 tuples stored at site A.
- The **Product** table has 100,000 tuples stored at site B.
- The **Sales** table has one million tuples stored at site A.

Now, assume that we need to process the following query:

```
Select Store_key from (Store JOIN Sales JOIN Product)
where Region= 'East' AND Brand='Wolf';
```

The above query performs the join operations on the **Store**, **Sales**, and **Product** tables and retrieves the **Store_key** values from the table generated in the result of join operations.

Next, assume every stored tuple is 200 bits long. That's equal to 25 Bytes.

Furthermore, estimated cardinalities of certain intermediate results are as follows:

- The number of the **Wolf** brand is 10.
- The number of **East** region stores is 3000 (since there are 10,000 rows in the store table, and 3000 have region as east).

Communication assumptions are the following:

- Data rate = 50M bits per second
- Access delay = 0.1 second

Parameters assumption

Before processing the query using different approaches, let's define some parameters:

a = Total access delay

b = Data rate

v = Total data volume

Now, let's compute the total communication time, T , according to the following formula:

$$T = a + \frac{v}{b}$$

Let's try the following possible approaches to execute the query.

Possible approaches

- Move the **Product** table to site A and process the query at A.

$$T = 0.1 + \frac{100,000 \times 200}{50,000,000} = 0.5 \text{ seconds}$$

Here, 0.1 is the access delay of the table on site A, and 100,000 is the number of tuples in the **Product** table. The size of each tuple in bits is 200, and 50,000,000 is the data rate. The 200 and 50,000,000 figures are the same for all of the following calculations.

- Move **Store** and **Sales** to site B and process the query at B:

$$T = 0.2 + \frac{(10,000+1,000,000) \times 200}{50,000,000} = 4.24 \text{ seconds}$$

Here, 0.2 is the access delay of the **Store** and **Sales** tables. The numbers  10,000 and 1,000,000 are the number of tuples in the **Store** and **Sales** tables, respectively. 

- Restrict **Brand** at site B to **Wolf** (called selection) and move the result to site A:

$$T = 0.1 + \frac{10 \times 200}{50,000,000} \approx 0.1 \text{ seconds}$$

Here, 0.1 is the access delay of the **Product** table. The number of the **Wolf** brand is 10, hence the number of tuples.

When we compare the three approaches, the third approach provides us the least latency (0.1 seconds). We didn't calculate filtering at site A because the number of rows will be much larger, and hence data volume will be more than the third case (filtering at the site B and then fetching data). This example shows that careful query optimization is also critical in the distributed database.

Conclusion

Data distribution (vertical and horizontal sharding) across multiple nodes aims to improve the following features, considering that the queries are optimized:

- Reliability (fault-tolerance)

- Performance
- Balanced storage capacity and dollar costs

Both centralized and distributed databases have their pros and cons. We should choose them according to the needs of our application.

[**← Back**](#)

[**Mark As Completed**](#)

[**Next →**](#)

Data Partitioning

System Design: The Key-value Store
