Introduction to Databases

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?

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.

There are two basic types of databases:

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

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.

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).

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

Relational databases provide the atomicity, consistency, isolation, and durability (ACID) properties to maintain the integrity of the database.

Let's discuss ACID in detail:

Atomicity: A transaction is considered an atomic unit. Therefore, either all the statements within a transaction will successfully execute, or none of them will execute. If a statement fails within a transaction, it should be aborted and rolled back.

Consistency: At any given time, the database should be in a consistent state, and it should remain in a consistent state after every transaction. For example, if multiple users want to view a record from the database, it should return a similar result each time.

Isolation: In the case of multiple transactions running concurrently, they shouldn't be affected by each other. The final state of the database should be the same as the transactions were executed sequentially.

Durability: The system should guarantee that completed transactions will survive permanently in the database even in system failure events.

Why Relational Databases?

Flexibility → DDL provides flexibility

Reduced Redundancy : Relational database eliminates data redundancy.

Concurrency.

Integration

Backup & disaster Recovery

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.

Non Relational Databases

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.

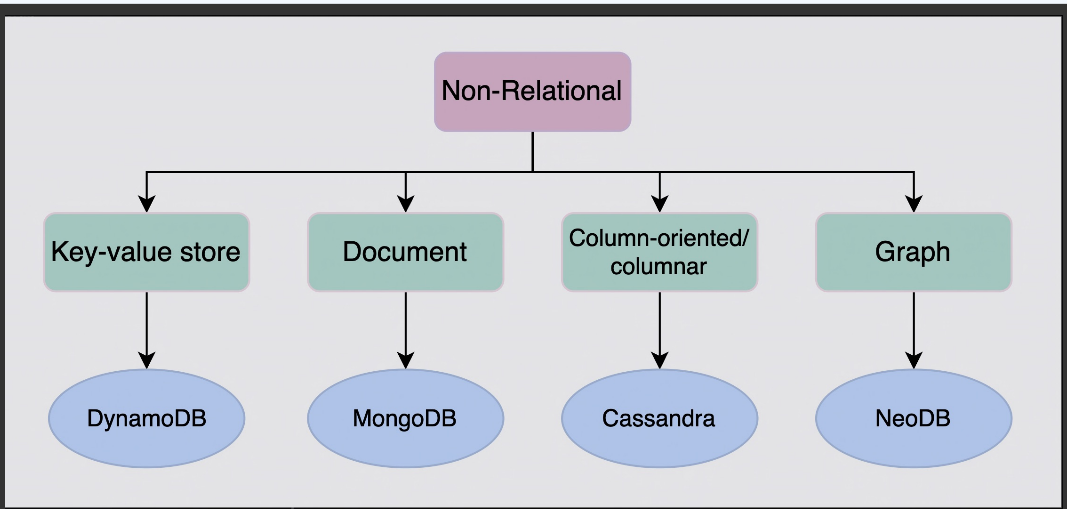
Simple design: Unlike relational databases, NoSQL doesn't require dealing with the impedance mismatch—for example, storing all the employees' data in one document instead of multiple tables that require join operations. This strategy makes it simple and easier to write less code, debug, and maintain.

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.



Data Replication

```
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).
```

Replication

Replications refers to keeping multiple copies of the data at various nodes. (preferably geographically) to achieve availability, scalability & performance.

Addional complexities could arise

→ How do we keep multiple copies of data consistent with each other?

→ How do we deal with failed replica node?

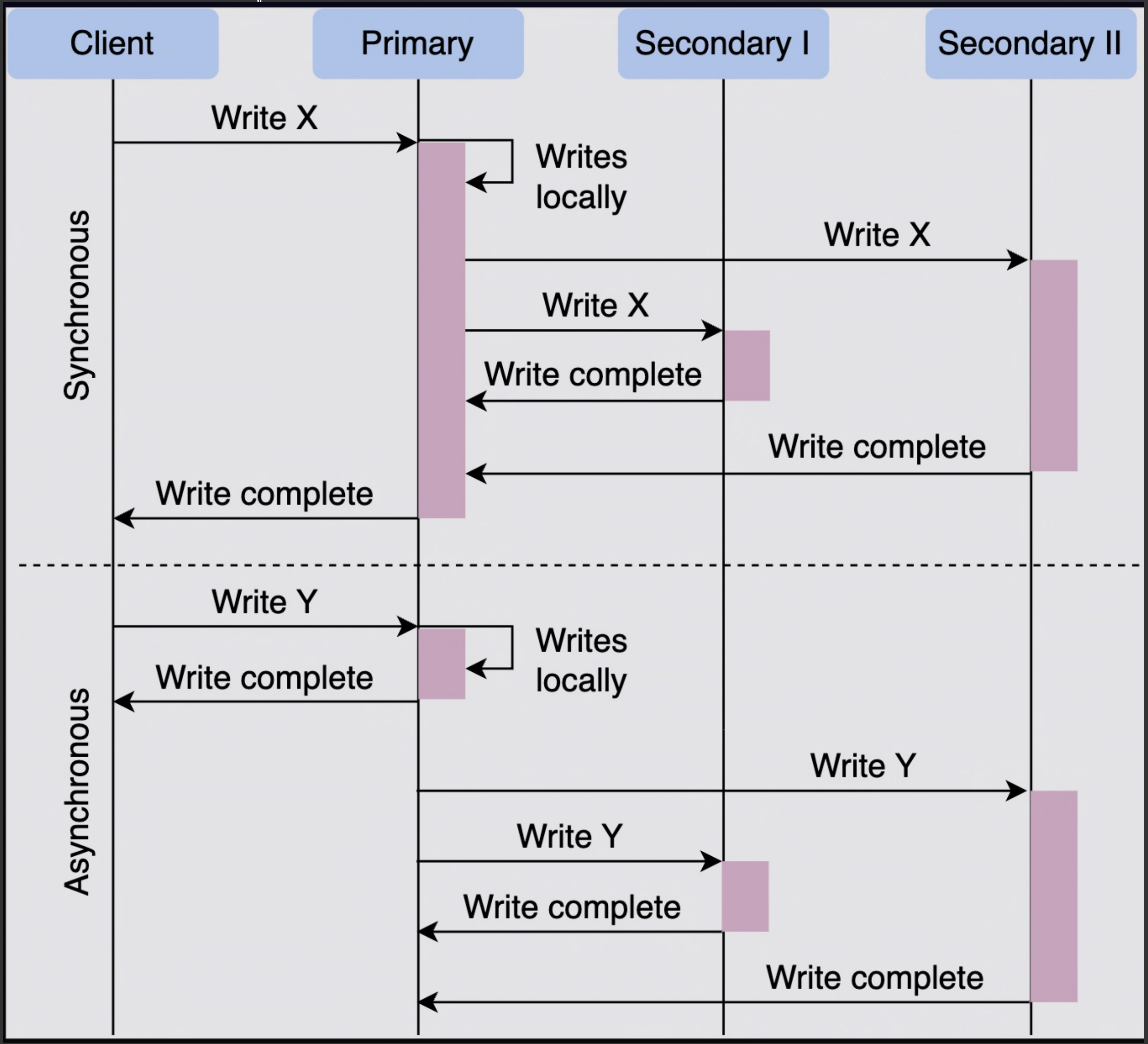
→ Should we replicate synchronous or asynchronously?

→ How do we handle concurrent writes?

Synchronous Replication

In synchronous replication, the primary node waits for acknowledgement 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 dose'nt wait for the acknowledgment from the secondary nodes & reports success to the client after updating itself.

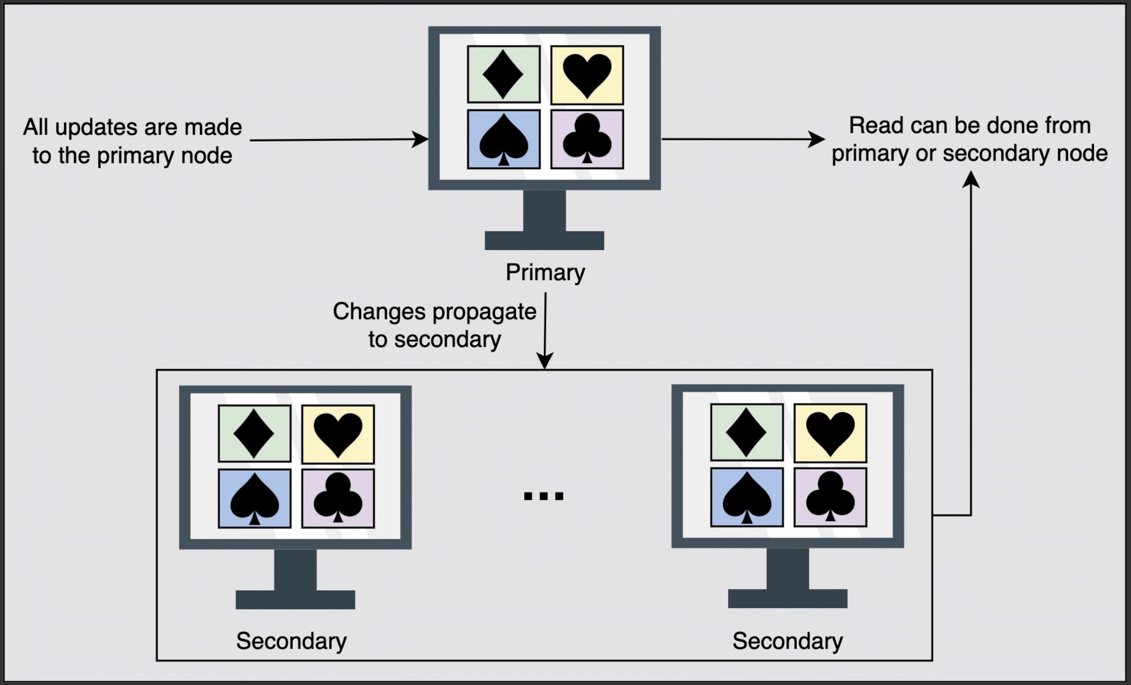


Data Replication Models

```
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.
```


All updates are made to the primary node
Read can be done from primary or secondary node
Primary
Changes propagate to secondary
...
Secondary
Secondary

What happens when the primary node fails?

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.

There are many replication methods.

1.) statement based Replication

2) Write-ahead log (WAL) shipping

3) Logical (row-based) log replication.

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.

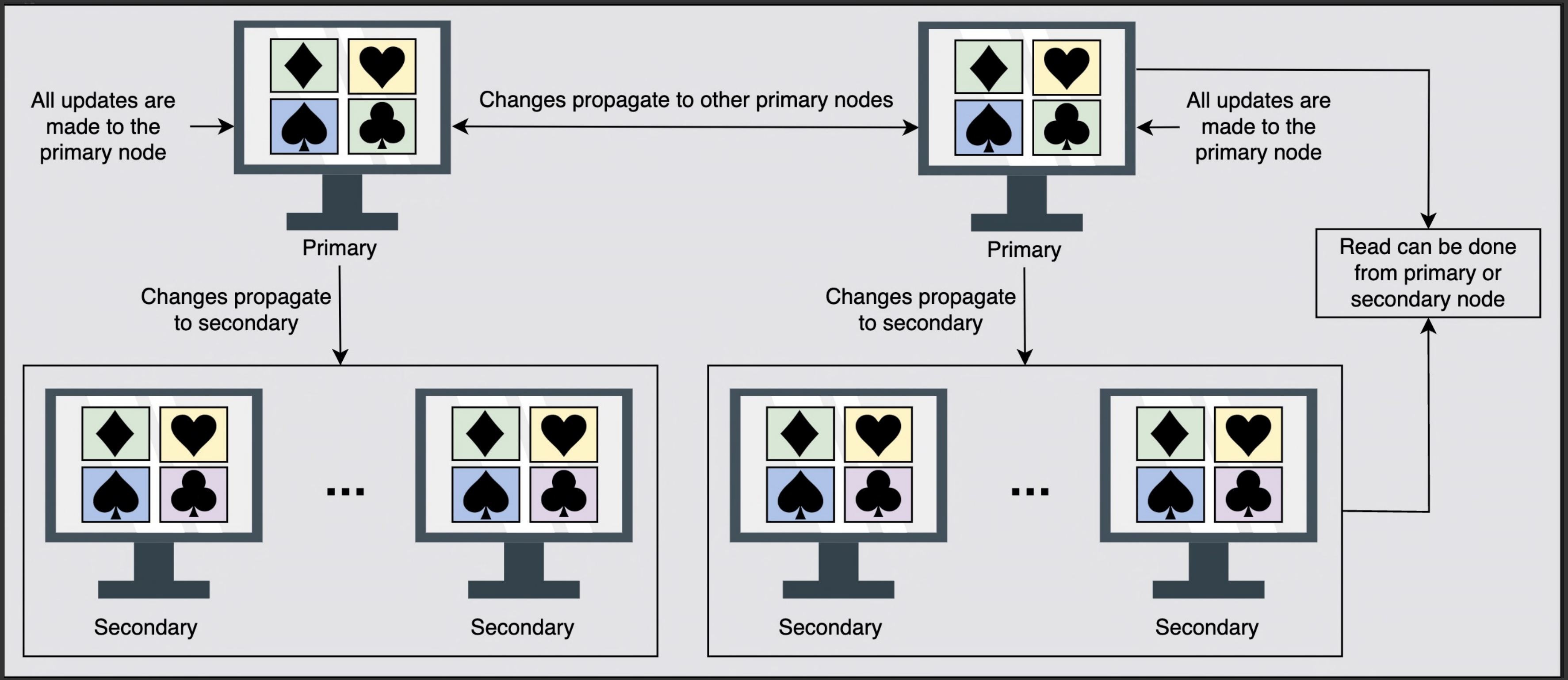
This type of approach seems good, but it has its disadvantages. For example, any nondeterministic function (such as NOW()) might result in distinct writes on the follower and leader. Furthermore, if a write statement is dependent on a prior write, and both of them reach the follower in the wrong order,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

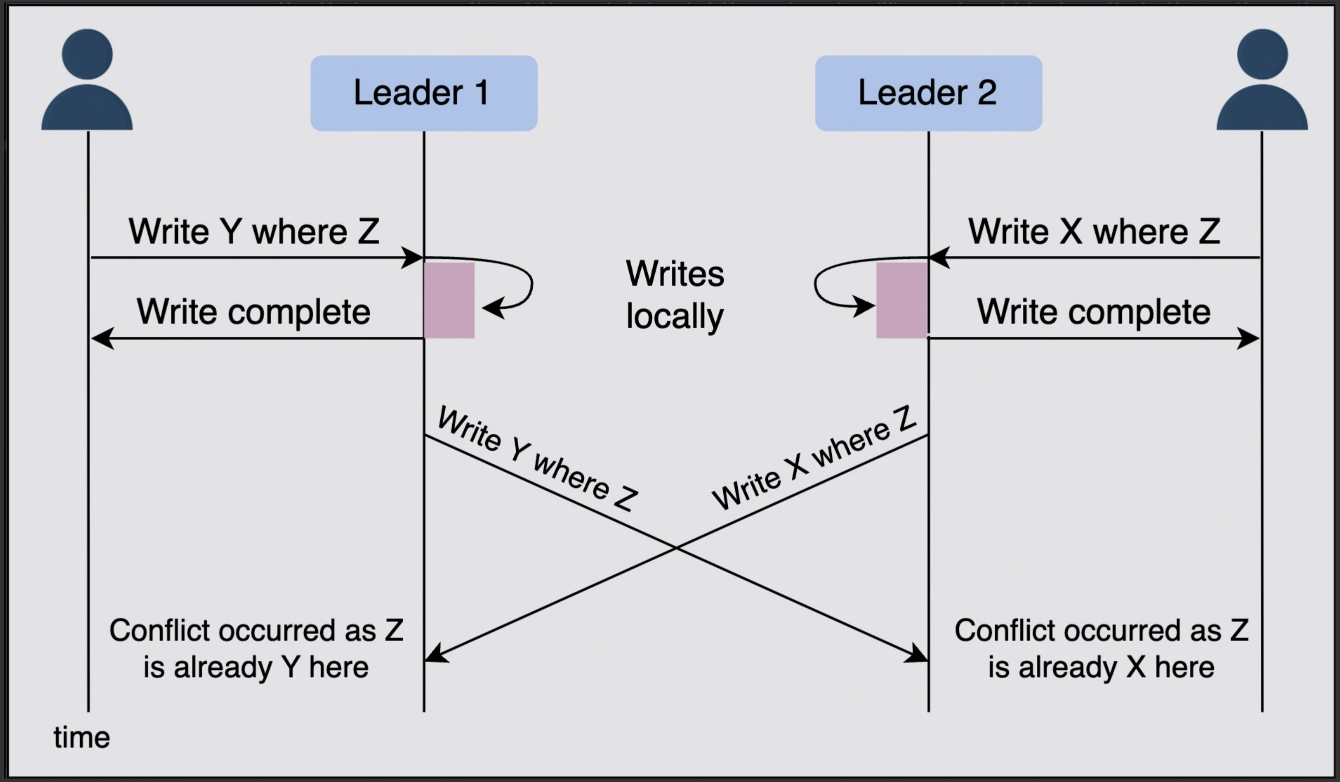
Multi-leader Replication

```
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.
```


All updates are made to the primary node
Changes propagate to other primary nodes
All updates are made to the primary node
Primary
Primary
Read can be done from primary or secondary node
Changes propagate to secondary
Changes propagate to secondary
...
...
Secondary
Secondary
Secondary
Secondary

Handle Conflicts

multileader replication gives better performance & scalability than single leader replication, but it also has a significant disadvantage.



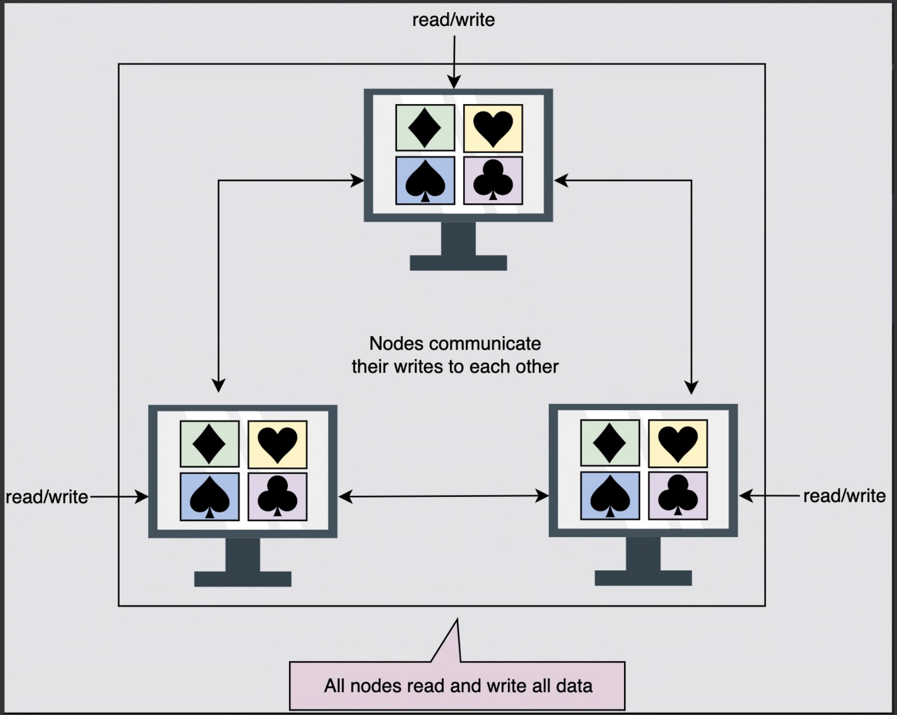
Last-write-wins
Using their local clock, all nodes assign a timestamp to each update. When a conflict occurs, the update with the latest timestamp is selected.

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.

Peer-to-peer/leaderless replication

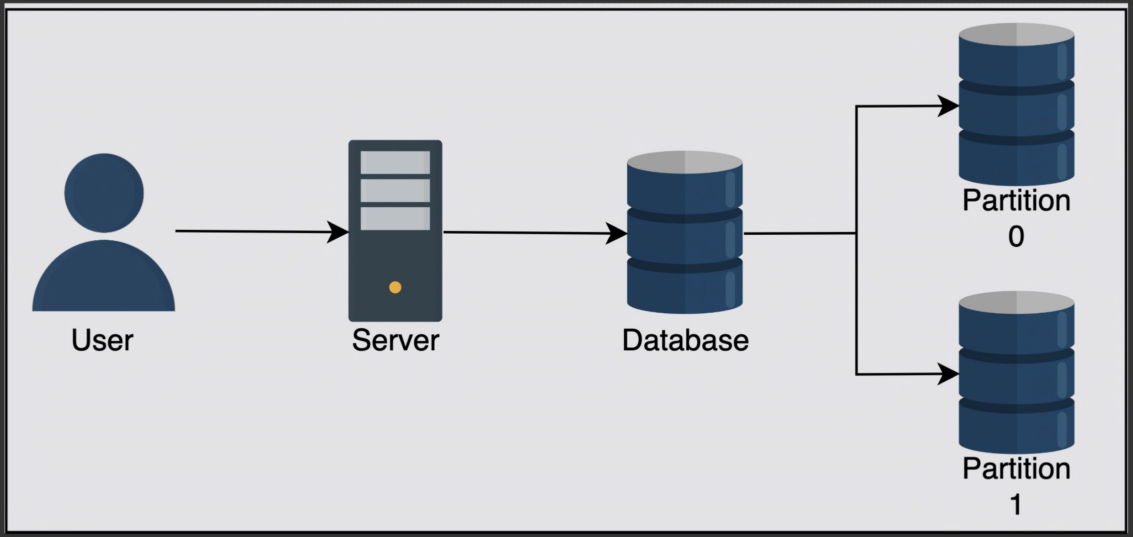
In primary-secondary replication, the primary node is a bottleneck and a single point of failure. Moreover, it helps to achieve read scalability but fails to provide write scalability. The peer-to-peer replication model resolves these problems by not having a single primary node. All the nodes have equal weightage and can accept read and write requests. This replication scheme can be found in the Cassandra database.


read/write
Nodes communicate their writes to each other
read/write
read/write
All nodes read and write all data

Why do we partition Data?

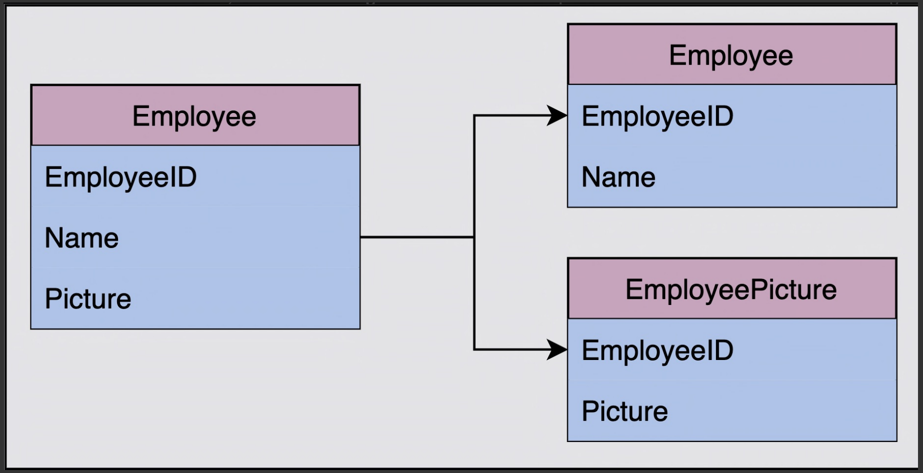
→ Single Node Not efficient to tackle the load.

→ Data partitioning (or sharding) enable us to use multiple nodes where each node manages some part of the whole data.



$$\begin{tikzcd}[row sep=small] & \text{Vertical Sharding} \\ \underline{\text{Sharding}} \arrow[ru] \arrow[rd] & \\ & \text{Horizontal Sharding} \end{tikzcd}$$

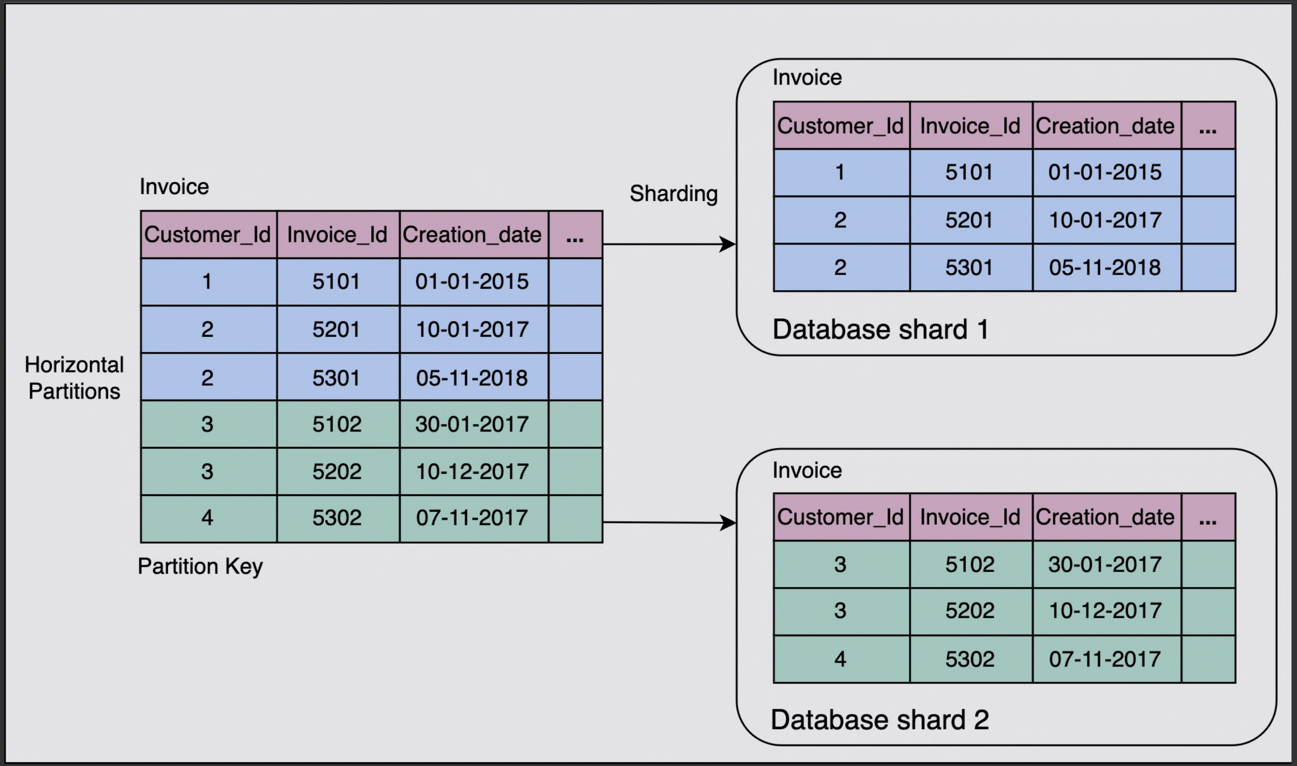
Vertical sharding
We can put different tables in various database instances, which might be running on a different physical server. We might break a table into multiple tables so that some columns are in one table while the rest are in the other. We should be careful if there are joins between multiple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



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.

Key-Range Based Sharding

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



Invoice

Customer_Id	Invoice_Id	Creation_date	...
1	5101	01-01-2015	
2	5201	10-01-2017	
2	5301	05-11-2018	
3	5102	30-01-2017	
3	5202	10-12-2017	
4	5302	07-11-2017	

Database shard 1 — Invoice

Customer_Id	Invoice_Id	Creation_date	...
1	5101	01-01-2015	
2	5201	10-01-2017	
2	5301	05-11-2018	

Database shard 2 — Invoice

Customer_Id	Invoice_Id	Creation_date	...
3	5102	30-01-2017	
3	5202	10-12-2017	
4	5302	07-11-2017	

Example of Horizontal Sharding.

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.

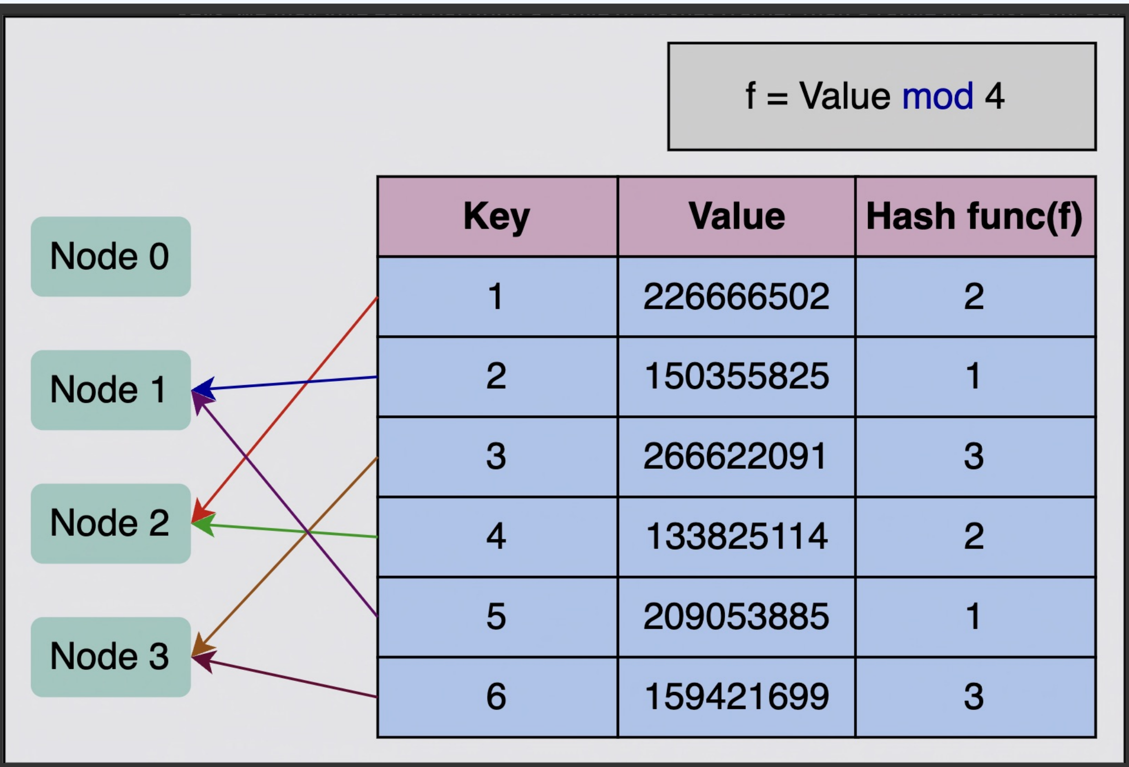
Range queries can be performed using the partitioning keys, and those can be kept in partitions in sorted order. How exactly such a sorting happens over time as new data comes in is implementation specific.

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.



Key	Value	Hash func(f)
1	226666502	2
2	150355825	1
3	266622091	3
4	133825114	2
5	209053885	1
6	159421699	3

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.

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.

Strategies rebalance partition.

→ Avoid hash mod n

→ Fixed number of partitions

→ Dynamic Partitions.

→ Partitions proportionally to nodes.

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.

Trade off's in Database

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 that 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 & processing Speed

a = Total access delay

b = data rate

V = Total data volume

Total communication time, T

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