**Databases**

* A database is an organized collection of data that can be managed and accessed easily.
* There are two basic types of databases:
  + SQL (relational databases)
  + NoSQL (non-relational databases)

**Relational databases:**

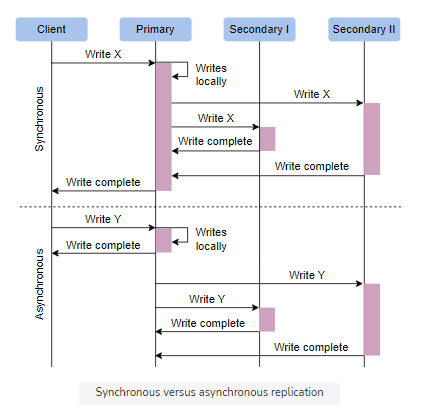
* Relational databases adhere to particular schemas before storing the data. The data stored in relational databases has prior structure.
* A Structure Query Language (SQL) is used for manipulating the database. This includes insertion, deletion, and retrieval of data.
* Relational databases provide the atomicity, consistency, isolation, and durability (ACID) properties to maintain the integrity of the database.
* 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:
  + Flexibility
  + Reduced Redundancy
  + Consistency
  + Integration
  + Backup and disaster recovery
* Drawbacks of relational databases:
  + Impedance mismatch: difference between the relational model and the in-memory data structures.

**Non-Relational (NoSQL) databases:**

* These databases are used in applications that require a large volume of semi-structured and unstructured data, low latency, and flexible data models.
* Advantages:
  1. Semi-Structured and unstructured data support
  2. Availability
  3. Horizontal scaling
  4. Cost
* Disadvantages:
  1. Lack of Standardization
  2. Consistency
* Types of NoSQL databases:
  1. Key-value:
* Key-value databases are efficient for session-oriented applications such as web applications, store users’ data in the main memory or in a database during a session.
* Amazon DynamoDB, Redis and Memcached DB
  1. Document:
     + A document database is designed to store and retrieve documents in formats like XML, JSON, BSON.
     + Document databases are suitable for unstructured catalog data, like JSON files or other complex structured hierarchical data. For example, in e-commerce applications.
     + MongoDB and Google Cloud Firestore
  2. Graph:
     + Graph databases can be used in social applications
     + Graph databases use the graph data structure to store data, where nodes represent entities, and edges show relationships between entities.
     + Neo4J and InfiniteGraph
  3. Columnar:
     + Columnar databases store data in columns instead of rows.
     + 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.
     + Cassandra, HBase, Hypertable and Amazon SimpleDB

**Data Replication**

* Replication refers to keeping multiple copies of the data at various nodes (preferably geographically distributed) to achieve availability, scalability, and performance.
* The main problem in replication arises when we have to maintain changes in the replicated data over time.
* **Synchronous versus 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.



* **Data replication models:**

1. Single leader or primary-secondary replication
2. Multi-leader replication
3. Peer-to-peer or leaderless replication

1. Single leader or 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.
* Advantages of Primary Secondary replication
  + Primary-secondary replication is appropriate when our workload is read-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.
* Dis-advantages of Primary Secondary replication
  + Primary-secondary replication is inappropriate if our workload is write-heavy.
  + 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.
* In case of failure of the primary node, a secondary node can be appointed as a primary node

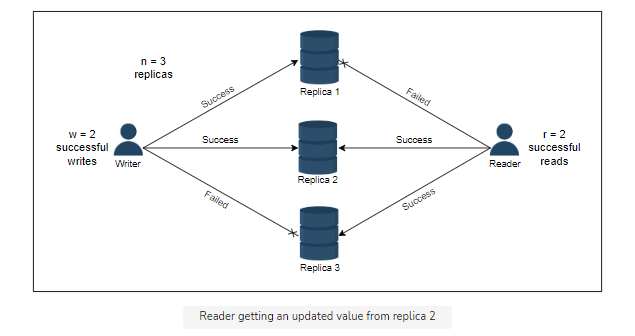
2. Multi-leader replication

* There are multiple primary nodes that process the writes and send them to all other primary and secondary nodes to replicate.
* Since all the primary nodes concurrently deal with the write requests, they may modify the same data, which can create a **conflict** between them.
* Conflict avoidance methods:
  + Conflict avoidance
  + Last-write-wins
  + Custom logic

3. Peer-to-peer/leaderless replication

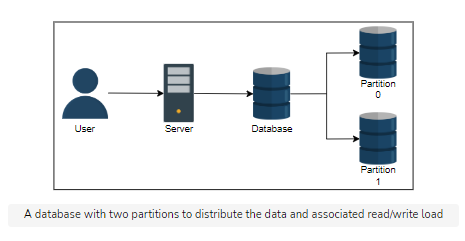
* In primary-secondary replication, the primary node is a bottleneck and a single point of failure. Moreover, it helps to achieve read scalability but fails in providing write scalability. The peer-to-peer replication model resolves these problems by not having a single primary node. All the nodes have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

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.

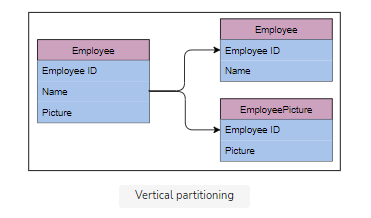


**Data Partitioning**

* 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.



* 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.
* Generally, we use the following two ways to shard the data:
  + Vertical sharding
  + Horizontal sharding
* Vertical sharding
  + 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.



* 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.
  + Each partition of the original table distributed over database servers is called a shard.
  + Usually, there are two strategies available:
    1. Key-range based sharding
    2. Hash based sharding
    3. Consistent hashing
* Rebalance the Partitions:
  + Query load can be imbalanced across the nodes due to many reasons.
  + We can apply the following strategies to rebalance partitions.
    1. Avoid hash mod n
    2. Fixed number of partitions
    3. Dynamic partitioning
    4. Partition proportionally to nodes