Understanding Big Data - 2

Lecture 2

A General View

- Databases are designed to do two things: *store data* and *retrieve data*.
- To meet these objectives, the database management systems must do three things:
  - Store Data Persistently.
  - Maintain Data Consistency.
  - Ensure Data Availability.

You can have consistent data with high availability, but transactions will require longer times to execute than if you did not have those requirements.

A General View

- Big data supports storing and querying *huge amounts of variant data* with *multiple copies*.
  - E.g., Facebook and google stores and process Exabyte and Zettabyte of data.
- So, we need *innovative storage strategies* and technologies that are Scalable, Flexible, Available, and Cost-effective.
- As data volumes increase, it becomes more difficult and expensive to *scale up*. A more appealing option is to *scale out*.
- However, *running over a cluster introduces complexity*—so it's not something to do unless the benefits are compelling.

Outline

In order to understand how to meet these objectives with particular attention to balancing *consistency*, *availability*, and *protection for network failures*, the following topics are introduced in this Lecture

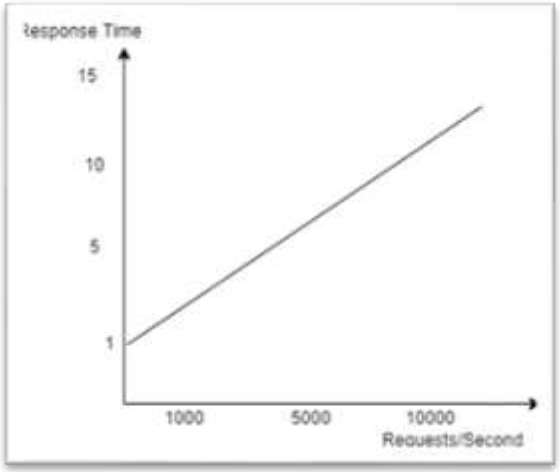
- Scalability
- Data Distribution
- CAP Theorem
- ACID
- BASE
- Challenges for Big Data Infrastructure

Scalability

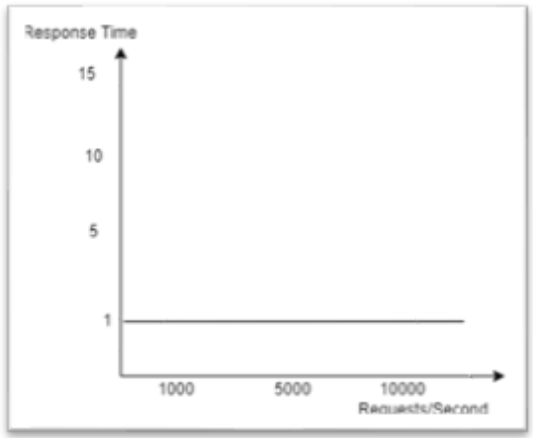
Handling the workload as per the magnitude of the work

- Scalability is the measure of a system's ability to increase or decrease in performance and cost in response to changes in system processing demands.
- It is a property of a system to handle a growing amount of work by adding resources to the system
  - How well a hardware system performs when the number of users is increased
  - How well a database withstands growing numbers of queries.

An overview of scalability principles.



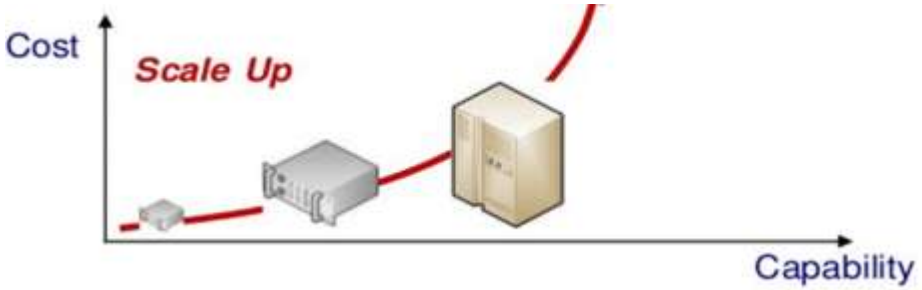
System that do not scale



A scalable System

Scale up vs Scale out

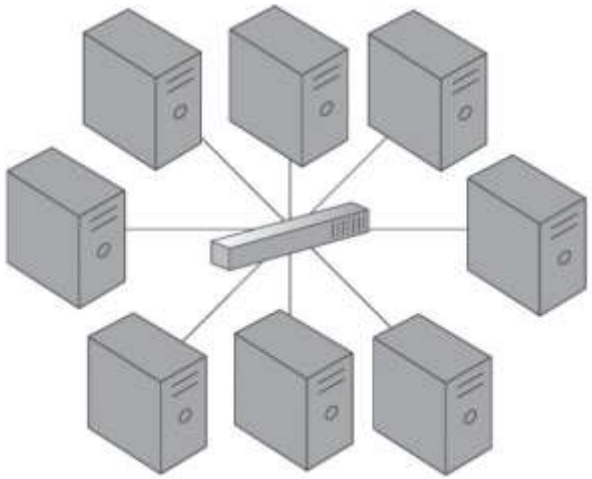
- **Scaling up**
  - Making a component bigger or faster so that it can handle more load.
    - Upgrade storage or processors.



- **Scaling out**
  - Adding more components in parallel to spread out a load.
    - Independent CPU, independent memory, etc.

Clusters

- A cluster is a tightly coupled collection of servers, or nodes connected together via a network to work as a single unit.
- Each node in the cluster has its own dedicated resources, such as memory, a processor, and a hard drive.
- A cluster can execute a task by splitting it into small pieces and distributing their execution onto different computers that belong to the cluster



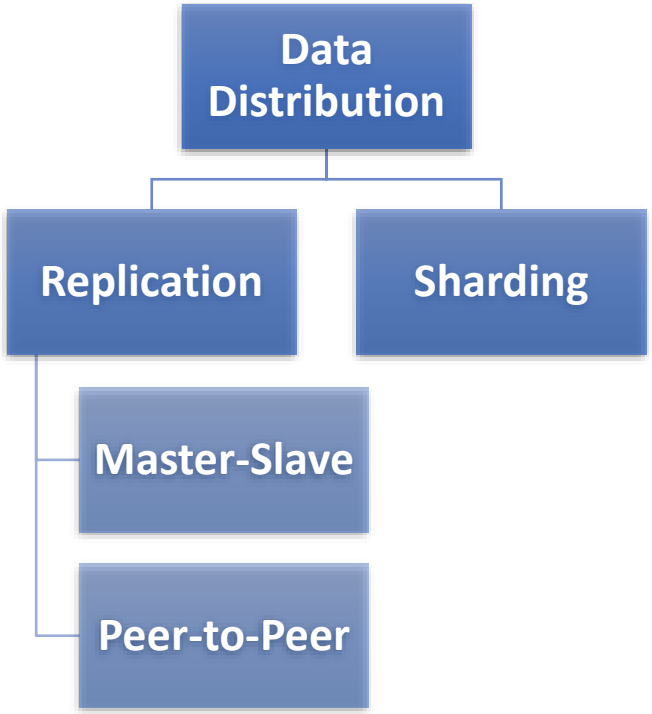
Erl T, Khattak W, Buhler P. *Big data fundamentals: concepts, drivers & techniques*. Prentice Hall Press; 2016 Jan

Distributed File Systems

- A distributed file system is a file system that can store large files spread across the nodes of a cluster.
- To the client, files appear to be local; however, this is only a logical view as physically the files are distributed throughout the cluster.
- This local view is presented via the distributed file system and it enables the files to be accessed from multiple locations.
  - Examples include the Google File System (GFS) and Hadoop Distributed File System (HDFS).

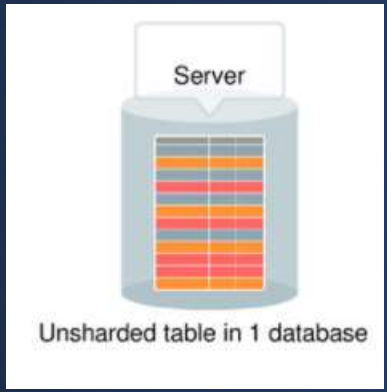
Data Distribution

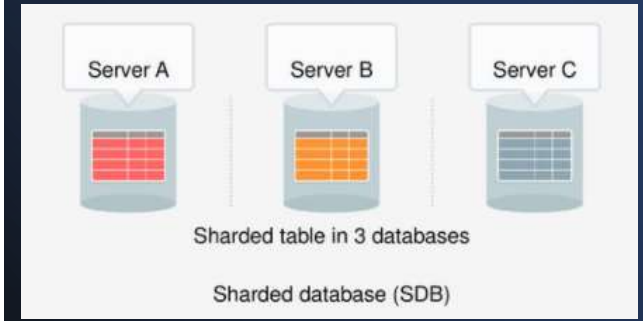
- **Data distribution models**
  - Single server (is an option for some applications).
  - Multiple servers.
- **Orthogonal aspects of data distribution**
  - Sharding: *different data* on *different nodes*.
  - Replication: the *same data* copied over *multiple nodes*.



Sharding

- Sharding is the process of *horizontally partitioning* a large dataset into a collection of smaller, *more manageable* datasets called *shards*.
- The shards are *distributed across multiple nodes*, where a node is a server or a machine.
- Each shard *shares the same schema*, and all shards collectively represent the complete dataset.
- It allows the distribution of processing loads across multiple nodes to achieve horizontal scalability.





Sharding

- Since each node is responsible for only a part of the whole dataset, read/write times are greatly improved.
- It provides partial tolerance toward failures, in case of a node failure, only data stored on that node is affected.
- With regards to data partitioning, query patterns need to be taken into account so that shards themselves do not become performance bottlenecks.
  - For example, queries requiring data from multiple shards will impose performance penalties.
- Data Locality keeps commonly accessed data co-located on a single shard and helps counter such performance issues.

Sharding - Improving Performance

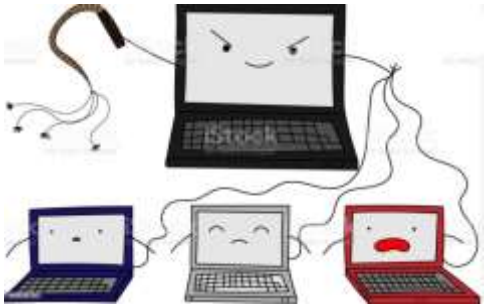
- **Main rules of sharding:**

1. Place the data close to where it's accessed – *Data Locality*
   - Orders form Aswan: data in the Upper Egypt data centre.
2. Try to keep the *load even*
   - All nodes should get equal amounts of the load.
3. Put together data that may be read in sequence
   - Same orders, same node.

Replication

- Replication stores *multiple copies* of a dataset, known as replicas, *on multiple nodes*.
- It provides *scalability* and *availability* since the same data is replicated on various nodes.
- *Fault tolerance* is also achieved since data redundancy ensures that data is not lost when an individual node fails.
- There are two different methods that are used to implement replication:
  - Master-Slave
  - Peer-to-peer

Master-Slave



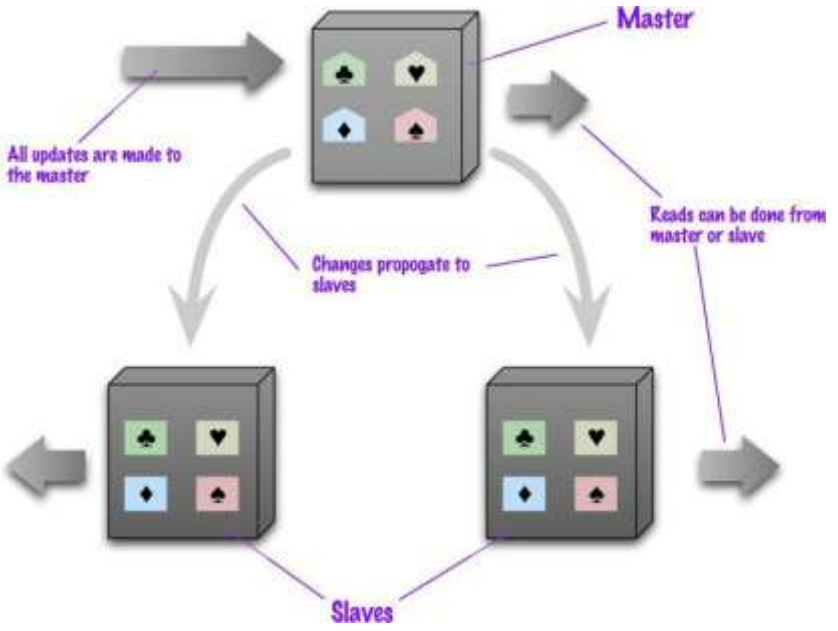
https://www.istockphoto.com/vector/master-slave-topology-gm899039750-248082021

Peer-to-Peer

https://www.dreamstime.com/stock-image-peer-to-peer-network-image26762941

Master-Slave

- In a master-slave configuration, *all data* is written to a *master node*. Once saved, the data is *replicated* over to multiple *slave nodes*.
- All external write requests, including insert, update and delete, occur on the master node.
- Whereas read requests can be fulfilled by any slave node.
- Ideal for *read intensive loads* rather than write intensive loads
  - Write performance will suffer as the amount of writes increases. WHY ???
  - If the master node fails, reads are still possible via any of the slave nodes.

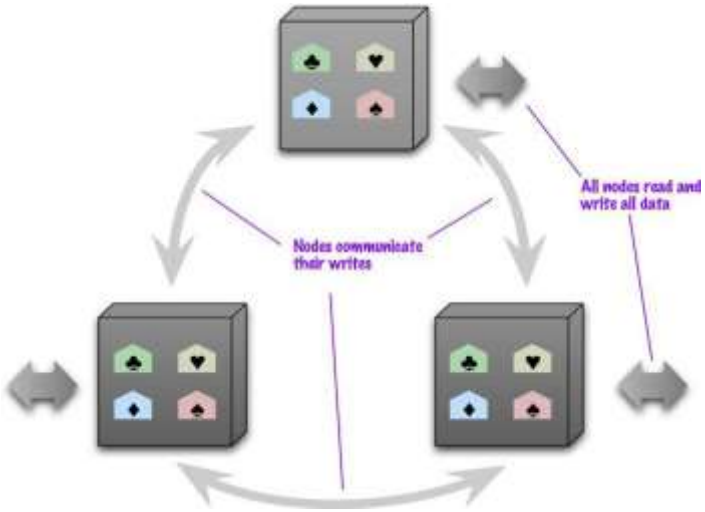


Master-Slave

- A *slave node* can be configured as a *backup* node for the master node.
- In the event that the *master node fails*, writes are not supported until a master node is re-established.
- The master node is either restore to life from a backup of the master node, or a new master node is chosen from the slave nodes.
- One concern with master-slave replication is *read inconsistency*, which can be an issue if a slave node is read prior to an update to the master being copied to it.
  - A voting system can be implemented where a read is declared consistent if the majority of the slaves contain the same version of the record.
  - Implementation of such a voting system requires a reliable and fast communication mechanism between the slaves.

Peer-to-Peer Replication

- All the replicas have equal weight, there is not a master-slave relationship between the nodes.
- The loss of any of them doesn't prevent access to the data store.
- Each node, known as a peer, is equally capable of handling reads and writes.
- Each write is copied to all peers - prone to *write inconsistencies*.

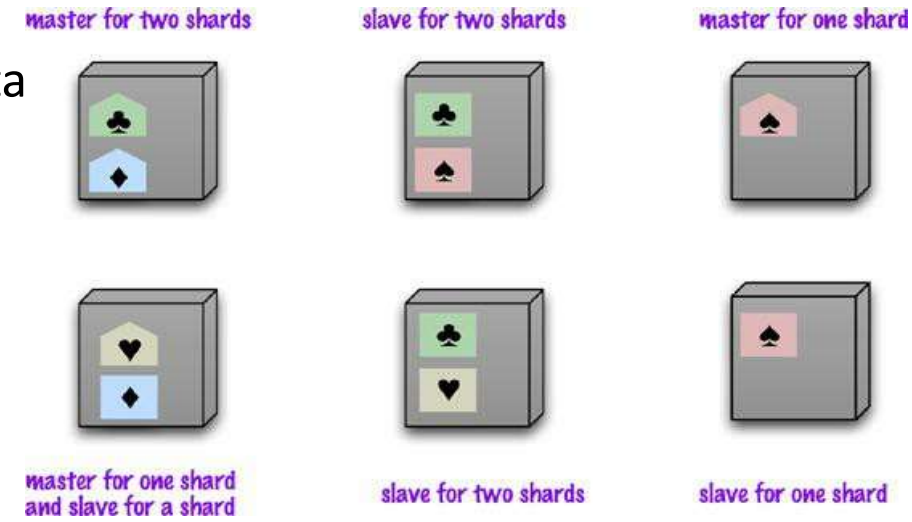


Peer-to-Peer Replication

- Peer-to-peer replication is prone to *write inconsistencies* that occur as a result of a simultaneous update of the same data across multiple peers.
- This can be addressed by implementing either one of the following concurrency strategy:
- Pessimistic concurrency - proactive strategy that prevents inconsistency.
  - It uses locking to ensure that only one update to a record can occur at a time. However, this is detrimental to availability since the database record being updated remains unavailable until all locks are released.
- Optimistic concurrency - reactive strategy that does not use locking.
  - Instead, it allows inconsistency to occur with knowledge that eventually consistency will be achieved after all updates have propagated.
- To ensure read consistency, a *voting system* can be implemented

Combining Sharding and Master-Slave Replication

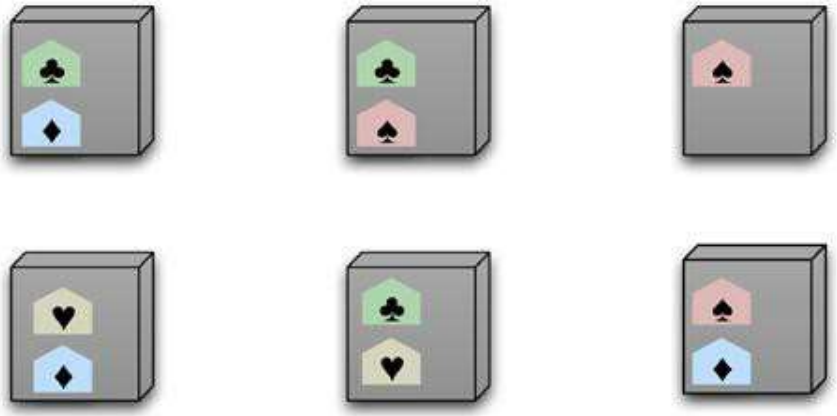
- We have multiple masters, but each data item has a single master.
- Two schemes:
  - A node can be a master for some data and slaves for others.
  - Nodes are dedicated for master or slave duties.



- Write consistency is maintained by the master-shard. However, if the master-shard becomes non-operational, fault tolerance with regards to write operations is impacted.
- Replicas of shards are kept on multiple slave nodes to provide scalability and fault tolerance for read operations.

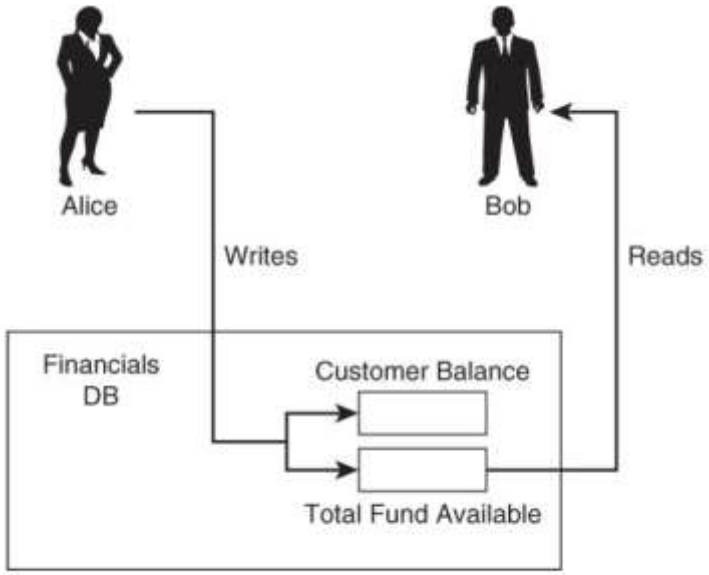
Combining Sharding and Peer-to-Peer Replication

- Each shard is replicated to multiple peers.
- Each peer is only responsible for a subset of the overall dataset.
- Collectively, this helps achieve increased scalability and fault tolerance.
  - As there is no master involved, there is no single point of failure and fault-tolerance for both read and write operations is supported.



Maintain Data Consistency

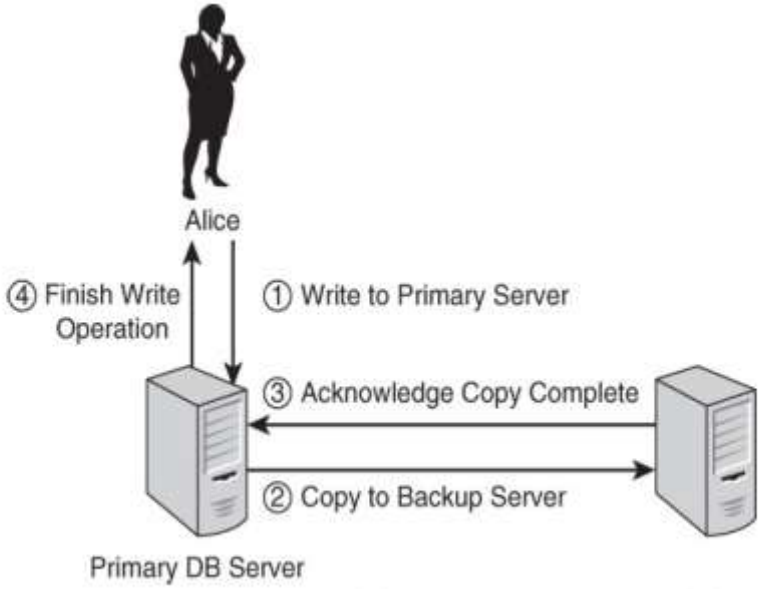
- Consistency means that only valid data, according to all defined rules, will be written to the persistent storage.
- In the context of distributed systems, consistency also refers to maintaining a single and logically coherent view of data in.
- Relational database systems are designed to support consistency by the concept of atomic transaction.



Data should reflect a consistent state.

Ensure Data Availability

- Availability is the degree to which data can be instantly accessed even if a failure occurs.
- One way to avoid unavailability is to have two database servers:
  - Primary Server
  - Backup Server.
- To keep them consistent, a Two-phase Commit Protocol can be used.



Two-phase commits are complete only when operations on both databases are complete.

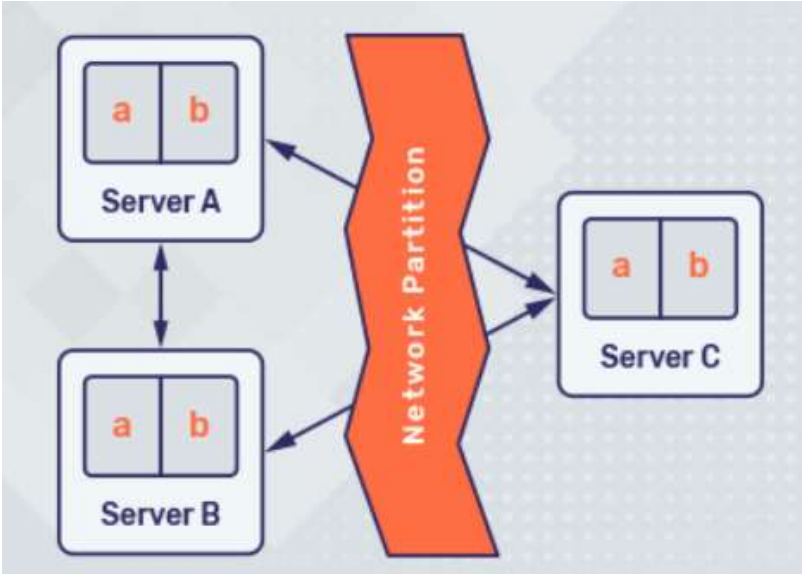
Consistency and Availability

- There is a fundamental *trade-off* between Consistency and availability
- There are two broad options:
  - Ensuring consistency among all replicas on the cost availability.
  - Accepting and coping with inconsistent writes to ensure availability.
- These points are at the ends of a spectrum where we trade off consistency for availability.
- Different domains have *different tolerances for inconsistency*, and we need to take this tolerance into account as we make our decisions.

Network Partition

- Network partitioning is a network failure that causes the members to split into multiple groups such that a member in a group cannot communicate with members in other groups.

- In a partition scenario, all sides of the original cluster operate independently assuming members in other sides are failed.

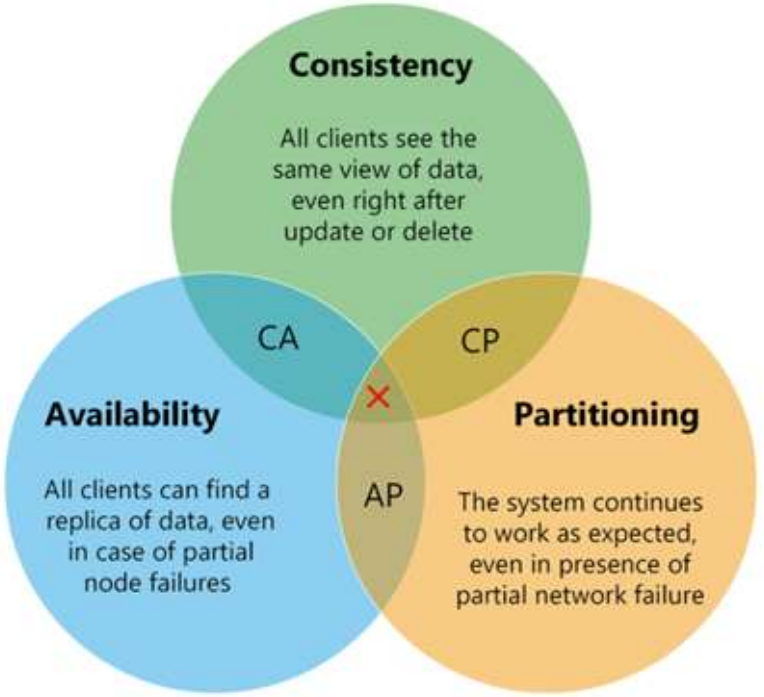


https://www.yugabyte.com/blog/achieving-fast-failovers-after-network-partitions-in-a-distributed-sql-database/

The CAP Theorem

Brewer's Theorem

- A triple constraint related to distributed database systems
  - **Consistency** - read from any node results in the same data across multiple nodes
  - **Availability** - a guarantee that every request receives a response about whether it was successful or failed)
  - **Partition tolerance** - the system continues to operate despite arbitrary message loss or failure of part of the system



"Given the properties of Consistency, Availability, and Partition tolerance, you can only get two"

The CAP Theorem

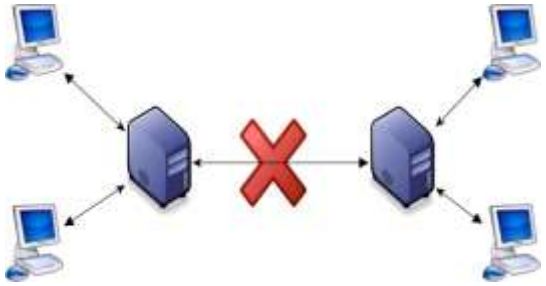
- It is impossible for a distributed system to simultaneously provide all three properties together.
  - If consistency (C) and availability (A) are required, available nodes need to communicate to ensure consistency (C). Therefore, partition tolerance (P) is not possible.
  - If consistency (C) and partition tolerance (P) are required, nodes cannot remain available (A) as the nodes will become unavailable while achieving a state of consistency (C).
  - If availability (A) and partition tolerance (P) are required, then consistency (C) is not possible because of the data communication requirement between the nodes. So, the database can remain available (A) but with inconsistent results.

The CAP Theorem

- In a distributed database, scalability and fault tolerance can be improved through additional nodes, although this challenges consistency (C).
- The addition of nodes can also cause availability (A) to suffer due to the latency caused by increased communication between nodes.
- In distributed database systems, partition tolerance (P) must always be supported; therefore, CAP is generally a choice between choosing either ***C+P*** or ***A+P***.
- The requirements of the system will dictate which is chosen.

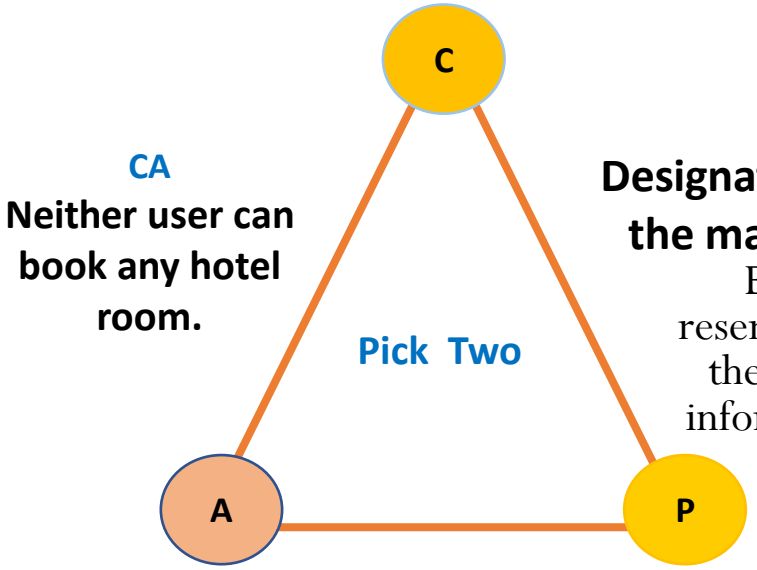
An Example

- Alice is trying to book a room of the ABC Hotel on a node located in London of a booking system.
- Bob is trying to do the same on a node located in Mumbai
- The booking system uses a peer-to-peer distribution
- There is only one room available
- The network link breaks



Possible Solutions

These situations are closely tied to the domain. Financial exchanges, Blogs, Shopping charts.



CA

Neither user can book any hotel room.

CP

Designate Mumbai node as the master for ABC hotel

Bob can make the reservation, Alice can see the inconsistent room information, and cannot book the room.

AP

Both nodes accept the hotel reservation

Overbooking

[CAP Theorem and Distributed Database Management Systems]
Syed Sadat Nazrul | Apr 24, 2018

ACID -- Requirement for SQL DBs

- ACID is a database design principle related to transaction management. It is an acronym that stands for:
  - **Atomicity**. All of the operations in the transaction will complete, or none will.
  - **Consistency**. Transactions never observe or result in inconsistent data.
  - **Isolation**. The transaction will behave as if it is the only operation being performed upon the database (i.e. uncommitted transactions are isolated)
  - **Durability**. Upon completion of the transaction, the operation will not be reversed (i.e. committed transactions are permanent)

Forms of Consistency

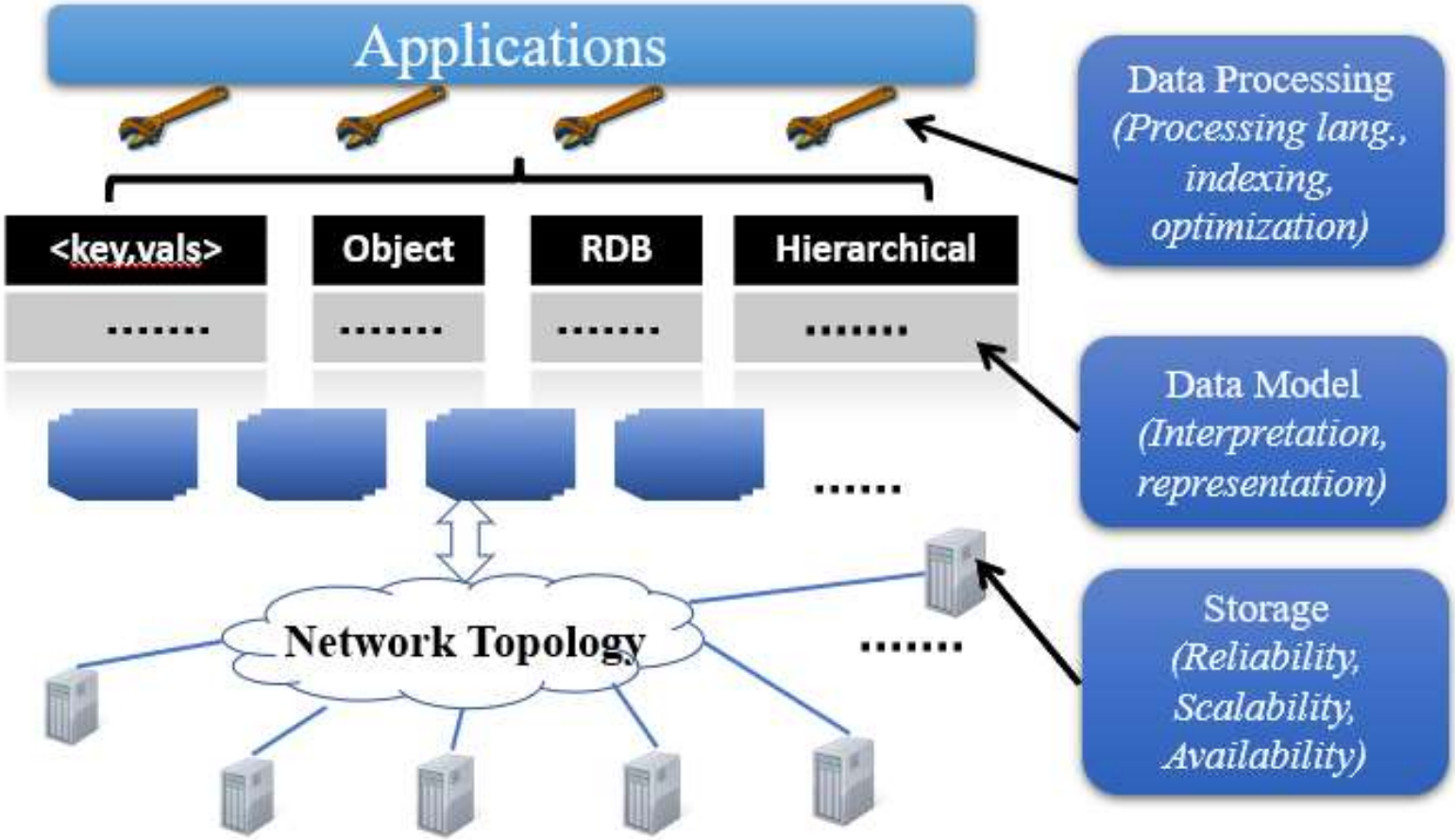
Strong (or immediate) consistency	• ACID transaction
Logical consistency	• No read-write conflicts (atomic transactions)
Sequential consistency	• Updates are serialized
Session (or read-your-writes) consistency	• Within a user's session
Eventual consistency	• You may have replication inconsistencies but eventually all nodes will be updated to the same value

The BASE Property

It favours availability over consistency by relaxing the strong consistency constraints mandated by the ACID properties

- BASE is a database design principle based on the CAP theorem and leveraged by database systems that use distributed technology.
  - ***BA**sically Available* means that there can be a partial failure in some parts of the distributed system and the rest of the system continues to function.
  - ***S**oft State* means data will expire if it is not refreshed - data may eventually be overwritten with more recent data
  - ***E**ventually Consistent* means that there may be times when the database is in an inconsistent state.
    - Casual consistency
    - Read-your-writes consistency
    - Session consistency
    - Monotonic read consistency
    - Monotonic write consistency

Challenges for Current Data Service Infrastructure



Data Model Challenges

Volume	Scale up and scale out
Velocity	"Interactive" properties to facilitate processing
Variety	Simple but unified to adapt heterogeneity

Existing data models are not satisfactory

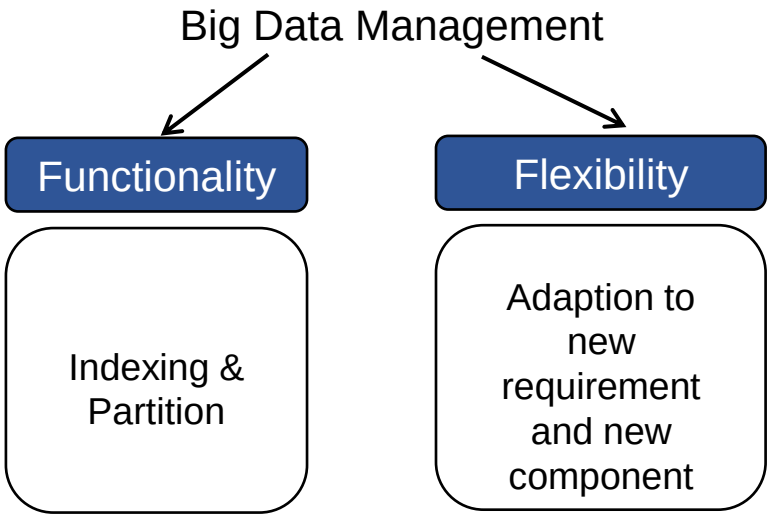
<key,vals>	Object	E-R	Hierarchical
……	……	……	……

Functionality vs. Simplicity

Storage Challenges

- For storage, big data does not propose essential new challenges, because the same challenging problems have been identified when it come to large scale of distributed storage.

- For storage system, it concerns about reliability, availability and scalability. Reliability mainly contains two part: fault tolerance and consistency.
- The CAP theorem is the bottleneck. No one-for-all solution exists



Challenges on Processing Big Data

- For example, new query language (algebra) for big data

Desired	Sacrifices & Overhead
Flexibility	Complexity in data modeling
Relational Supporting	Poor scalability
Uncertain Supporting	Poor scalability and significant computing overhead
Scalability	Less functionality
Efficiency & Effectiveness	Poor scalability

Challenges on Processing Big Data

- For example, new computing paradigm for processing big data

Distributed Computing Paradigm	Limitations
Message Passing	Poor scalability and fault tolerance
Unified Access	Invalidated efficiency over large computing nodes
MapReduce	Poor functionality

Challenges on Processing Big Data

- For example, new optimization methodology for processing big data