Architectural approaches for large scale data management

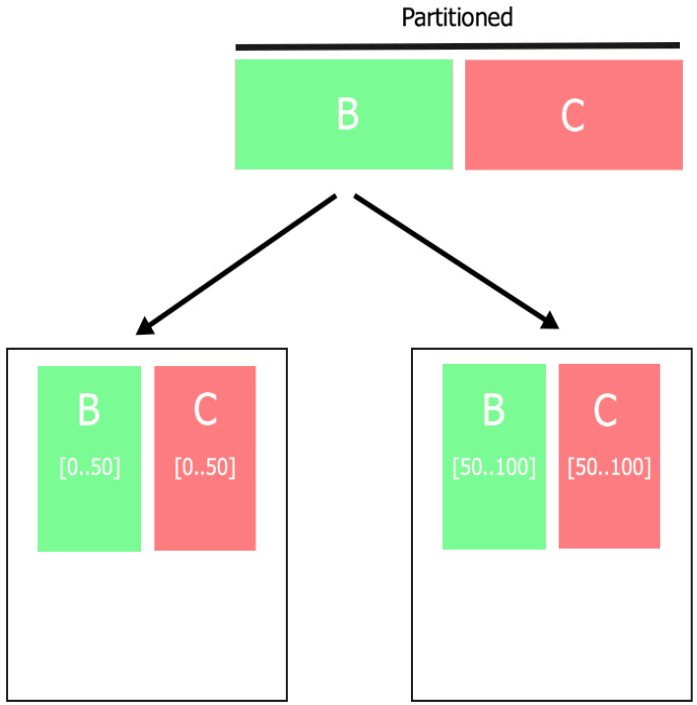
Properties of distributed data systems

- Data distribution
  - Partitioning
  - Replication
- Fault tolerance/Relialibility
- Performance/ Availability
- Scalability

Partitioning

Partitioning

Splitting a big database into smaller subsets called partitions so that different partitions can be assigned to different nodes
(also known as *sharding*)



At the basis of intra-operation parallelism

- different instances of the same operation are processed in parallel, by executing them in parallel over different nodes
- commonly referred to as *data parallelism*
- scalable approach
- at the basis of cluster computing

Assumptions

- Shared nothing architecture
- Scalable distributed data system: spread the data and the query load evenly across nodes
- *Data locality principle*: data placement is a critical performance issue

- *Data* as a set of *n* records R with attributes (*K*,*A*,*B*,*C*) [denoted by R(*K*,*A*,*B*,*C*)]
- *Load*: set of frequent read/write accesses with respect to attribute *K*
- K can be chosen as partition key

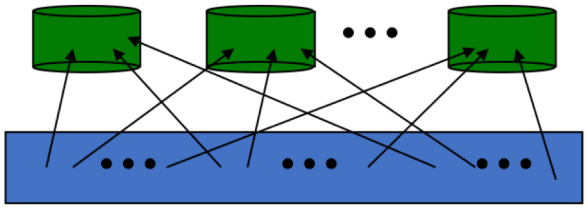
How to partition

- Each dataset is divided in p partitions where p depends on dataset size and access frequency
- Favour balanced partitioning
  - Skewed partitions non scalable (the load is not evenly distributed)
- Avoid hot spots: partitions with disproportionately high load (larger partition, higher number of requests)

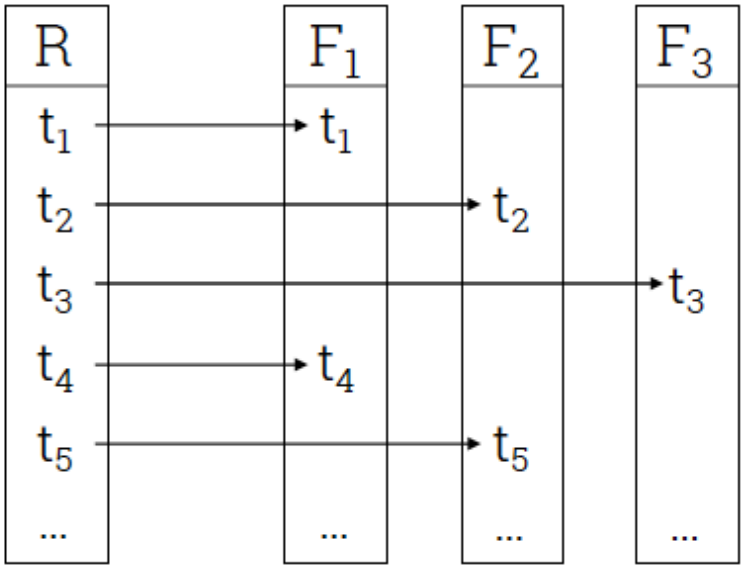
What kind of requests

- Batch query
  - read all data items
  - typical of analytical processing
- Point query
  - selection of a single record
  - typical of transactional processing
- Multipoint/Range query
  - selection of all the records satisfying a given condition (e.g., A > 3)
  - typical of transactional processing

Block-based partitioning



- Arbitrarily partition the data such that the same amount of data *n/p* is placed at each node
- Use a Round-Robin approach or place the first n/p into the first node, the second n/p into the second node and so on
- No hot spots

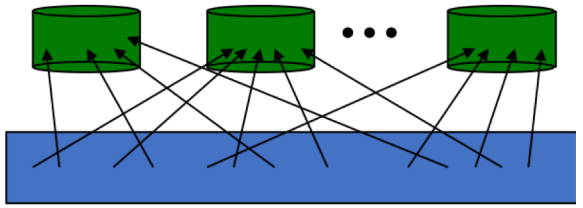


Distributes data evenly
Good for scanning full relation
Not good for point or range queries

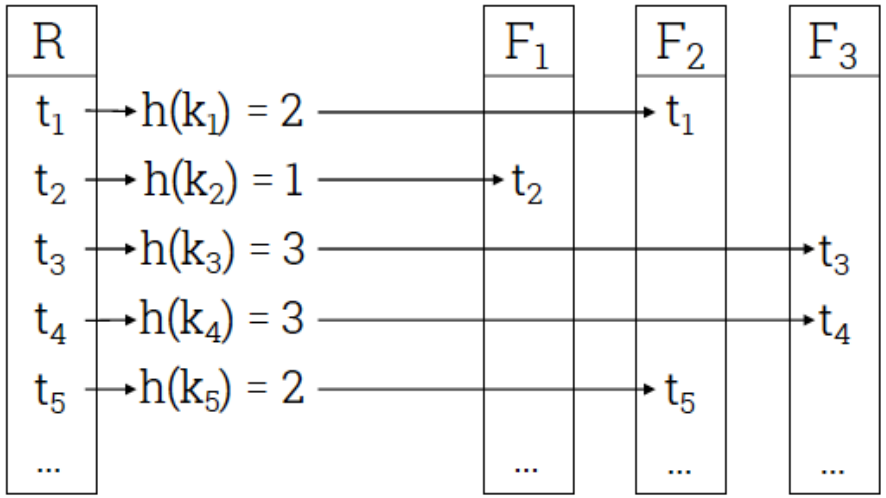
(all nodes have to be accessed and all nodes contains data of interest)

(all nodes have to be accessed but only few nodes might contain data of interest)

Hash-based partitioning



- Applies a hash function to some attribute that yields the partition number in *{1,...,p}*

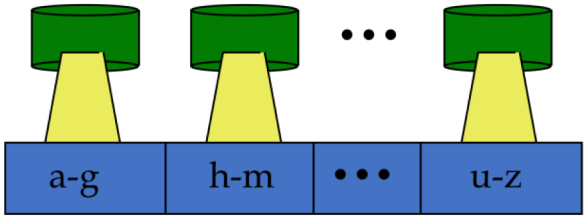


(just one node has to be accessed and the node contains all data of interest)

Distributes data evenly if hash function is good
Good for point queries on key and joins
Not good for range queries and point queries not on key

(all nodes have to be accessed but only few nodes contain data of interest)

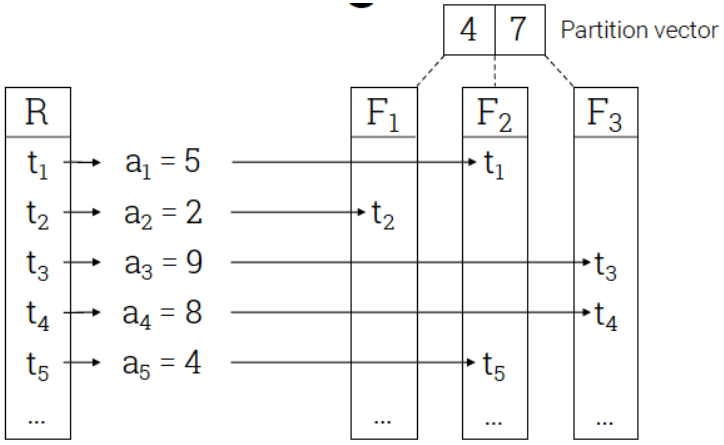
Range-based partitioning



Range-based

- Partition the data according to a specific range of an attribute (such that close values are in the same or nearby servers)
  - find separating points k1, . . . ,kp
  - send to the first node the tuples such that $-\infty \leq t.K \leq k1$
  - Send to the second node the tuples such that $k1 < t.K \leq k2$
  - Send to the n-th node the tuples such that $kp \leq t.K \leq +\infty$
- Partition boundaries might be manually chosen by an administrator or automatically chosen by the system
- Might generate hot spots

(few nodes, those containing data of interest, have to be accessed)



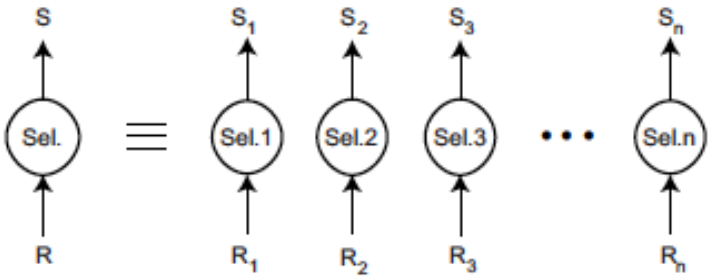
Good for some range queries on partition attribute
Need to select good vector to avoid data/execution skew

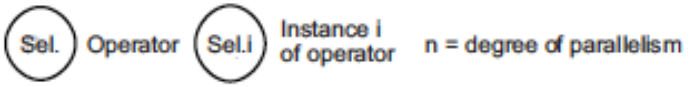
Partitioning

- Each distributed data system supports a specific partitioning scheme
- The partition key can be selected for any dataset
- The selection of the partition key might favour some requests and make some others very inefficient
- Depends on the reference workload
- Choice to be taken at design time

From partitioning to intra-operator parallelism

- Intra-operator parallelism*:* the (read) operator can be decomposed in many sub-operators, each of them executed on a given partition, in an independent way



- The execution  g is limited to nodes containing relevant data (i.e., data contributing to the result)

Partitioning can make the execution of operator supporting intra-operator parallelism more efficient

Partitioning: hot spot

- Hash-based and range-based partitioning help in determining the partition containing a given key and can reduce hot spot when the load is evenly distributed
- When all reads and writes are for the same key, you still end up with all requests being routed to the same partition
- *Partitioning does not help in solving this problem*
- *Replication helps in this case*

Time for exercizes

Partitioning: secondary index

- Partitions work as a kind of global partition key index
- What happens if our queries want to access data with respect to properties that do not correspond to the partition key?
- Partitions cannot help …
- Need for a secondary index

Partitioning: secondary index

- each partition is completely separate
- each partition maintains its own secondary indexes, covering only the documents in that partition
- local index

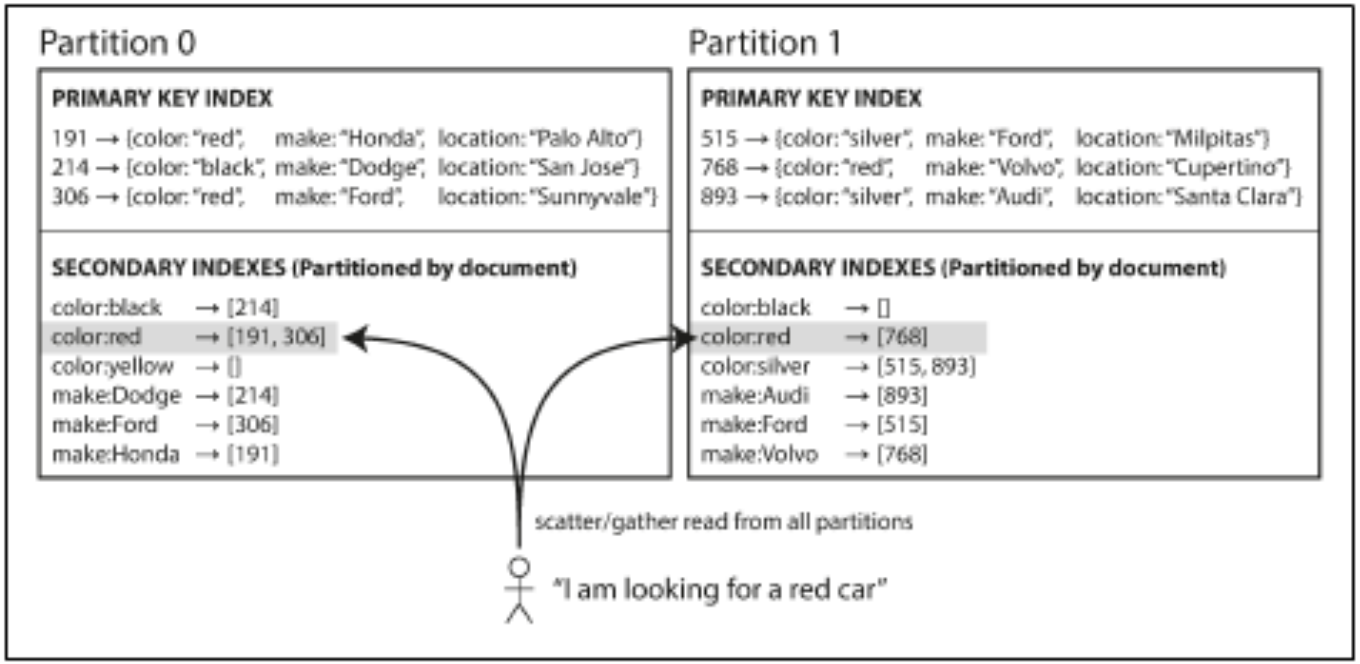


Figure 6-4. Partitioning secondary indexes by document.

- easy write
- read quite expensive
- widely used

Partitioning: secondary index

- a global index covers data in all partitions
- a global index is data as well and must be partitioned, possibly differently from the primary key index

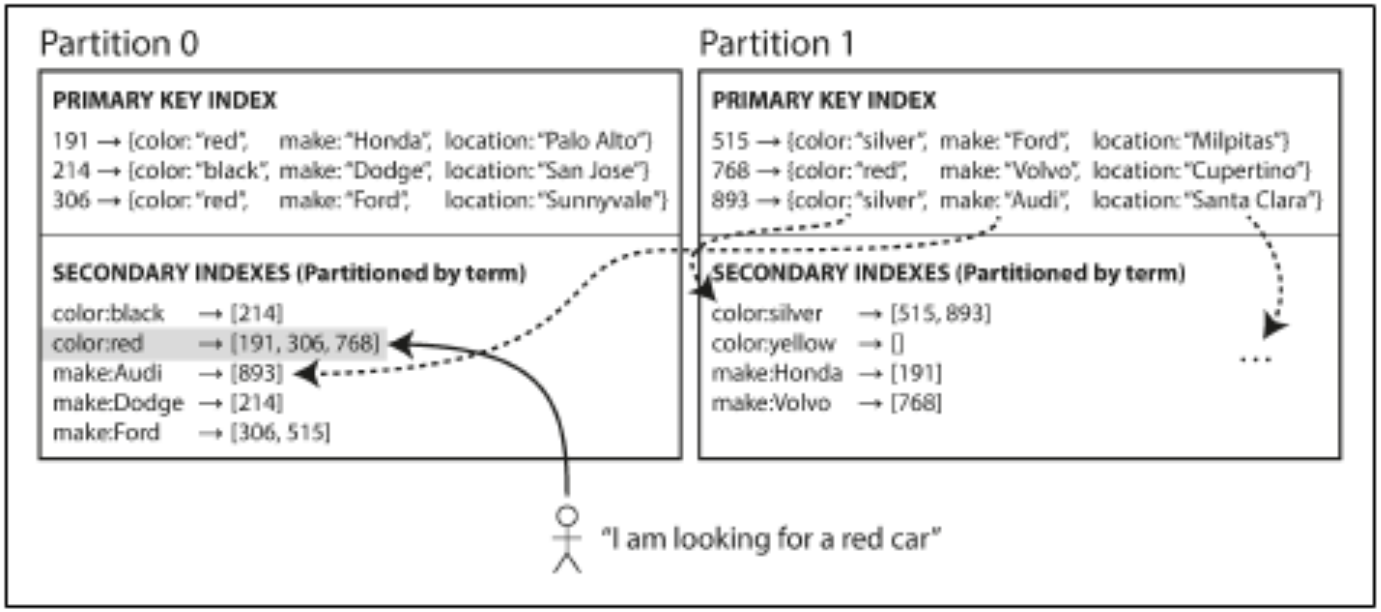


Figure 6-5. Partitioning secondary indexes by term.

- easy read
- expensive write
- rarely used

Rebalancing

- The load may change
  - The number of requests or the dataset size increases, so you want to add more nodes to handle the increased load
- A machine fails, and other machines need to take over the failed machine's responsibilities

Rebalancing: process of moving load from one node in the cluster to another

Example

- Suppose a hash-based partitioning is applied:
  - *H(v) = v mod p*, where *p* is the number of nodes
- If the number of nodes changes, most of the data must be moved

- *Before p = 10*
  H(10) = 0
  H(11) = 1
  H(12) = 2

- *After p = 11*
  H(10) = 10
  H(11) = 0
  H(12) = 1

Rebalancing: automatic or manual

- Rebalancing is an expensive operation, because it requires moving a large amount of data from one node to another
- Fully automatic rebalancing: the system decides automatically when to move partitions from one node to another, without any administrator interaction
  - less operational work to do for normal maintenance
  - it can be unpredictable
- Fully manual rebalancing: the assignment of partitions to nodes is explicitly configured by an administrator, and only changes when the administrator explicitly reconfigures it
  - It's slower than a fully automatic process, but it can help preventing operational surprises
  - *Typically applied in distributed data systems*

Request routing

When a client wants to make a request, how does it know which node to connect to?

- As partitions are rebalanced, the assignment of partitions to nodes changes
- Somebody needs to stay on top of those changes in order to answer the question

Request routing

- Three main approaches
  - Allow clients to contact any node. If that node coincidentally owns the partition to which the request applies, it can handle the request directly; otherwise, it forwards the request to the appropriate node, receives the reply, and passes the reply along to the client
  - Send all requests from clients to a routing tier first, which determines the node that should handle each request and forwards it accordingly. This routing tier does not itself handle any requests; it only acts as a partition-aware load balancer
  - Require that clients be aware of the partitioning and the assignment of partitions to nodes. In this case, a client can connect directly to the appropriate node without any intermediary

Request routing

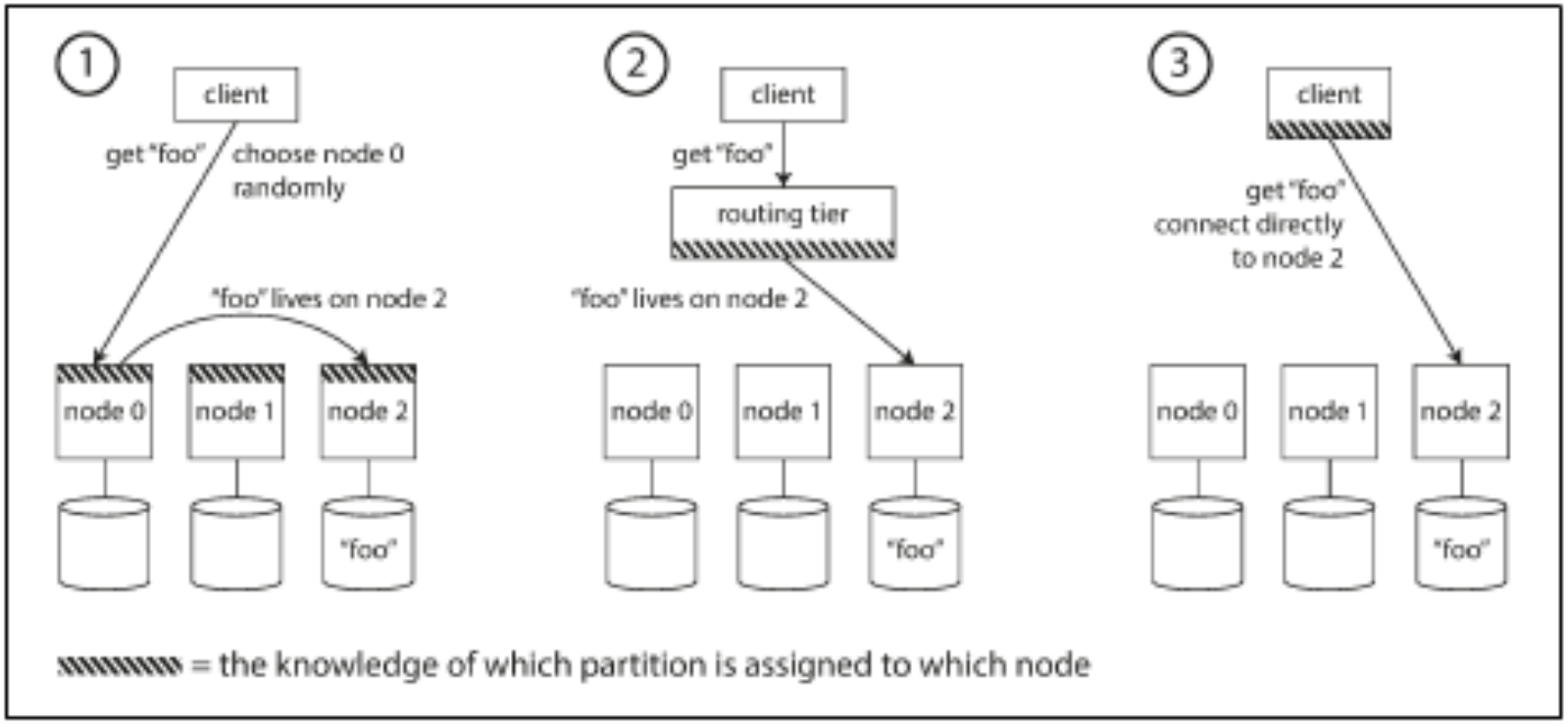


Figure 6-7. Three different ways of routing a request to the right node.

Request routing

How does the component making the routing decision (which may be one of the nodes, or the routing tier, or the client) learn about changes in the assignment of partitions to nodes?

Request routing – coordination service

1. Protocols for achieving consensus in a distributed system, but they are hard to implement correctly
2. Rely on a separate coordination service (e.g., ZooKeeper) to keep track of this cluster metadata

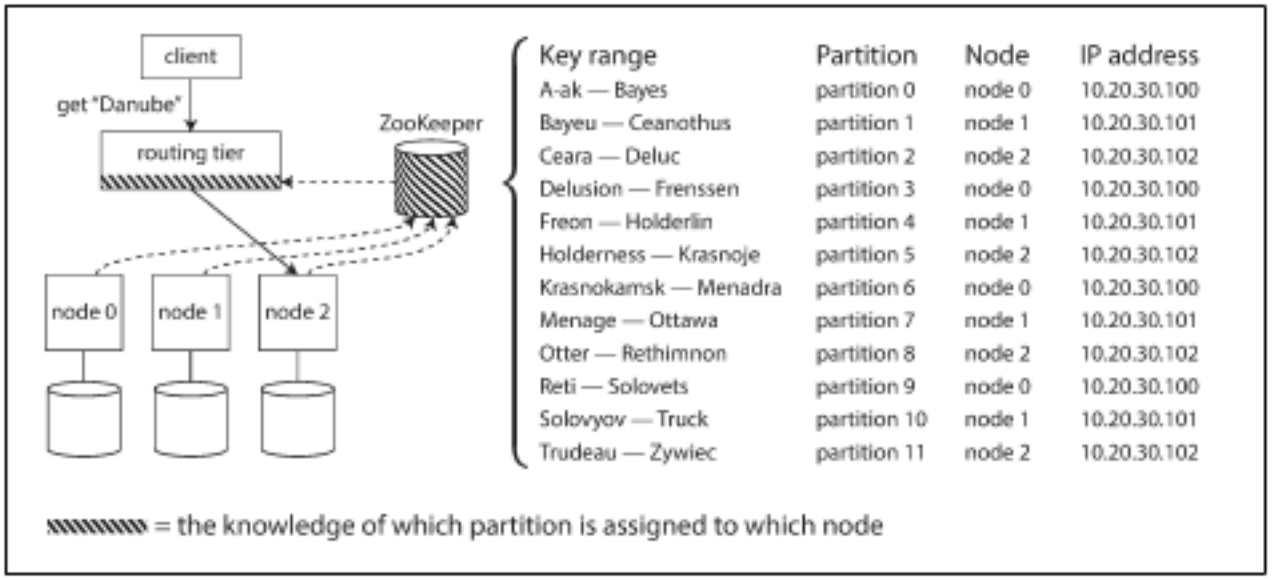
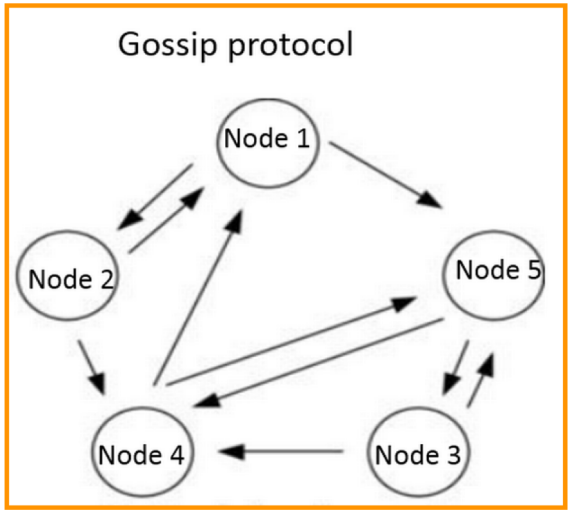


Figure 6-8. Using ZooKeeper to keep track of assignment of partitions to nodes.

Request routing – gossip protocol

- Some systems rely on a gossip protocol among the nodes to disseminate any changes in cluster state
- Requests can be sent to any node, and that node forwards them to the appropriate node for the requested partition
- More complexity in the database nodes but no dependency on an external coordination service



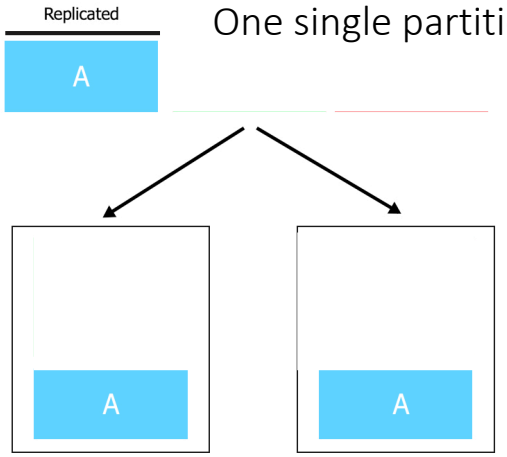
Take away

- Data partitioning is the key mechanism for scalability
- Three main approaches: block-based, range-based, and key-based
- Searches against non partitioning columns require specialized management (secondary indexes)
- Rebalancing (either automatic or manual) is an issue for guaranteeing scaling
- Request routing has to be carefully managed
- Request routing relies on either a coordination service or on some gossip protocol

Replication

Replication

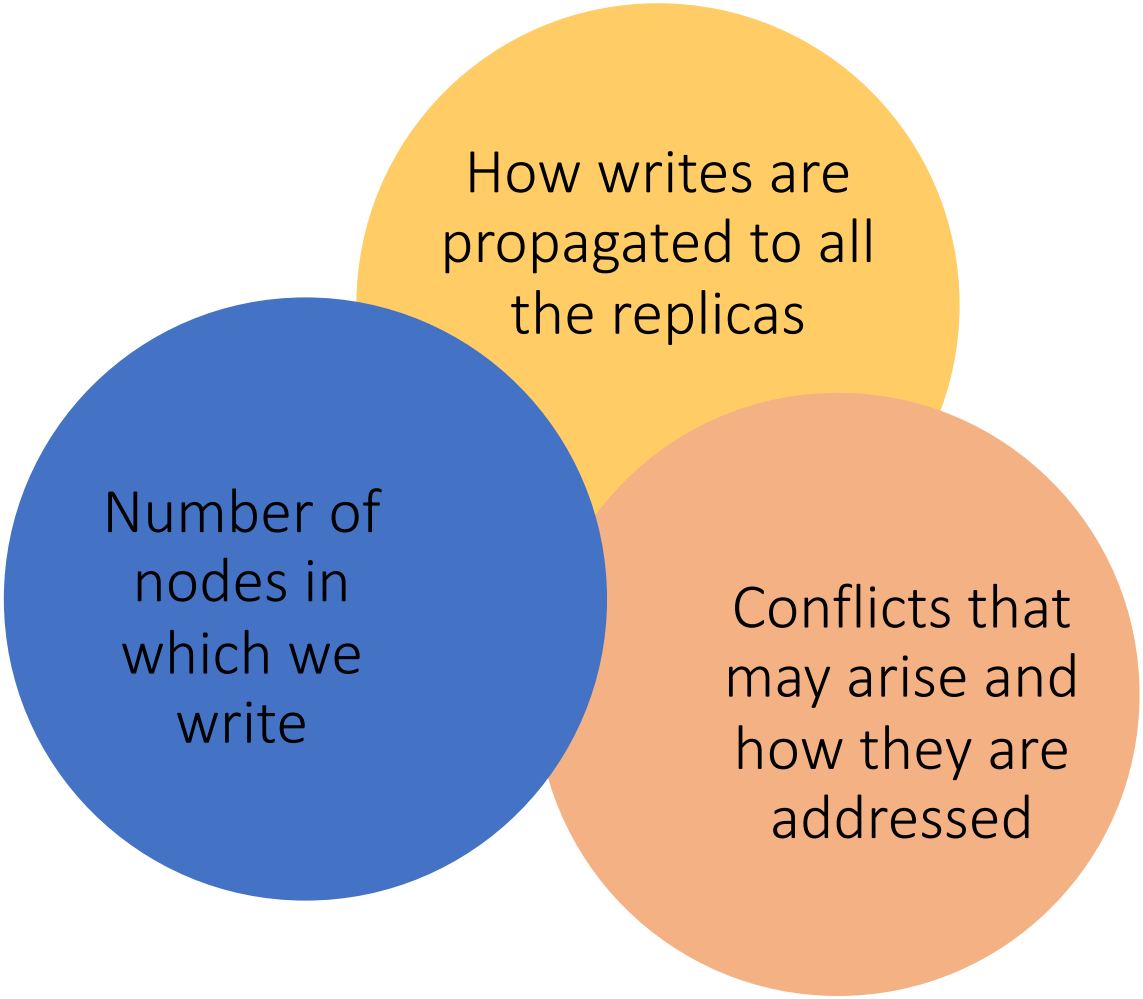
Assumption:
One single partition



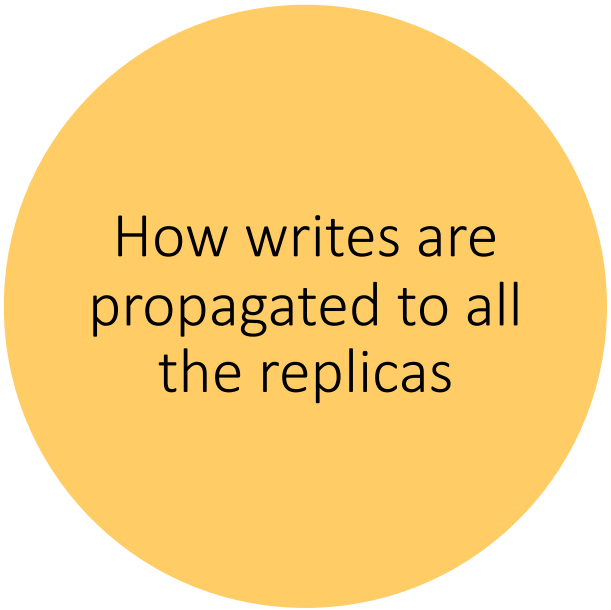
Replication means keeping a copy of the same data on multiple machines that are connected via a network

- Each node that stores a copy of the database is called a replica
- Most of the properties required by a distributed system depend on the replication of data
  - To keep data geographically close to your users (and thus reduce latency)
  - To allow the system to continue working even if some of its parts have failed (and thus increase availability)
  - To scale out the number of machines that can serve read/write queries (on different items) (and thus increase read throughput, scalability)
- Cons
  - Performance. Writing several copies of an item takes more time, which may affect the throughput of the system
  - Consistency. how do we ensure that all the data ends up on all the replicas?

Replication protocols



Replication protocols

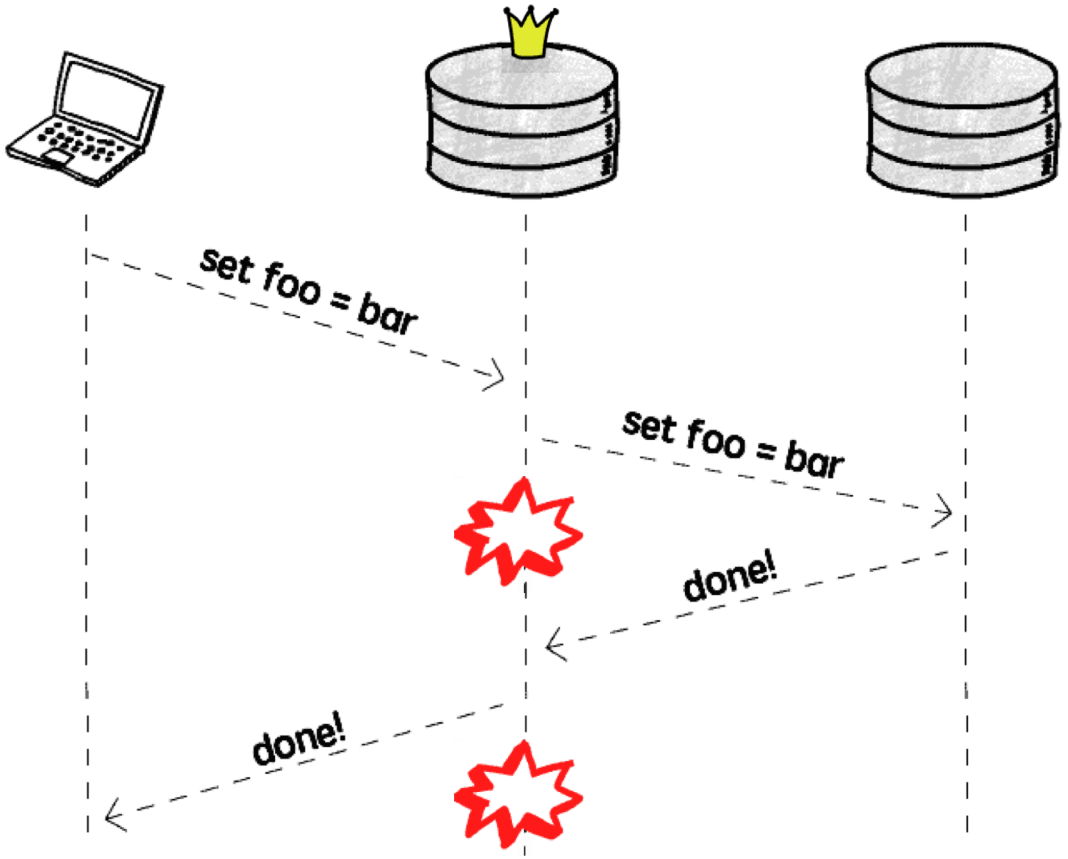


- Synchronous
- Asynchronous

Leader: nodes receiving first the write request

Follower: any other node storing a replica

Synchronous replication



Pros:

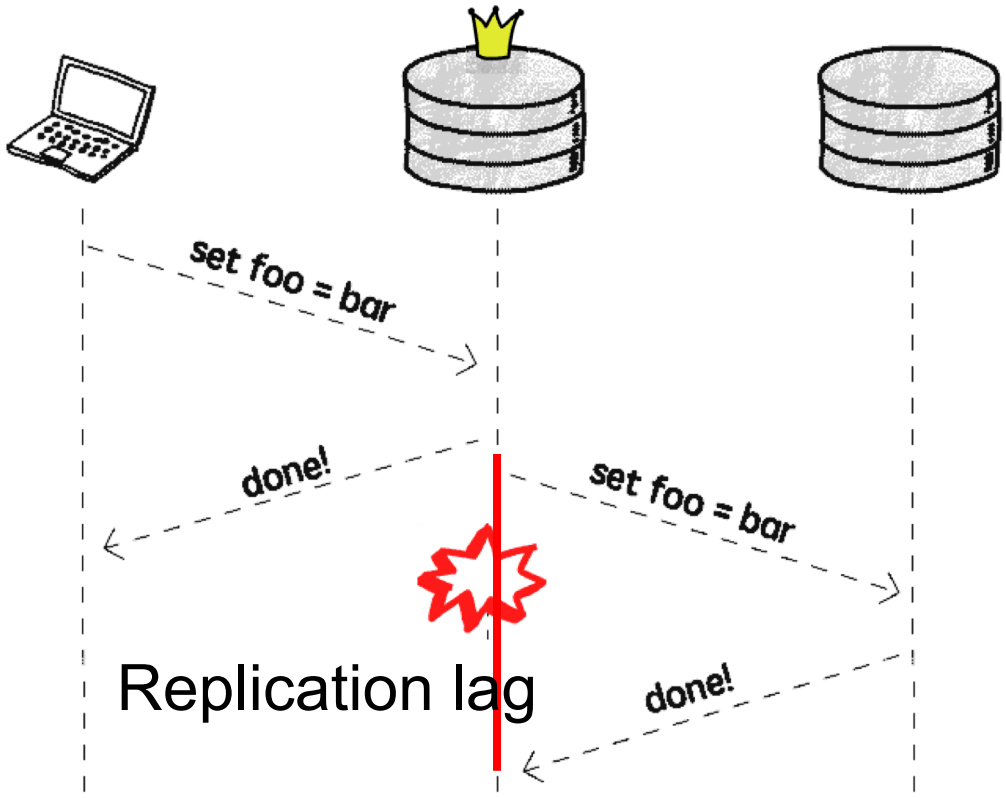
- the follower is guaranteed to have an up-to-date copy of the data (strong consistency)
- if the leader fails, we can be sure that the data is still available on the follower
- write requests are sequentially executed by the leader: no concurrent writes but lower throughput and latency

Cons:

- if the synchronous follower doesn't respond, the write cannot be processed (limited availability)
- applications have to wait for the completion of other clients' requests

the leader waits until the follower has confirmed that it received the write before reporting success to the user, making the write visible to other clients

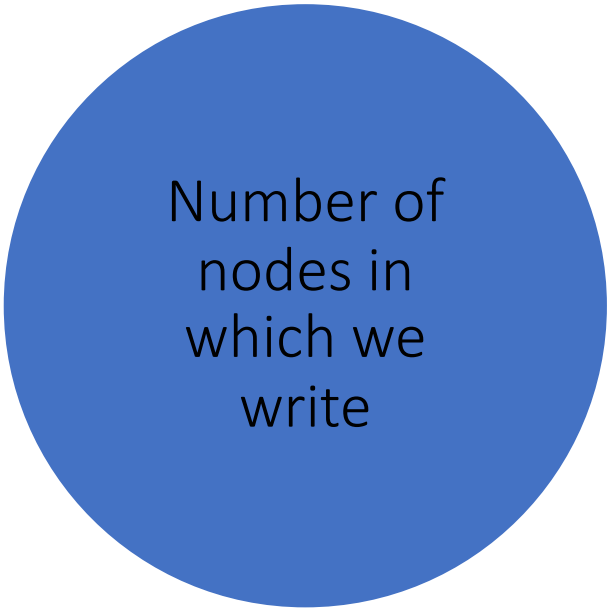
Asynchronous replication



- Pros:
  - the leader can continue processing writes, even if all of its followers have fallen behind (high throughput, low latency)
  - at some point the replicas will become consistent (eventual consistency)
  - replication lag: delay between a write happening on the leader and being reflected on a follower
- Cons:
  - some of the replicas may be out of date (data at some point are inconsistent)
  - if the leader fails and is not recoverable, any writes that have not yet been replicated to followers are lost

the leader sends the message, but doesn't wait for a response from the followers

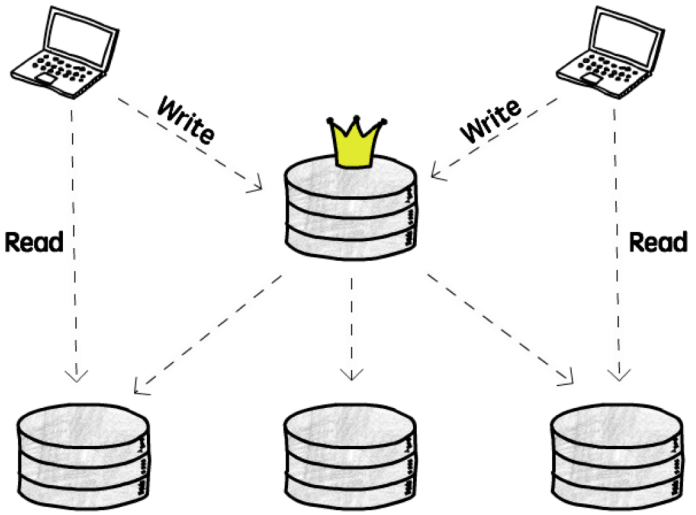
Replication protocols



- Single-leader replication (master-slave replication)
- Multi-leader replication (master-master replication)
- Leaderless replication

Single leader replication

- Just one leader, storing the primary copy of data



READ

- Clients can read from either the leader or any of the followers
- Followers are read-only from the client's point of view

WRITE

- Clients must send their write requests to the leader
  - it first writes the new data to its local storage
  - it sends the data change to all of its followers
- Each follower takes the log from the leader and updates its local copy of the database accordingly

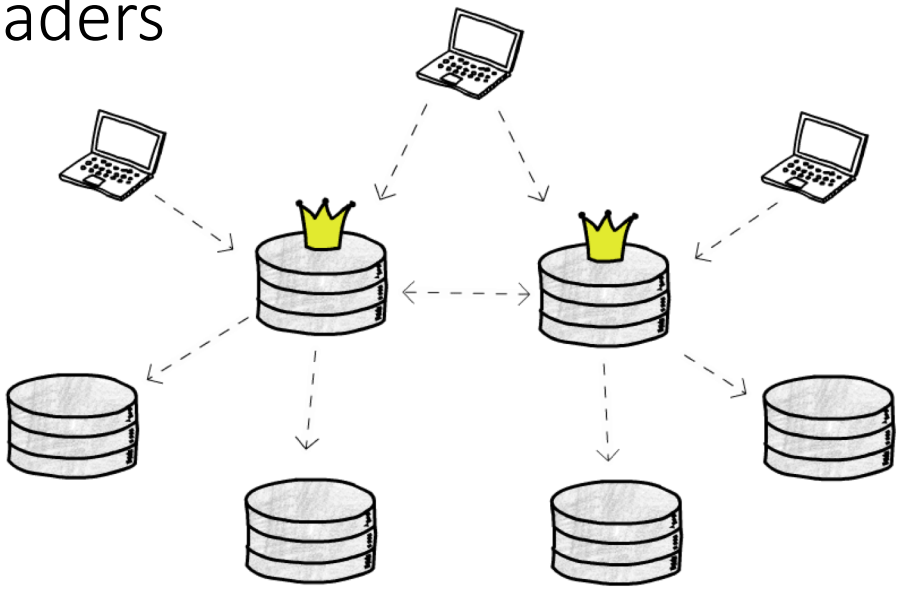
Single leader replication

- Pros
  - Simple implementation
  - No concurrent writes (no write conflicts can arise)

- Cons
  - Limited throughput: all write operations are sequentially executed
  - Limited availability: single point of failure
  - Under an asynchronous protocol, read conflicts (reading old versions of a data item) may arise (see later)

Multi-leader (master-master) replication

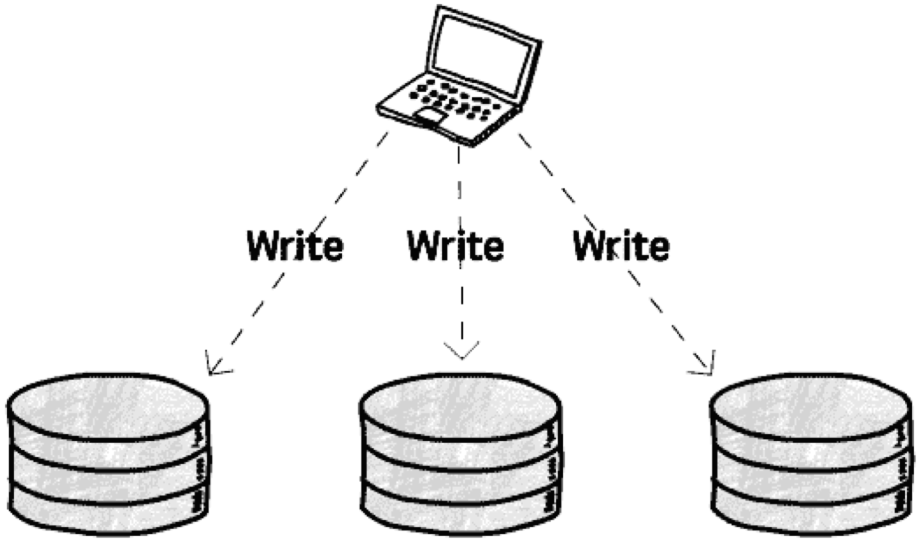
- More than one node can accept writes (more than one leaders, often one for each data center)
- Replication still happens in the same way:
  - each leader node that processes a write must forward that data change to all the other nodes
- Typically asynchronous in order not to loose benefits of multiple leaders



Multi-leader replication

- Pros:
  - increased write throughput
  - increased availability: in case of server failure, writes can be sent to other leaders
- Cons:
  - Under an asynchronous protocol read conflicts
  - the same data may be concurrently modified through different leaders, and those write conflicts must be resolved (see later)
- Usually one leader in each datacenter
  - within each datacenter, regular single leader replication is used
  - between datacenters, each leader (asynchronously) replicates its changes to the leaders in other datacenters

Leader-less replication

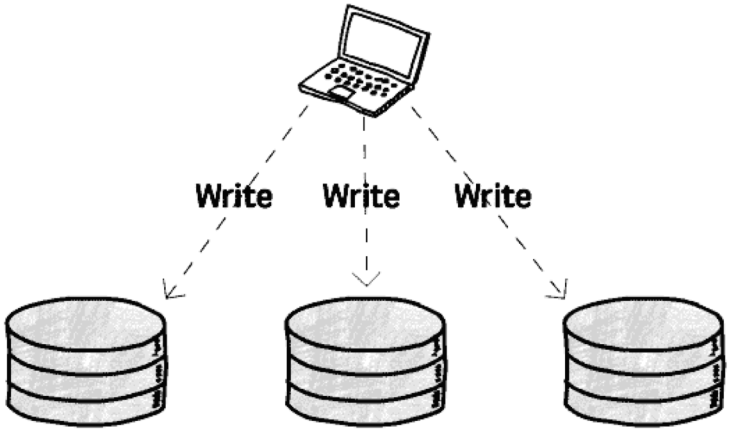


Every replica can accept writes

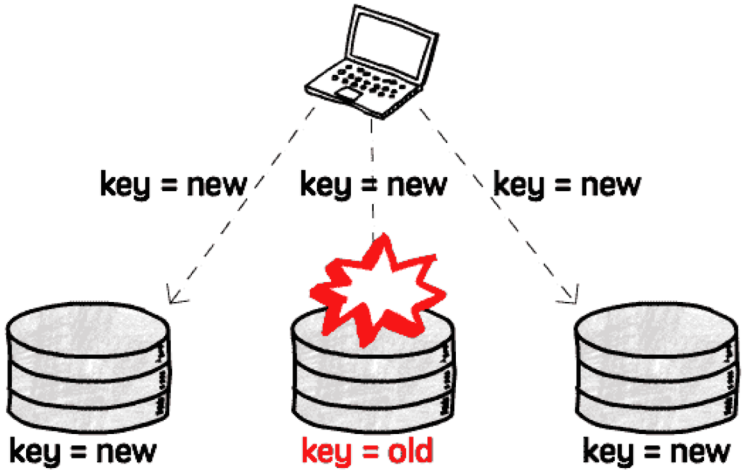
- Pros
  - No leader = no failover
  - High throuput and availability
- Cons
  - Read conflicts
  - Write conflicts

The client sends this write request concurrently to several replicas, and as soon as it gets a confirmation from *some* of them it can consider that write a success and move on.
Typical of P2P systems

Leader-less replication



Write on multiple nodes

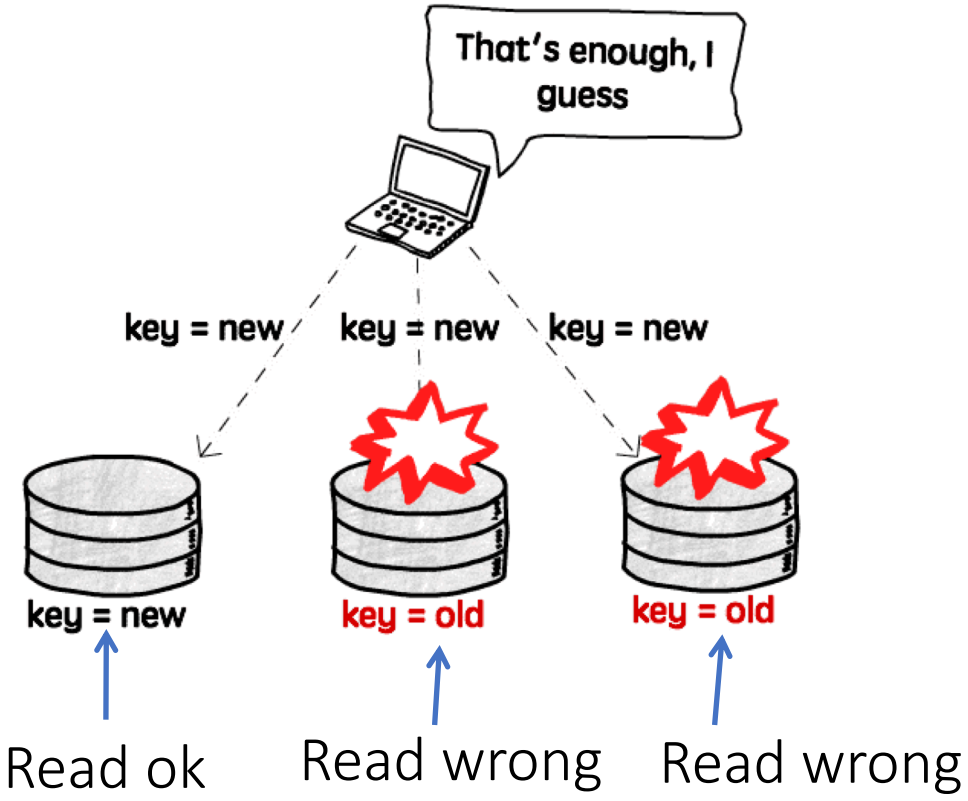


Read from multiple nodes

How to be sure that at least one value is up-to-date?

How to be sure that at least one value is up-to-date?

n: number of replicas
w: number of successfull writes – write quorum
read requests sent to r nodes in parallel – read quorum

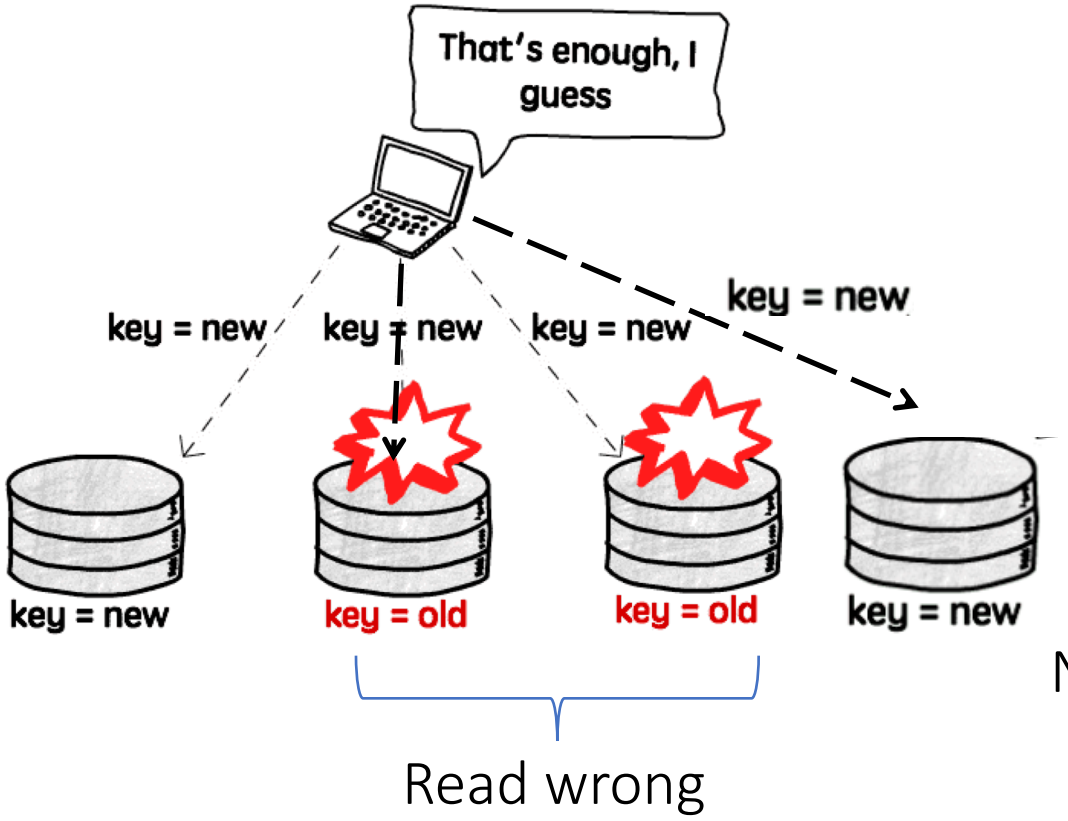


n = 3
w = 1
r = 1

No guarantee of reading the correct value

How to be sure that at least one value is up-to-date?

n: number of replicas
w: number of successfull writes – write quorum
read requests are sent to r nodes in parallel – read quorum



n = 4

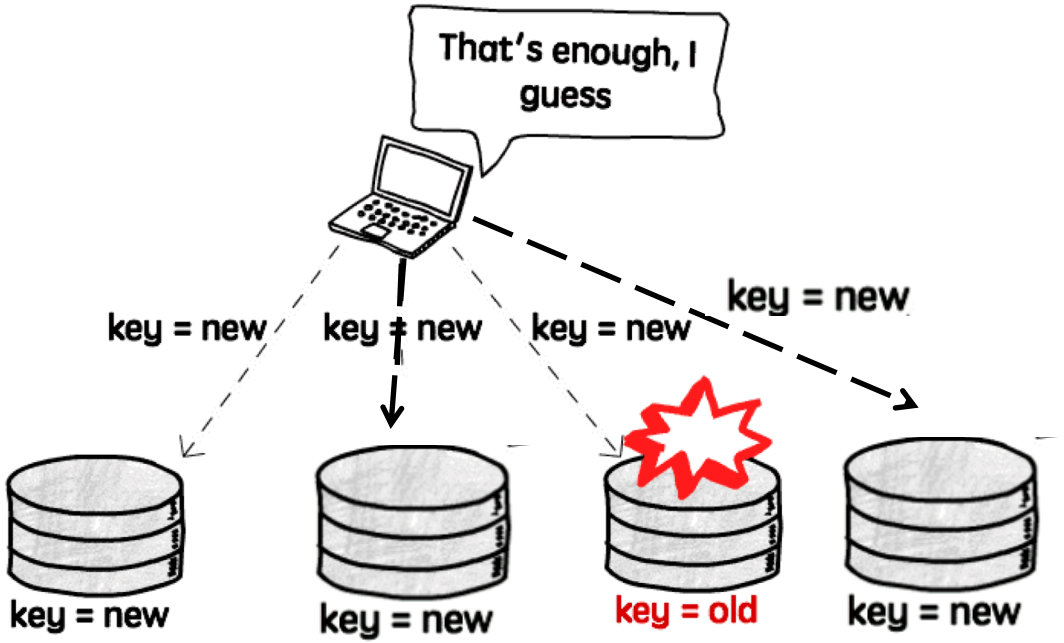
w = 2

r = 2

No guarantee of reading the correct value

How to be sure that at least one value is up-to-date?

n: number of replicas
w: number of successfull writes – write quorum
read requests are sent to r nodes in parallel – read quorum



n = 4

w = 3

r = 2

Any pair of reads returns at least once the correct value

How to be sure that at least one value is up-to-date?

n: number of replicas
w: number of successfull writes – write quorum
read requests are sent to r nodes in parallel – read quorum

$$w+r > n$$

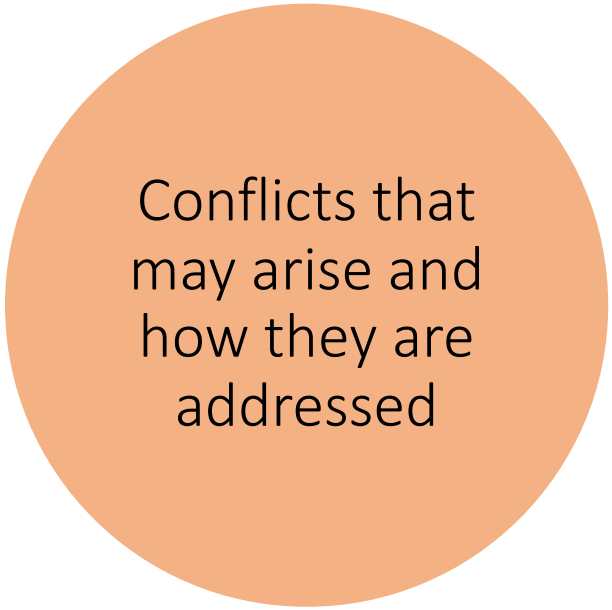
At least one read value is up-to-date

The power of quorums

- w and r are usually configurable

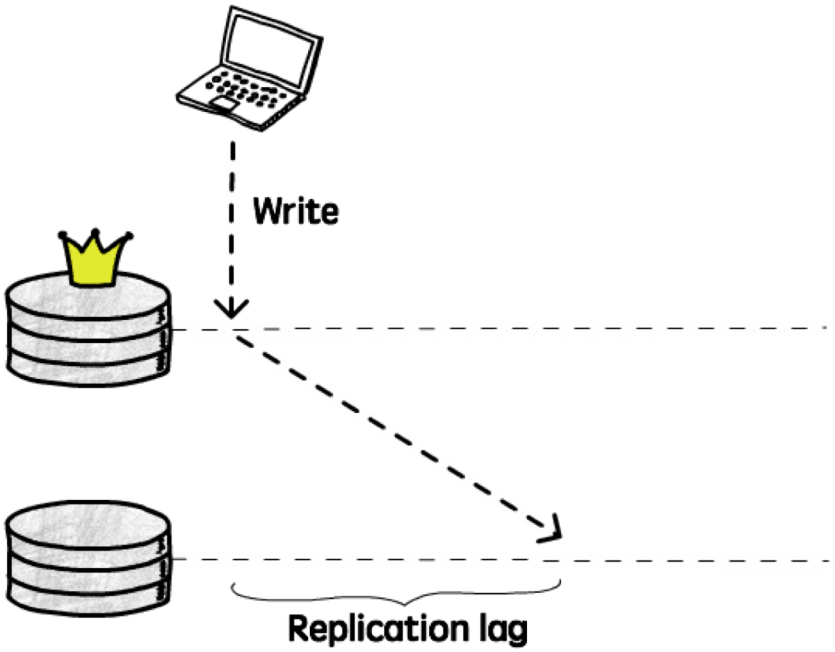
- w < n: we can still process writes if a node is unavailable
- r < n: we can still process reads if a node is unavailable
- n = 3, w = 2, r = 2: we can tolerate one unavailable node
- n = 5, w = 3, r = 3: we can tolerate two unavailable nodes
- w = n, r = 1: fast read but just one failed node causes all database writes to fail

Replication protocols



- Read conflicts
  - receiving wrong data for a read request, after the execution of some write
- Write conflicts
  - conflicts due to writes that are concurrently executed

Read conflict – replication lag



- If a client reads from a replica during the replication lag, it will receive outdated information, because the latest update(s) were not applied yet
- In normal operations, the replication lag may be only a fraction of a second, and not noticeable in practice
- if the system is operating near capacity or if there is a problem in the network, the lag can easily increase to several seconds or even minutes
- a real problem for applications

Typical of asynchronous single-leader and multi-leader replication protocols

Read conflict – replication lag

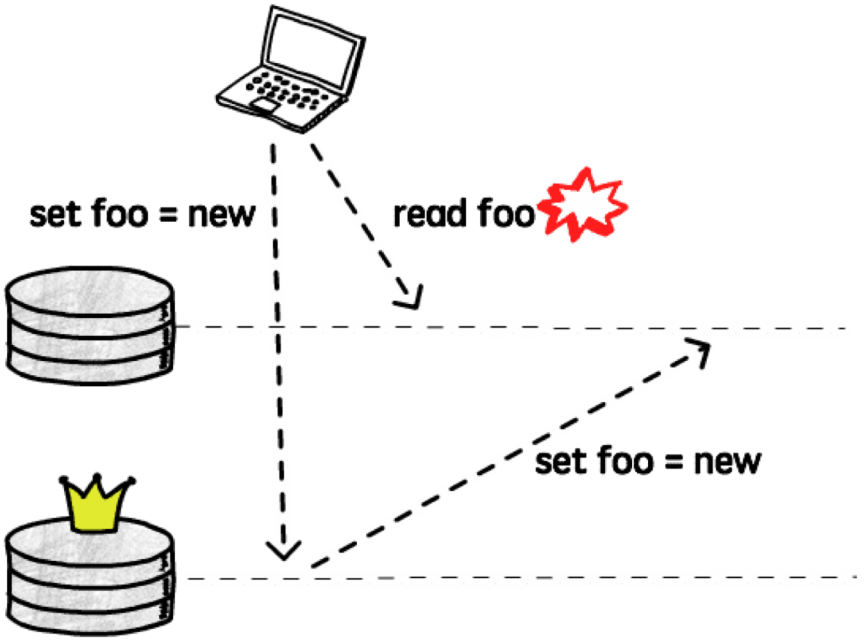
- During the replication lag, data are not consistent, they will become consistent later
- Eventual consistency
- Eventual consistency is not always a problem – it depends on the replication lag and the reference application
- Example
  - You post a new picture on Facebook
  - Your friend is able to see it 30 seconds after you posted it
  - Is it a problem? Probably not

Read conflict

- There are cases where the replication lag and related read conflicts are a real problem and generate specific types of conflicts
- Relevant types of conflicts:
  - Read your write conflict
  - Monotonic read conflict
  - …
- For each type of conflict, a related consistency level is defined if the system is able to avoid it

Read your write conflict

- If the user views the data shortly after making a write, the new data may not yet have reached the replica



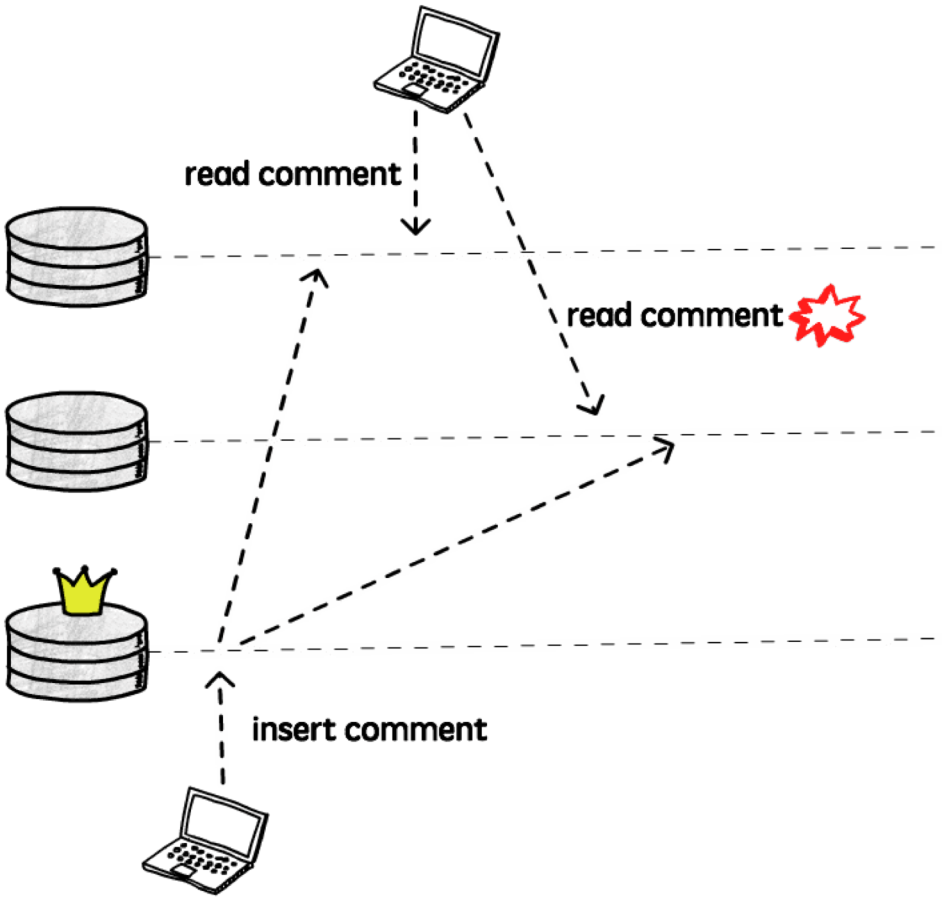
Read your write consistency

A client never reads the database in a state it was before it performed a write

1. When reading something that the user may have modified, read it from the leader; otherwise, read it from a follower.
2. Read from the leader for a certain time window after the update (but defining the right time window is not simple)
3. More sophisticated solutions based on timestamps

Monotonic read conflict

- After users have seen the data at one point in time, they shouldn't later see the data from some earlier point in time



Monotonic read consistency

If you make several reads to a given value, all the successive reads will be at least as recent as the previous one.

Time never moves backwards

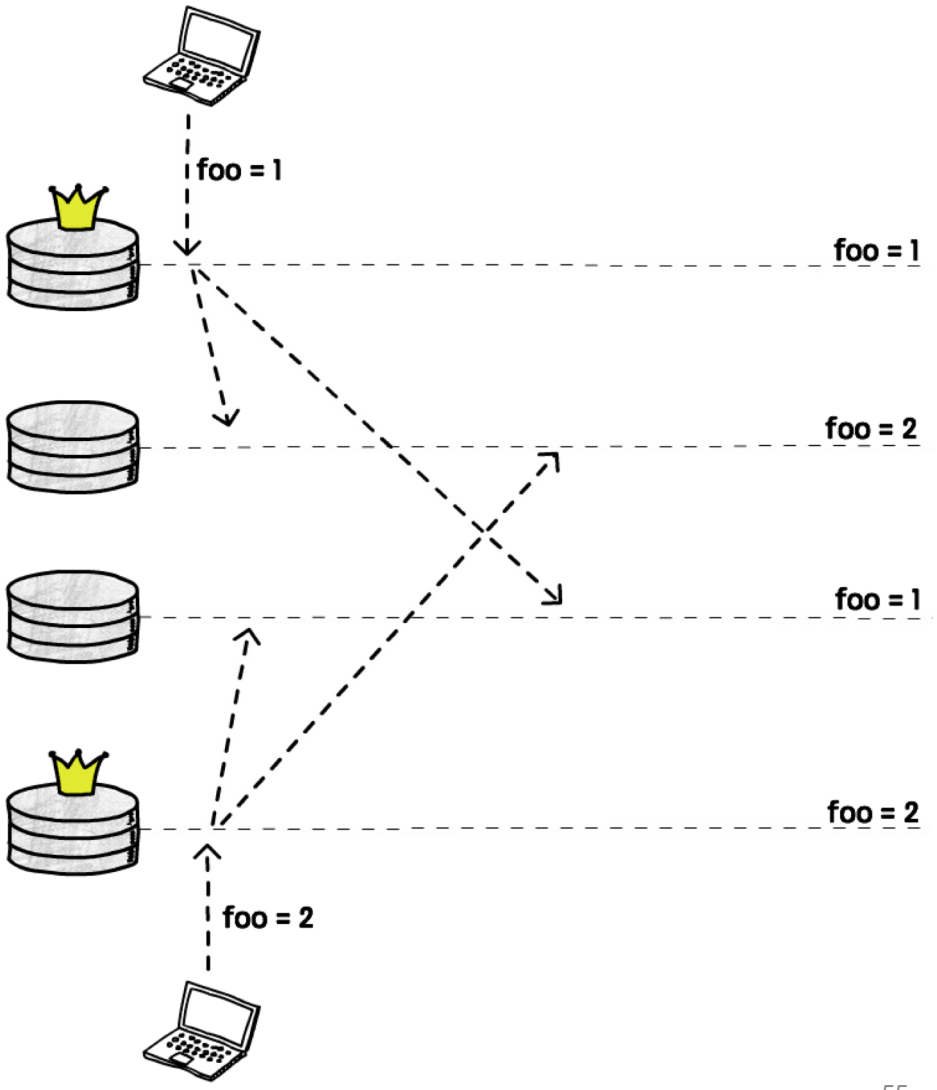
1. Each user always makes reads from the same replica (different users can read from different replicas)
2. More sophisticated solutions based on timestamps

Read conflicts and leader-less replication

- Read conflicts might happen also under leader-less replication
- Quorums appear to guarantee that a read returns at least once the latest written value
- in practice this is not so simple
  - If a write happens concurrently with a read, the write may be reflected on only some of the replicas
  - In this case, the read may return the old or the new value
- guarantees like read your writes and monotonic reads are not always achieved

Write conflicts (concurrent writes)

- Multi-leader and leaderless replication protocols allow several clients to concurrently write to the same item through two different leaders
- *Events may arrive in a different order at different nodes*, due to variable network delays and partial failures
- Write conflicts may occur even if quorums are used

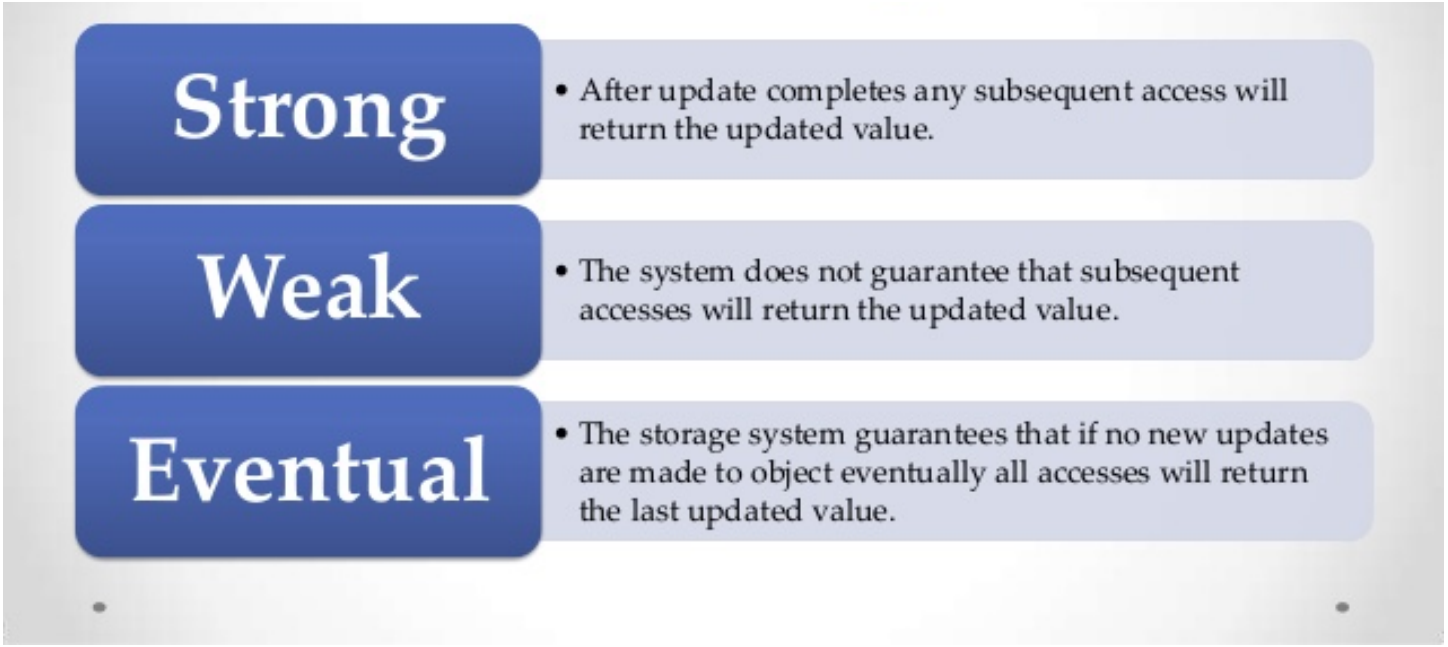


Write conflict – consistency

Avoid multiples values for the same replica, due to concurrent writes

1. No concurrent writes
2. More sophisticated solutions based on timestamps
3. In leader-less replication protocols, write some system-level custom conflict resolution code, when a write/read operation is executed, execute the code and choose one of the replicas as the correct one

Replication and consistency: recap



Read-your-write consistency

Eventual consistency but no conflict of type read-your-own-write

Monotonic read consistency

Eventual consistency but no conflict of type monotonic read

Monotonic write consistency

Eventual consistency but no write conflicts for write operations executed by the same process

...

Take away

- Synchronous vs asynchronous protocols
- Three main approaches, depending on how many leaders are available
- Read conflicts can arise with asynchronous protocols
- Write conflicts can arise when more than one node accepts writes
- Eventual consistency is the overall guarantee we want to achieve
- In general, trade-offs around replica consistency, durability, availability, and latency
- Many approaches for transferring write information
- Under read intensive scenarios, single leader could be a good option
- Under write intensive scenarios, multi-leader or leader-less protocols are better options
- Leader-less protocols often used in P2P systems

Faults and reliability: basic issues

Fault Tolerance – Centralized Systems

CENTRALIZED SYSTEMS

- if a program fails for any reason, the simple solution is to abort then restart the process
- either it works or it doesn't, total failure
- key concept for guaranteeing reliability: transaction

DISTRIBUTED SYSTEMS

- failure is a possibly frequent situation
- restarting is not always a good idea
- partial failure
- transactions still exist but ACID properties are revised

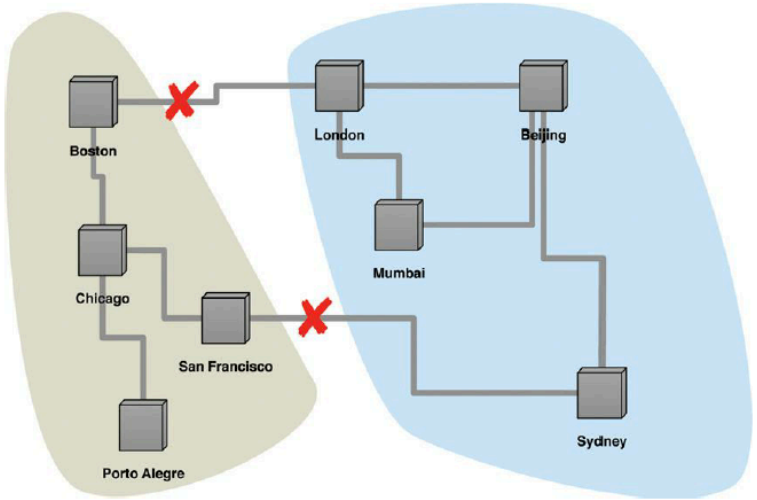
Fault Tolerance

Independence: the task handled by an individual node should be *independent* from those handled by the other components

- In case of failure, simply replace the node with another one
- Independence is best achieved in shared-nothing architectures
- Replication helps

Fault Tolerance

- Network partition/network fault: one part of the network is cut off from the rest due to a network fault
- Such faults can always occur and distributed data systems needs to be able to handle them



Issues

- How to detect that a system met a failure?
  - the client can wait for the failed node to come back (at least for a given amount of time, timeout)

- How to recover from the failure? (after the timeout)
  - Either abort the process and report an error (atomic/transactional behavior, limited availability) or
  - the failure is solved (through node replacement) and the process can be completed (more complex approach)
  - In both cases: reliability and recovery, based on consensus protocols
  - Systems do everything for us

Consensus protocol: example Leader election

- In single-leader replication, leader fault
- A new leader has to be identified (consensus protocol)
  - predefined successor node
  - choose the node that has the most recent update to minimize data loss
- All write requests must be sent to the new leader
  - Request re-routing
- All nodes need to agree on which node is the leader
  - What happens if the old leader comes back?
  - If there were two leaders, they would both accept writes and their data would diverge, leading to inconsistency and data loss
- Sometimes a manual procedure could be better …

Take away

- Different kind of failures can arise
- A distributed system should be able to recover from failures
- Node independence simplifies failure management: the task handled by an individual node is independent from those handled by the other components
- Task replication is also an useful principle for failure management
- Two main issues: how to detect a fault and how to recover it
- Fault detection is often based on timeout
- Recovery often based on consensus protocols

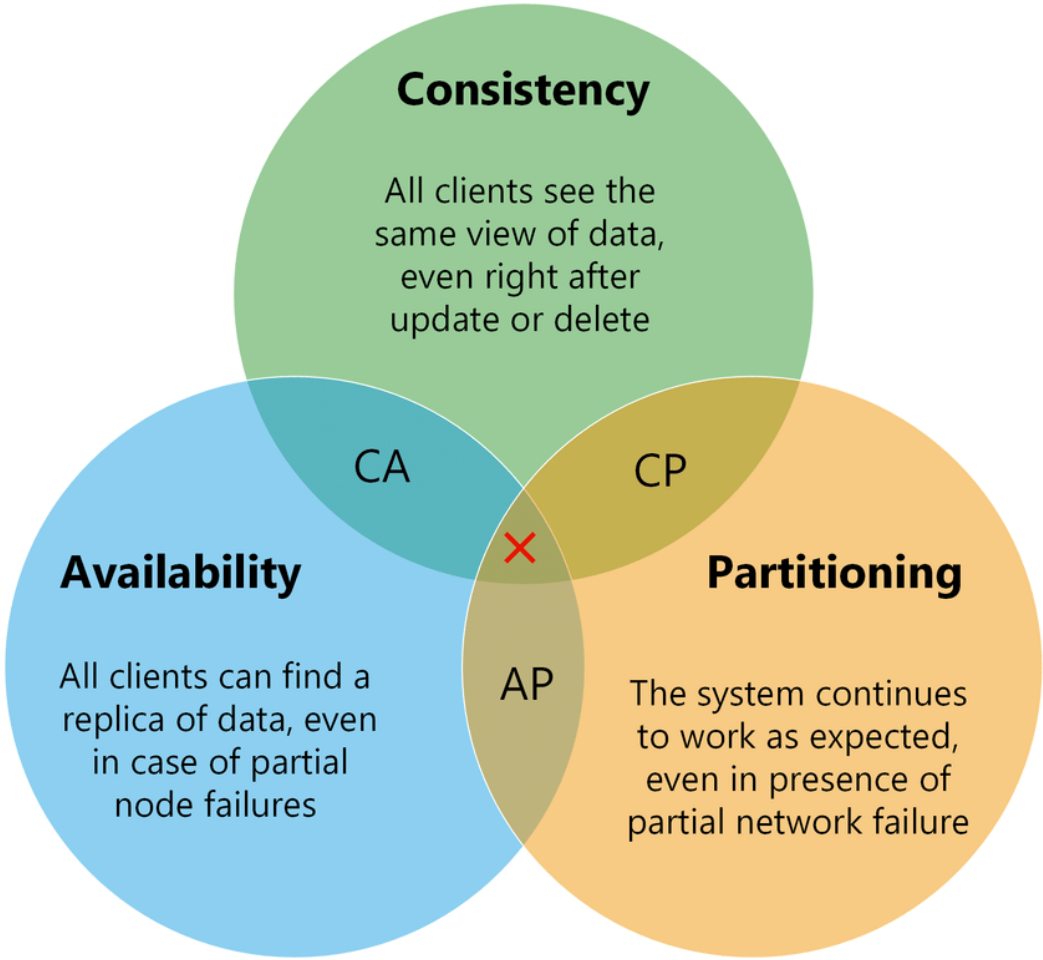
Putting everything together: the CAP theorem

PUTTING EVERYTHING TOGETHER: THE CAP THEOREM

- Conjecture by Eric Brewer in 2000
- Proposed formal proof by Seth Gilbert and Nancy Lynch in 2002

- No distributed system can simultaneously provide consistency, availability, partition tolerance

- Since partition tolerance is mandatory, the main tradeoff is between consistency and availability



consistency

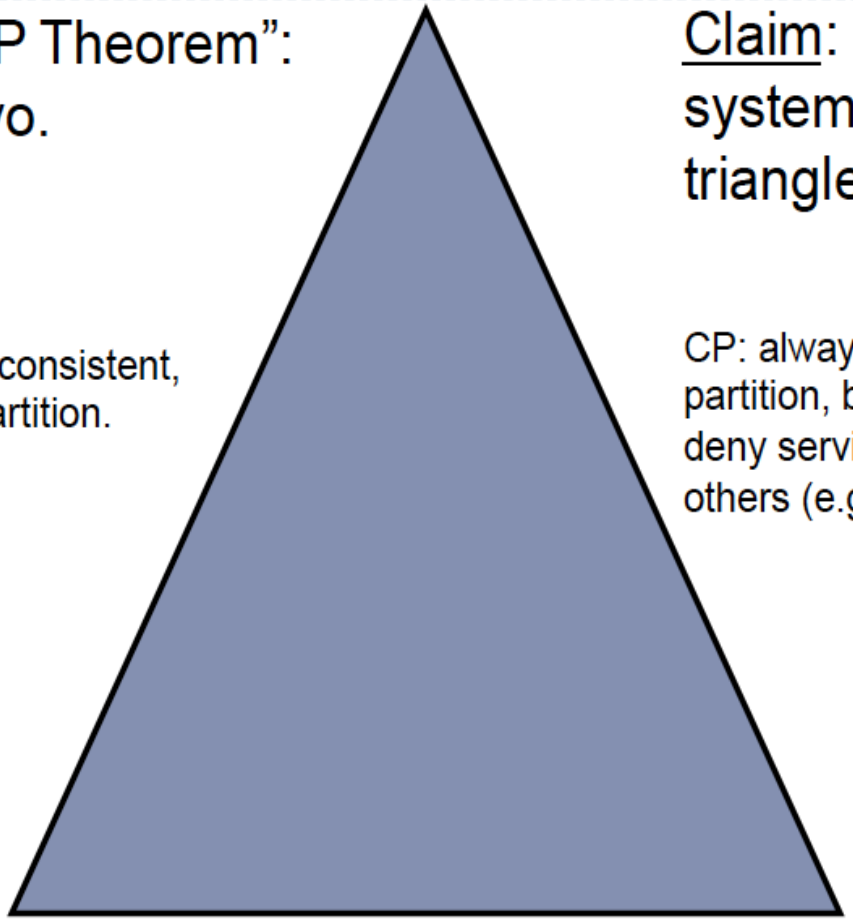
C

Fox&Brewer “CAP Theorem”:
C-A-P: choose two.

Claim: every distributed system is on one side of the triangle.

CA: available, and consistent, unless there is a partition.

CP: always consistent, even in a partition, but a reachable replica may deny service without agreement of the others (e.g., quorum).



A

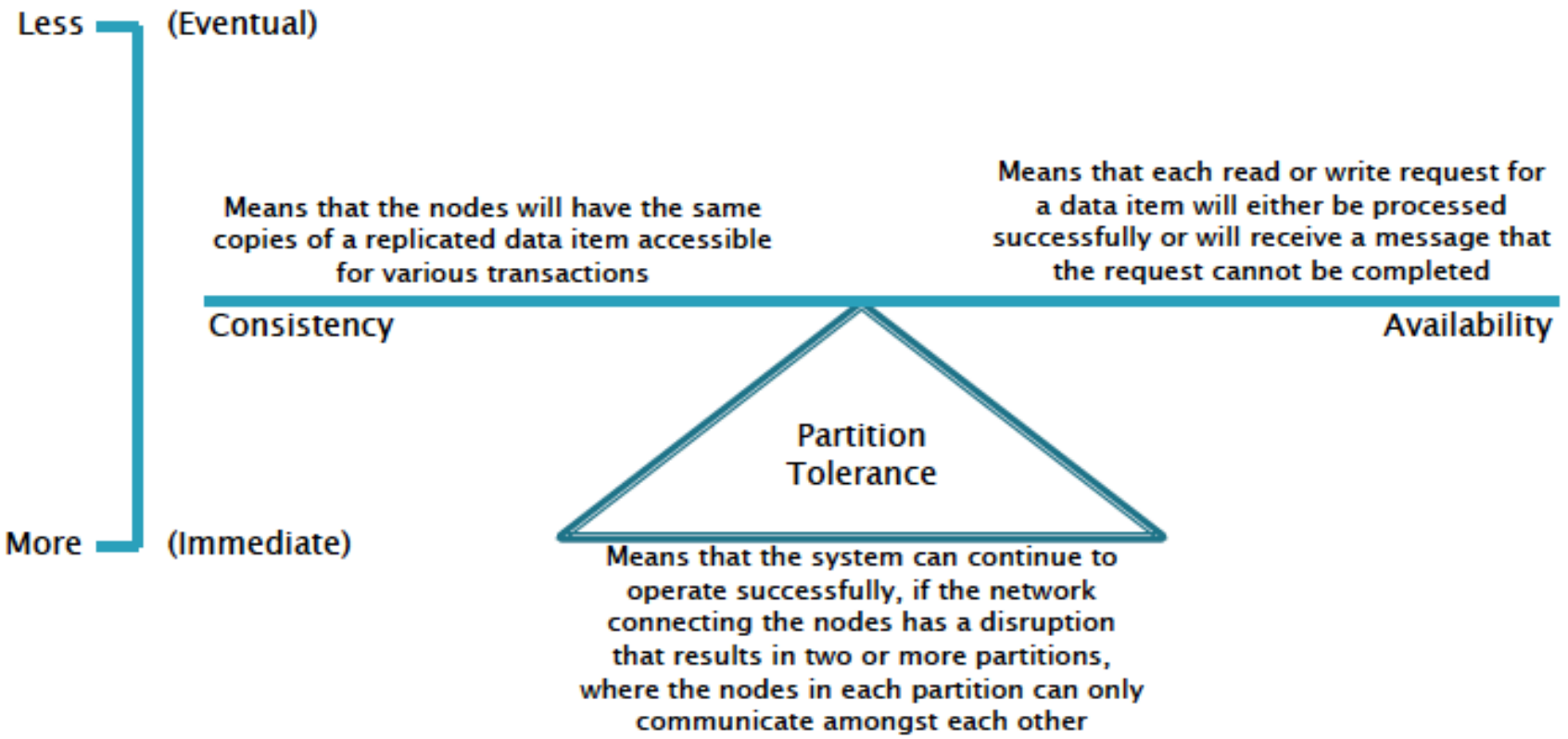
Availability

AP: a reachable replica provides service even in a partition, but may be inconsistent.

P

Partition-resilience

Consistency or Availability?



Take away

- The CAP theorem states that if you get a network partition, you have to trade off availability of data versus consistency
- Partition tolerance is mandatory in distributed systems, leading to either CP or AP systems
- Different levels of consistency and availability are however possible